

# IA159 Formal Verification Methods

## Static Analysis and Abstract Interpretation

Jan Strejček

Faculty of Informatics  
Masaryk University

## Focus

- lattices and fixpoints
- static analysis
- abstract interpretation

## Source

- P. Cousot and R. Cousot: *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, POPL 1977.

Special thanks to Marek Trtík for providing me his slides.

## Floyd's conjecture

To prove static properties of program it is often sufficient to consider sets of states associated with each program point.

### Examples

- to check safety properties (reachability of an error state), one only needs to know reachable states
- for many optimizations during compilation, static information is sufficient (e.g. detection of live variables, available expressions, etc.)

## Operational semantics

- defines how a state changes along program execution
- it is concerned about **computational sequences**
- computes a function relating input and output states

# Motivation for static analysis

## Operational semantics

- defines how a state changes along program execution
- it is concerned about **computational sequences**
- computes a function relating input and output states

## Static semantic

- observes which states pass which program location
- it is concerned about **observed sets of states at locations**
- computes a function assigning set of states to each program location

# Motivation for abstract interpretation

- It is usually impossible to compute the sets of reachable states precisely
- we can compute them on some level of abstraction
- for example, instead with precise numbers we work only with abstract values  $\{+, 0, -\}$
- abstraction brings some level of imprecision, for example,  $15 - 17$  is seen as  $(+) - (+)$ , which can be  $+, 0, -$

## Lattices and fixpoints

Let  $(L, \leq)$  be a partially ordered set and  $M \subseteq L$ .

- $x \in L$  is an **upper bound** of  $M$  iff  $y \leq x$  holds for all  $y \in M$
- $x \in L$  is a **lower bound** of  $M$  iff  $x \leq y$  holds for all  $y \in M$
- **supremum** of  $M$  is the least upper bound of  $M$
- **infimum** of  $M$  is the greatest lower bound of  $M$
- $\sup(M)$  and  $\inf(M)$  denote supremum and infimum of  $M$ , respectively



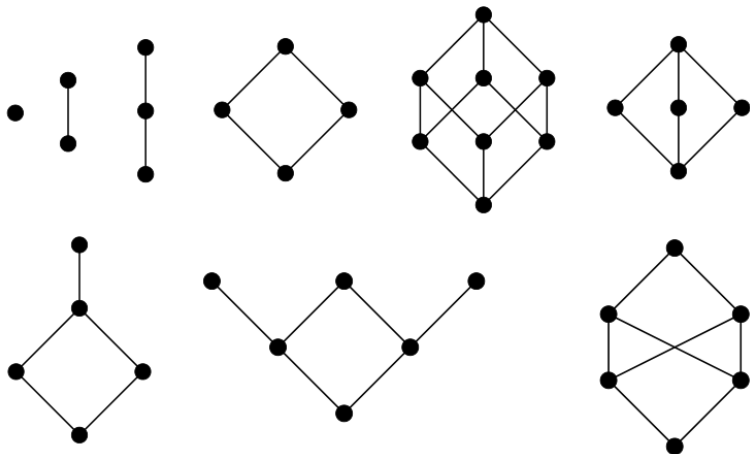
Let  $(L, \leq)$  be a partially ordered set and  $M \subseteq L$ .

- $x \in L$  is an **upper bound** of  $M$  iff  $y \leq x$  holds for all  $y \in M$
- $x \in L$  is a **lower bound** of  $M$  iff  $x \leq y$  holds for all  $y \in M$
- **supremum** of  $M$  is the least upper bound of  $M$
- **infimum** of  $M$  is the greatest lower bound of  $M$
- $\sup(M)$  and  $\inf(M)$  denote supremum and infimum of  $M$ , respectively

## Definition (Complete lattice)

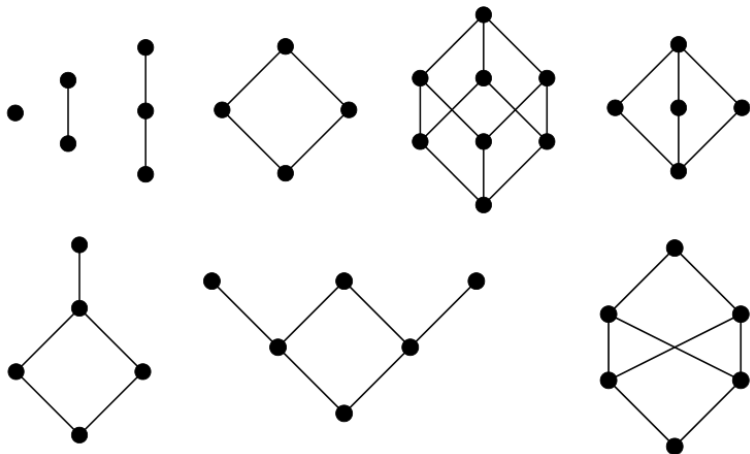
An ordered set  $(L, \leq)$  is called **complete lattice**, if for each  $M \subseteq L$  there exist both  $\sup(M)$  and  $\inf(M)$ .

# Introduction to lattices



Which of the partially ordered sets are complete lattices?

# Introduction to lattices

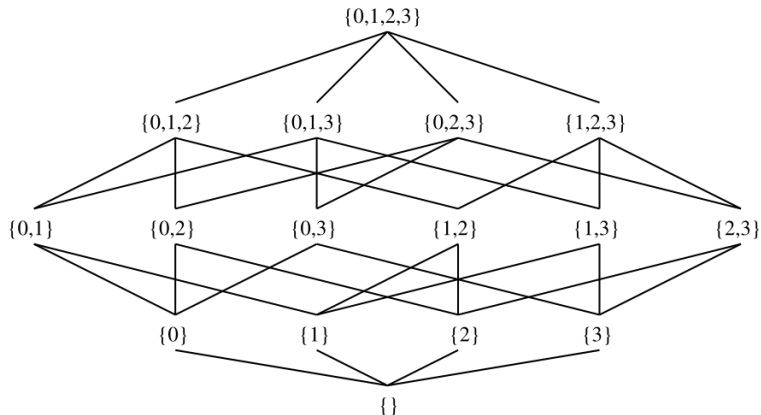


Which of the partially ordered sets are complete lattices?  
(All of the top row and the left of the bottom row.)

# Introduction to lattices

For every set  $S$ , the powerset  $\mathcal{P}(S)$  with the partial order  $\subseteq$  is a complete lattice.

For example,  $(\mathcal{P}(\{0, 1, 2, 3\}), \subseteq)$  looks like:



Let  $(L, \leq)$  be a complete lattice.

- the greatest element  $\top = \text{sup}(L)$  is called **one** of  $L$
- the least element  $\perp = \text{inf}(L)$  of  $L$  is called **zero** of  $L$
- the lattice is of **finite height** if there exists  $h \in \mathbb{N}$  such that the length of each strictly increasing chain of elements of  $L$  is less than or equal to  $h$
- minimal such  $h$  is called **lattice height**

# Fixpoint and Knaster-Tarski fixpoint theorem

Let  $(L, \leq)$  be a complete lattice.

- a function  $f : L \rightarrow L$  is **monotone** if for all  $x, y \in L$  it holds

$$x \leq y \implies f(x) \leq f(y)$$

- $x \in L$  is called a **fixpoint** of  $f$  if  $f(x) = x$

# Fixpoint and Knaster-Tarski fixpoint theorem

Let  $(L, \leq)$  be a complete lattice.

- a function  $f : L \rightarrow L$  is **monotone** if for all  $x, y \in L$  it holds

$$x \leq y \implies f(x) \leq f(y)$$

- $x \in L$  is called a **fixpoint** of  $f$  if  $f(x) = x$

## Theorem (Knaster-Tarski)

*Let  $(L, \leq)$  be a complete lattice and  $f : L \rightarrow L$  be a monotone function. Then the set of fixpoints of  $f$  with partial order  $\leq$  is also a complete lattice.*

## Theorem (Kleene)

*Let  $(L, \leq)$  be a complete lattice of finite height and  $f : L \rightarrow L$  a monotone function. Then there exists  $n \in \mathbb{N}$  such that for all  $k \in \mathbb{N}$  it is  $f^n(\perp) = f^{n+k}(\perp)$  and  $f^n(\perp)$  is the least fixpoint of  $f$ .*



## Theorem (Kleene)

*Let  $(L, \leq)$  be a complete lattice of finite height and  $f : L \rightarrow L$  a monotone function. Then there exists  $n \in \mathbb{N}$  such that for all  $k \in \mathbb{N}$  it is  $f^n(\perp) = f^{n+k}(\perp)$  and  $f^n(\perp)$  is the least fixpoint of  $f$ .*

Proof: Since  $\perp$  is the least element of  $L$ , we have  $\perp \leq f(\perp)$ . Since  $f$  is monotone, then  $f(\perp) \leq f(f(\perp))$  and by induction  $f^i(\perp) \leq f^{i+1}(\perp)$ . Thus, we have a nondecreasing chain  $\perp \leq f(\perp) \leq f^2(\perp) \leq \dots$ . Since  $L$  is assumed to be of a finite height, there must exist  $n \in \mathbb{N}$  such that  $f^n(\perp) = f^{n+1}(\perp)$ . To show that  $f^n(\perp)$  is a least fixpoint of  $f$ , let us assume  $x$  is another fixpoint of  $f$ . Since  $\perp \leq x$  and  $f(\perp) \leq f(x) = x$  from monotonicity of  $f$ , we get by induction  $f^n(\perp) \leq x$ . □

# Fixpoint computation

## Algorithm for the least fixpoint computation

```
x :=  $\perp$ ;  
do { t := x; x := f(x); } while (x  $\neq$  t);
```

If we start with  $x := \top$ , we get the greatest fixpoint.

## Lemma (Product lattice)

*Let  $(L_1, \leq_1), \dots, (L_n, \leq_n)$  be complete lattices and order  $\leq$  on  $L_1 \times \dots \times L_n$  is defined as  $(x_1, \dots, x_n) \leq (y_1, \dots, y_n)$  iff*

$$x_1 \leq_1 y_1 \wedge \dots \wedge x_n \leq_n y_n.$$

*Then  $(L_1 \times \dots \times L_n, \leq)$  is a complete lattice.*

# Fixpoints on product lattices

Let  $(L, \leq)$  be a complete lattice and  $(L^n, \sqsubseteq)$  be the corresponding product lattice. Further, let  $F_1, \dots, F_n : L^n \rightarrow L$  be monotone functions, i.e.  $(x_1, \dots, x_n) \sqsubseteq (y_1, \dots, y_n)$  implies  $F_i(x_1, \dots, x_n) \leq F_i(y_1, \dots, y_n)$  for each  $1 \leq i \leq n$ . Then the function  $F : L^n \rightarrow L^n$  defined as

$$F(x_1, \dots, x_n) = (F_1(x_1, \dots, x_n), \dots, F_n(x_1, \dots, x_n))$$

is a monotone function in  $(L^n, \sqsubseteq)$ . Further, the least fixpoint of  $F$  is the least solution of the system

$$\begin{aligned}x_1 &= F_1(x_1, \dots, x_n) \\ &\vdots \\ x_n &= F_n(x_1, \dots, x_n)\end{aligned}$$

# Fixpoint computation of product lattices

## Naive algorithm for fixpoint computation

```
 $\vec{X} := \vec{\perp};$   
do {  $\vec{t} := \vec{X}; \vec{X} := F(\vec{X});$  } while  $(\vec{X} \neq \vec{t});$ 
```

# Fixpoint computation of product lattices

## Naive algorithm for fixpoint computation

```
 $\vec{x} := \perp;$   
do {  $\vec{t} := \vec{x}; \vec{x} := F(\vec{x});$  } while  $(\vec{x} \neq \vec{t});$ 
```

## Better algorithm for fixpoint computation (faster convergence)

```
 $x_1 := \perp; \dots x_n := \perp;$   
do {  
     $t_1 := x_1; \dots t_n := x_n;$   
     $x_1 := F_1(x_1, \dots, x_n);$   
     $\vdots$   
     $x_n := F_n(x_1, \dots, x_n);$   
} while  $(x_1 \neq t_1 \vee \dots \vee x_n \neq t_n);$ 
```

Abstract interpretation

# Abstract interpretation

- an abstract interpretation of a program is kind of a **static semantic**, where original data domains are replaced with **abstract** ones
- abstract data domain must constitute a **complete lattice**
- semantic of program instructions have to be changed as well: we define unique **monotone function** for each program instruction



## Definition (Abstract interpretation)

An **abstract interpretation**  $I$  of a program  $P$  with  $n$  program locations is a tuple

$$I = \langle L, \circ, \leq, \top, \perp, F \rangle$$

where  $(L, \leq)$  is complete lattice,  $\top$  and  $\perp$  are one and zero of  $(L, \leq)$ ,  $\circ$  is equal either to join or meet operation, and  $F$  is a monotone function on product lattice  $(L^n, \leq)$  defining the interpretation of basic instructions.

The meet operator is defined as  $a \circ b = \inf(\{a, b\})$ , while the join operator is defined as  $a \circ b = \sup(\{a, b\})$ .

# Abstract interpretation: Definition

## Definition (Abstract interpretation)

An **abstract interpretation**  $I$  of a program  $P$  with  $n$  program locations is a tuple

$$I = \langle L, \circ, \leq, \top, \perp, F \rangle$$

where  $(L, \leq)$  is complete lattice,  $\top$  and  $\perp$  are one and zero of  $(L, \leq)$ ,  $\circ$  is equal either to join or meet operation, and  $F$  is a monotone function on product lattice  $(L^n, \leq)$  defining the interpretation of basic instructions.

The meet operator is defined as  $a \circ b = \inf(\{a, b\})$ , while the join operator is defined as  $a \circ b = \sup(\{a, b\})$ .

Typically,  $F(\vec{x}) = (F_1(\vec{x}), \dots, F_n(\vec{x}))$ , where each  $F_i : L^n \rightarrow L$  defines effect of  $i$ -th program instruction.

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

```
var x, y, z, a, b;  
z := a+b;  
y := a*b;  
while (y > a+b) {  
    a := a+1;  
    x := a+b;  
}
```

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

Available expressions:  $AExprs = \{a+b, a*b, y > a+b, a+1\}$

```
var x, y, z, a, b;  
z := a+b;  
y := a*b;  
while (y > a+b) {  
    a := a+1;  
    x := a+b;  
}
```

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

Available expressions:  $AExprs = \{a+b, a*b, y > a+b, a+1\}$

A.I.:  $I = \langle \mathcal{P}(AExprs), \cap, \subseteq, AExprs, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_6(\vec{x})) \rangle$

```
var x, y, z, a, b;  
z := a+b;  
y := a*b;  
while (y > a+b) {  
    a := a+1;  
    x := a+b;  
}
```

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

Available expressions:  $AExprs = \{a+b, a*b, y>a+b, a+1\}$

A.I.:  $I = \langle \mathcal{P}(AExprs), \cap, \subseteq, AExprs, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_6(\vec{x})) \rangle$

Product lattice:  $(\mathcal{P}^6(AExprs), \leq)$ .

```
var x, y, z, a, b;      X1
z := a+b;              X2
y := a*b;              X3
while (y > a+b) {     X4
    a := a+1;         X5
    x := a+b;         X6
}
```

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

Available expressions:  $AExprs = \{a+b, a*b, y>a+b, a+1\}$

A.I.:  $I = \langle \mathcal{P}(AExprs), \cap, \subseteq, AExprs, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_6(\vec{x})) \rangle$

Product lattice:  $(\mathcal{P}^6(AExprs), \leq)$ .

<code>var x, y, z, a, b;</code>	$x_1 = F_1(\vec{x}) = \emptyset$
<code>z := a+b;</code>	$x_2 = F_2(\vec{x}) = (x_1 \cup \{a+b\}) \setminus \emptyset$
<code>y := a*b;</code>	$x_3 = F_3(\vec{x}) = (x_2 \cup \{a*b\}) \setminus \{y>a+b\}$
<code>while (y &gt; a+b) {</code>	$x_4 = F_4(\vec{x}) = (x_3 \cap x_6) \cup \{a+b, y>a+b\}$
<code>a := a+1;</code>	$x_5 = F_5(\vec{x}) = (x_4 \cup \{a+1\}) \setminus AExprs$
<code>x := a+b;</code>	$x_6 = F_6(\vec{x}) = (x_5 \cup \{a+b\}) \setminus \emptyset$
<code>}</code>	



## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

Available expressions:  $AExprs = \{a+b, a*b, y>a+b, a+1\}$

A.I.:  $I = \langle \mathcal{P}(AExprs), \cap, \subseteq, AExprs, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_6(\vec{x})) \rangle$

Product lattice:  $(\mathcal{P}^6(AExprs), \leq)$ .

<code>var x, y, z, a, b;</code>	$x_1 = F_1(\vec{x}) = \emptyset$
<code>z := a+b;</code>	$x_2 = F_2(\vec{x}) = (x_1 \cup \{a+b\}) \setminus \emptyset$
<code>y := a*b;</code>	$x_3 = F_3(\vec{x}) = (x_2 \cup \{a*b\}) \setminus \{y>a+b\}$
<code>while (y &gt; a+b) {</code>	$x_4 = F_4(\vec{x}) = (x_3 \cap x_6) \cup \{a+b, y>a+b\}$
<code>a := a+1;</code>	$x_5 = F_5(\vec{x}) = (x_4 \cup \{a+1\}) \setminus AExprs$
<code>x := a+b;</code>	$x_6 = F_6(\vec{x}) = (x_5 \cup \{a+b\}) \setminus \emptyset$
<code>}</code>	

**Direction: Forward**

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

Available expressions:  $AExprs = \{a+b, a*b, y>a+b, a+1\}$

A.I.:  $I = \langle \mathcal{P}(AExprs), \cap, \subseteq, AExprs, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_6(\vec{x})) \rangle$

Product lattice:  $(\mathcal{P}^6(AExprs), \leq)$ .

<code>var x, y, z, a, b;</code>	$x_1 = F_1(\vec{x}) = \emptyset$
<code>z := a+b;</code>	$x_2 = F_2(\vec{x}) = (x_1 \cup \{a+b\}) \setminus \emptyset$
<code>y := a*b;</code>	$x_3 = F_3(\vec{x}) = (x_2 \cup \{a*b\}) \setminus \{y>a+b\}$
<code>while (y &gt; a+b) {</code>	$x_4 = F_4(\vec{x}) = (x_3 \cap x_6) \cup \{a+b, y>a+b\}$
<code>a := a+1;</code>	$x_5 = F_5(\vec{x}) = (x_4 \cup \{a+1\}) \setminus AExprs$
<code>x := a+b;</code>	$x_6 = F_6(\vec{x}) = (x_5 \cup \{a+b\}) \setminus \emptyset$
<code>}</code>	

**Analysis: Must**

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

Available expressions:  $AExprs = \{a+b, a*b, y>a+b, a+1\}$

A.I.:  $I = \langle \mathcal{P}(AExprs), \cap, \subseteq, AExprs, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_6(\vec{x})) \rangle$

Product lattice:  $(\mathcal{P}^6(AExprs), \leq)$ .

<code>var x, y, z, a, b;</code>	$x_1 = F_1(\vec{x}) = \emptyset$
<code>z := a+b;</code>	$x_2 = F_2(\vec{x}) = (x_1 \cup \{a+b\}) \setminus \emptyset$
<code>y := a*b;</code>	$x_3 = F_3(\vec{x}) = (x_2 \cup \{a*b\}) \setminus \{y>a+b\}$
<code>while (y &gt; a+b) {</code>	$x_4 = F_4(\vec{x}) = (x_3 \cap x_6) \cup \{a+b, y>a+b\}$
<code>a := a+1;</code>	$x_5 = F_5(\vec{x}) = (x_4 \cup \{a+1\}) \setminus AExprs$
<code>x := a+b;</code>	$x_6 = F_6(\vec{x}) = (x_5 \cup \{a+b\}) \setminus \emptyset$
<code>}</code>	

Are all functions  $F_i$  monotone?

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

Available expressions:  $AExprs = \{a+b, a*b, y>a+b, a+1\}$

A.I.:  $I = \langle \mathcal{P}(AExprs), \cap, \subseteq, AExprs, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_6(\vec{x})) \rangle$

Product lattice:  $(\mathcal{P}^6(AExprs), \leq)$ .

<code>var x, y, z, a, b;</code>	$x_1 = F_1(\vec{x}) = \emptyset$
<code>z := a+b;</code>	$x_2 = F_2(\vec{x}) = (x_1 \cup \{a+b\}) \setminus \emptyset$
<code>y := a*b;</code>	$x_3 = F_3(\vec{x}) = (x_2 \cup \{a*b\}) \setminus \{y>a+b\}$
<code>while (y &gt; a+b) {</code>	$x_4 = F_4(\vec{x}) = (x_3 \cap x_6) \cup \{a+b, y>a+b\}$
<code>a := a+1;</code>	$x_5 = F_5(\vec{x}) = (x_4 \cup \{a+1\}) \setminus AExprs$
<code>x := a+b;</code>	$x_6 = F_6(\vec{x}) = (x_5 \cup \{a+b\}) \setminus \emptyset$
<code>}</code>	

**Proof  $F_4$ :** Let  $\vec{x}, \vec{y} \in \mathcal{P}^6(AExprs)$  such that  $\vec{x} \leq \vec{y}$ . ...

## Example: Available expressions

A nontrivial expression in a program is **available** at a program location if its current value has already been computed earlier in the execution.

Available expressions:  $AExprs = \{a+b, a*b, y>a+b, a+1\}$

A.I.:  $I = \langle \mathcal{P}(AExprs), \cap, \subseteq, AExprs, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_6(\vec{x})) \rangle$

Product lattice:  $(\mathcal{P}^6(AExprs), \leq)$ .

<code>var x, y, z, a, b;</code>	$x_1 = F_1(\vec{x}) = \emptyset$
<code>z := a+b;</code>	$x_2 = F_2(\vec{x}) = (x_1 \cup \{a+b\}) \setminus \emptyset$
<code>y := a*b;</code>	$x_3 = F_3(\vec{x}) = (x_2 \cup \{a*b\}) \setminus \{y>a+b\}$
<code>while (y &gt; a+b) {</code>	$x_4 = F_4(\vec{x}) = (x_3 \cap x_6) \cup \{a+b, y>a+b\}$
<code>a := a+1;</code>	$x_5 = F_5(\vec{x}) = (x_4 \cup \{a+1\}) \setminus AExprs$
<code>x := a+b;</code>	$x_6 = F_6(\vec{x}) = (x_5 \cup \{a+b\}) \setminus \emptyset$
<code>}</code>	

Then  $x_3 \subseteq y_3$  and  $x_6 \subseteq y_6$ , which implies  $(x_3 \cap x_6) \subseteq (y_3 \cap y_6) \dots$

# Example: Available expressions

After fixpoint computation ...

<code>var x, y, z, a, b;</code>	$x_1 = \emptyset$
<code>z := a+b;</code>	$x_2 = \{a+b\}$
<code>y := a*b;</code>	$x_3 = \{a+b, a*b\}$
<code>while (y &gt; a+b) {</code>	$x_4 = \{a+b, y > a+b\}$
<code>a := a+1;</code>	$x_5 = \emptyset$
<code>x := a+b;</code>	$x_6 = \{a+b\}$
<code>}</code>	

**Solution: Minimal**

# Example: Available expressions

After fixpoint computation ...

<code>var x, y, z, a, b;</code>	$x_1 = \emptyset$
<code>z := a+b;</code>	$x_2 = \{a+b\}$
<code>y := a*b;</code>	$x_3 = \{a+b, a*b\}$
<code>while (y &gt; a+b) {</code>	$x_4 = \{a+b, y > a+b\}$
<code>a := a+1;</code>	$x_5 = \emptyset$
<code>x := a+b;</code>	$x_6 = \{a+b\}$
<code>}</code>	

## Example: Live variables

A variable is **live** at a program point if its current value may be read during the remaining execution of the program.

```
var x, y, z;  
x := input;  
while (x > 1) {  
    y := x/2;  
    if (y > 3)  
        x := x - y;  
    z := x - 4;  
    if (z > 0)  
        x := x/2;  
    z := z - 1; }  
output x;
```



## Example: Live variables

A variable is **live** at a program point if its current value may be read during the remaining execution of the program.

$$\text{Vars} = \{x, y, z\} \text{ and} \\ I = \langle \mathcal{P}(\text{Vars}), \cup, \subseteq, \text{Vars}, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_{11}(\vec{x})) \rangle$$

```
var x, y, z;
x := input;
while (x > 1) {
  y := x / 2;
  if (y > 3)
    x := x - y;
  z := x - 4;
  if (z > 0)
    x := x / 2;
  z := z - 1; }
output x;
```

## Example: Live variables

A variable is **live** at a program point if its current value may be read during the remaining execution of the program.

Product lattice is  $(\mathcal{P}^{11}(\text{Vars}), \leq)$ .

$X_1 = X_2 \setminus \{x, y, z\}$	<code>var x, y, z;</code>
$X_2 = X_3 \setminus \{x\}$	<code>x := input;</code>
$X_3 = (X_4 \cup X_{11}) \cup \{x\}$	<code>while (x&gt;1) {</code>
$X_4 = (X_5 \setminus \{y\}) \cup \{x\}$	<code>    y := x/2;</code>
$X_5 = (X_6 \cup X_7) \cup \{y\}$	<code>    if (y&gt;3)</code>
$X_6 = (X_7 \setminus \{x\}) \cup \{x, y\}$	<code>        x := x-y;</code>
$X_7 = (X_8 \setminus \{z\}) \cup \{x\}$	<code>        z := x-4;</code>
$X_8 = (X_9 \cup X_{10}) \cup \{z\}$	<code>        if (z&gt;0)</code>
$X_9 = (X_{10} \setminus \{x\}) \cup \{x\}$	<code>            x := x/2;</code>
$X_{10} = (X_3 \setminus \{z\}) \cup \{z\}$	<code>            z := z-1; }</code>
$X_{11} = \{x\}$	<code>output x;</code>

## Example: Live variables

A variable is **live** at a program point if its current value may be read during the remaining execution of the program.

Direction: Backward

$X_1 = X_2 \setminus \{x, y, z\}$	<code>var x, y, z;</code>
$X_2 = X_3 \setminus \{x\}$	<code>x := input;</code>
$X_3 = (X_4 \cup X_{11}) \cup \{x\}$	<code>while (x&gt;1) {</code>
$X_4 = (X_5 \setminus \{y\}) \cup \{x\}$	<code>    y := x/2;</code>
$X_5 = (X_6 \cup X_7) \cup \{y\}$	<code>    if (y&gt;3)</code>
$X_6 = (X_7 \setminus \{x\}) \cup \{x, y\}$	<code>        x := x-y;</code>
$X_7 = (X_8 \setminus \{z\}) \cup \{x\}$	<code>        z := x-4;</code>
$X_8 = (X_9 \cup X_{10}) \cup \{z\}$	<code>        if (z&gt;0)</code>
$X_9 = (X_{10} \setminus \{x\}) \cup \{x\}$	<code>            x := x/2;</code>
$X_{10} = (X_3 \setminus \{z\}) \cup \{z\}$	<code>            z := z-1; }</code>
$X_{11} = \{x\}$	<code>output x;</code>

# Example: Live variables

A variable is **live** at a program point if its current value may be read during the remaining execution of the program.

## Analysis: May

$X_1 = X_2 \setminus \{x, y, z\}$	<code>var x, y, z;</code>
$X_2 = X_3 \setminus \{x\}$	<code>x := input;</code>
$X_3 = (X_4 \cup X_{11}) \cup \{x\}$	<code>while (x&gt;1) {</code>
$X_4 = (X_5 \setminus \{y\}) \cup \{x\}$	<code>    y := x/2;</code>
$X_5 = (X_6 \cup X_7) \cup \{y\}$	<code>    if (y&gt;3)</code>
$X_6 = (X_7 \setminus \{x\}) \cup \{x, y\}$	<code>        x := x-y;</code>
$X_7 = (X_8 \setminus \{z\}) \cup \{x\}$	<code>        z := x-4;</code>
$X_8 = (X_9 \cup X_{10}) \cup \{z\}$	<code>        if (z&gt;0)</code>
$X_9 = (X_{10} \setminus \{x\}) \cup \{x\}$	<code>            x := x/2;</code>
$X_{10} = (X_3 \setminus \{z\}) \cup \{z\}$	<code>            z := z-1; }</code>
$X_{11} = \{x\}$	<code>output x;</code>

# Example: Live variables

A variable is **live** at a program point if its current value may be read during the remaining execution of the program.

## Solution: Minimal

$X_1 = \emptyset$	$X_1 = X_2 \setminus \{x, y, z\}$	<code>var x, y, z;</code>
$X_2 = \emptyset$	$X_2 = X_3 \setminus \{x\}$	<code>x := input;</code>
$X_3 = \{x\}$	$X_3 = (X_4 \cup X_{11}) \cup \{x\}$	<code>while (x&gt;1) {</code>
$X_4 = \{x\}$	$X_4 = (X_5 \setminus \{y\}) \cup \{x\}$	<code>    y := x/2;</code>
$X_5 = \{x, y\}$	$X_5 = (X_6 \cup X_7) \cup \{y\}$	<code>        if (y&gt;3)</code>
$X_6 = \{x, y\}$	$X_6 = (X_7 \setminus \{x\}) \cup \{x, y\}$	<code>            x := x-y;</code>
$X_7 = \{x\}$	$X_7 = (X_8 \setminus \{z\}) \cup \{x\}$	<code>            z := x-4;</code>
$X_8 = \{x, z\}$	$X_8 = (X_9 \cup X_{10}) \cup \{z\}$	<code>                if (z&gt;0)</code>
$X_9 = \{x, z\}$	$X_9 = (X_{10} \setminus \{x\}) \cup \{x\}$	<code>                    x := x/2;</code>
$X_{10} = \{x, z\}$	$X_{10} = (X_3 \setminus \{z\}) \cup \{z\}$	<code>                    z := z-1; }</code>
$X_{11} = \{x\}$	$X_{11} = \{x\}$	<code>output x;</code>

## Example: Live variables

A variable is **live** at a program point if its current value may be read during the remaining execution of the program.

Variables  $y, z$  are never live together.

$X_1 = \emptyset$	$X_1 = X_2 \setminus \{x, y, z\}$	var $x, y, z$ ;
$X_2 = \emptyset$	$X_2 = X_3 \setminus \{x\}$	$x := \text{input};$
$X_3 = \{x\}$	$X_3 = (X_4 \cup X_{11}) \cup \{x\}$	while ( $x > 1$ ) {
$X_4 = \{x\}$	$X_4 = (X_5 \setminus \{y\}) \cup \{x\}$	$y := x/2$ ;
$X_5 = \{x, y\}$	$X_5 = (X_6 \cup X_7) \cup \{y\}$	if ( $y > 3$ )
$X_6 = \{x, y\}$	$X_6 = (X_7 \setminus \{x\}) \cup \{x, y\}$	$x := x - y$ ;
$X_7 = \{x\}$	$X_7 = (X_8 \setminus \{z\}) \cup \{x\}$	$z := x - 4$ ;
$X_8 = \{x, z\}$	$X_8 = (X_9 \cup X_{10}) \cup \{z\}$	if ( $z > 0$ )
$X_9 = \{x, z\}$	$X_9 = (X_{10} \setminus \{x\}) \cup \{x\}$	$x := x/2$ ;
$X_{10} = \{x, z\}$	$X_{10} = (X_3 \setminus \{z\}) \cup \{z\}$	$z := z - 1$ ; }
$X_{11} = \{x\}$	$X_{11} = \{x\}$	output $x$ ;

## Example: Reaching definitions

The **reaching definitions** for a given program point are those assignments that may have defined the current values of variables.

```
var x,y,z;  
x := input;  
while (x>1) {  
    y := x/2;  
    if (y>3)  
        x := x-y;  
    z := x-4;  
    if (z>0)  
        x := x/2;  
    z := z-1; }  
output x;
```

## Example: Reaching definitions

The **reaching definitions** for a given program point are those assignments that may have defined the current values of variables.

```
var x, y, z;  
x := input;  
while (x>1) {  
  y := x/2;  
  if (y>3)  
    x := x-y;  
  z := x-4;  
  if (z>0)  
    x := x/2;  
  z := z-1; }  
output x;
```

Assignments:

*Asgns* = {x=input, y=x/2, x=x-y,  
z=x-4, x=x/2, z=z-1}



## Example: Reaching definitions

The **reaching definitions** for a given program point are those assignments that may have defined the current values of variables.

```
var x, y, z;  
x := input;  
while (x>1) {  
  y := x/2;  
  if (y>3)  
    x := x-y;  
  z := x-4;  
  if (z>0)  
    x := x/2;  
  z := z-1; }  
output x;
```

Assignments:

$$\mathit{Asgns} = \{x=\text{input}, y=x/2, x=x-y, \\ z=x-4, x=x/2, z=z-1\}$$
$$I = \langle \mathcal{P}(\mathit{Asgns}), \cup, \subseteq, \mathit{Asgns}, \emptyset, \\ \lambda \vec{x}. (F_1(\vec{x}), \dots, F_{11}(\vec{x})) \rangle$$

## Example: Reaching definitions

The **reaching definitions** for a given program point are those assignments that may have defined the current values of variables.

```
var x, y, z;  
x := input;  
while (x > 1) {  
  y := x/2;  
  if (y > 3)  
    x := x - y;  
  z := x - 4;  
  if (z > 0)  
    x := x/2;  
  z := z - 1; }  
output x;
```

Assignments:

$Asgns = \{x=input, y=x/2, x=x-y, z=x-4, x=x/2, z=z-1\}$

$I = \langle \mathcal{P}(Asgns), \cup, \subseteq, Asgns, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_{11}(\vec{x})) \rangle$

Product lattice:  $(\mathcal{P}^{11}(Asgns), \subseteq)$

## Example: Reaching definitions

The **reaching definitions** for a given program point are those assignments that may have defined the current values of variables.

```
var x, y, z;
x := input;
while (x>1) {
  y := x/2;
  if (y>3)
    x := x-y;
  z := x-4;
  if (z>0)
    x := x/2;
  z := z-1; }
output x;
```

Assignments:

$Asgns = \{x=input, y=x/2, x=x-y, z=x-4, x=x/2, z=z-1\}$

$I = \langle \mathcal{P}(Asgns), \cup, \subseteq, Asgns, \emptyset, \lambda \vec{x}. (F_1(\vec{x}), \dots, F_{11}(\vec{x})) \rangle$

Product lattice:  $(\mathcal{P}^{11}(Asgns), \subseteq)$

Direction: Forward

Analysis: May

Solution: Minimal

## Example: Busy expressions

An expression is **busy** if it will definitely be evaluated again before its value changes.

## Example: Busy expressions

An expression is **busy** if it will definitely be evaluated again before its value changes.

Direction: Backward  
Analysis: Must  
Solution: Minimal

We may consider different abstraction levels of variable values:

- sets of integer values:  $\mathcal{P}(\mathbb{Z})$
- intervals:  $\{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, \infty\}, l \leq u\} \cup \{\perp\}$
- only signs with zero:  $\mathcal{P}(\{-, 0, +\})$
- initialized or not:  $\{\perp, \top\}$

We may consider different abstraction levels of variable values:

- sets of integer values:  $\mathcal{P}(\mathbb{Z})$
- intervals:  $\{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, \infty\}, l \leq u\} \cup \{\perp\}$
- only signs with zero:  $\mathcal{P}(\{-, 0, +\})$
- initialized or not:  $\{\perp, \top\}$

Which abstraction is more precise than other?

Widening and narrowing



When the extreme fixpoints of the system of equations cannot be computed in finitely many steps, they can be approximated.

Generally, we have these two approaches:

- 1 we can find more abstract interpretation
- 2 we can make approximations in the current interpretation to accelerate convergence of Kleene's sequence

Here we are concerned about second approach – the technique called **widening**.

Widening makes Kleene's sequence to converge

- to a fixpoint possibly greater than the least one or
- to an element  $s$ , such that  $s > F(s)$ .

In the second case, since  $s$  is greater than the least fixpoint, we can use **narrowing** to make the solution more precise – i.e. to find some fixpoint smaller than  $s$  but possibly greater than the least fixpoint.

- If the Kleene's sequence does not converge, then there exists a location  $x_i$  on a program loop where the sequence does not converge.
- We need a **widening function**  $\nabla : L \times L \rightarrow L$ , which is applied every time the location  $x_i$  is updated:  $x_i = x_i \nabla F_i(\vec{x})$ .
- We must define  $\nabla$  such that
  - for each  $x, y \in L$ ,  $x \circ y \leq x \nabla y$ , i.e.  $\nabla$  overapproximates operation  $\circ$ ,
  - it ensures that every infinite sequence of elements occurring in  $x_i$  is not strictly increasing.

# Widening

Example: Interval bounds of integer variable  $x$

```
{locations are after}
1  x := 1;
2  while (x <= 100) {
3      x := x + 1;
4  }
```

# Widening

Example: Interval bounds of integer variable  $x$

{locations are after}

```
1  x := 1;
2  while (x <= 100) {
3      x := x + 1;
4  }
```

{functions}

```
 $x_1 = [1, 1]$ 
 $x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$ 
 $x_3 = x_2 + [1, 1]$ 
 $x_4 = (x_1 \cup x_3) \cap [101, \infty]$ 
```

# Widening

Example: Interval bounds of integer variable  $x$

{locations after}	{functions}
1 $x := 1;$	$x_1 = [1, 1]$
2 $\text{while } (x \leq 100) \{$	$x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$
3 $x := x + 1;$	$x_3 = x_2 + [1, 1]$
4 $\}$	$x_4 = (x_1 \cup x_3) \cap [101, \infty]$

Widening operator  $\nabla$ :

$$[i, j] \nabla [k, l] = [\text{ite}(k < i, -\infty, i), \text{ite}(l > j, \infty, j)]$$

# Widening

Example: Interval bounds of integer variable  $x$

{locations after}	{functions}
1 $x := 1;$	$x_1 = [1, 1]$
2 while ( $x \leq 100$ ) {	$x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$
3 $x := x + 1;$	$x_3 = x_2 + [1, 1]$
4 }	$x_4 = (x_1 \cup x_3) \cap [101, \infty]$

Widening operator  $\nabla$ :

$[i, j] \nabla [k, l] = [ite(k < i, -\infty, i), ite(l > j, \infty, j)]$

{no widening}

$x_1 = [1, 1]$

$x_2 = [1, 100]$

$x_3 = [2, 101]$

$x_4 = [101, 101]$

100 iterations

# Widening

Example: Interval bounds of integer variable  $x$

{locations after}	{functions}
1 $x := 1;$	$x_1 = [1, 1]$
2 while ( $x \leq 100$ ) {	$x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$
3 $x := x + 1;$	$x_3 = x_2 + [1, 1]$
4 }	$x_4 = (x_1 \cup x_3) \cap [101, \infty]$

Widening operator  $\nabla$ :

$$[i, j] \nabla [k, l] = [\text{ite}(k < i, -\infty, i), \text{ite}(l > j, \infty, j)]$$

{no widening}	$\{x_3 = x_3 \nabla (x_2 + [1, 1])\}$
$x_1 = [1, 1]$	$x_1 = [1, 1]$
$x_2 = [1, 100]$	$x_2 = [1, 100]$
$x_3 = [2, 101]$	$x_3 = [2, \infty]$
$x_4 = [101, 101]$	$x_4 = [101, \infty]$
100 iterations	2 iterations



# Widening

Example: Interval bounds of integer variable  $x$

{locations after}	{functions}
1 $x := 1;$	$x_1 = [1, 1]$
2 $\text{while } (x \leq 100) \{$	$x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$
3 $x := x + 1;$	$x_3 = x_2 + [1, 1]$
4 $\}$	$x_4 = (x_1 \cup x_3) \cap [101, \infty]$

Widening operator  $\nabla$ :

$$[i, j] \nabla [k, l] = [\text{ite}(k < i, -\infty, i), \text{ite}(l > j, \infty, j)]$$

{no widening}	$\{x_3 = x_3 \nabla (x_2 + [1, 1])\}$
$x_1 = [1, 1]$	$x_1 = [1, 1]$
$x_2 = [1, 100]$	$x_2 = [1, 100]$
$x_3 = [2, 101]$	$x_3 = [2, \infty]$
$x_4 = [101, 101]$	$x_4 = [101, \infty]$
<b>100 iterations</b>	<b>2 iterations</b>

- When widening ends with  $s > F(s)$ , we improve solution  $s$  as follows:  $s \geq F(s) \geq \dots \geq F^n(s) \geq \dots \geq s_0$ , where  $s_0$  is the least fixpoint.
- When the sequence is finite, its limit is better approximation of  $s_0$ .
- If the sequence is infinite, we apply **narrowing function**  $\Delta: L \times L \rightarrow L$  at not stabilizing location  $x_i$  such that  $x_i = x_i \Delta F_i(\vec{x})$ .
- Operator  $\Delta$  must satisfy:
  - for each  $x, y \in L$ ,  $x > y \rightarrow (x \geq x \Delta y \geq y)$ , i.e.  $\Delta$  tries to slow down the decreasing of the sequence,
  - it ensures, that every infinite sequence of elements starting from any  $s$  is not strictly decreasing.

# Narrowing

Example: Interval bounds of integer variable  $x$

```
{locations are after}
1  x := 1;
2  while (x <= 100) {
3      x := x + 1;
4  }
```

# Narrowing

Example: Interval bounds of integer variable  $x$

{locations are after}

1  $x := 1;$

2 while ( $x \leq 100$ ) {

3      $x := x + 1;$

4 }

{functions}

$x_1 = [1, 1]$

$x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$

$x_3 = x_2 + [1, 1]$

$x_4 = (x_1 \cup x_3) \cap [101, \infty]$

# Narrowing

Example: Interval bounds of integer variable  $x$

{locations after}	{functions}
1 $x := 1;$	$x_1 = [1, 1]$
2 $\text{while } (x \leq 100) \{$	$x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$
3 $x := x + 1;$	$x_3 = x_2 + [1, 1]$
4 $\}$	$x_4 = (x_1 \cup x_3) \cap [101, \infty]$

Narrowing operator  $\Delta$ :

$$[i, j] \Delta [k, l] = [\text{ite}(i = -\infty, k, \min(i, k)), \text{ite}(j = \infty, l, \max(j, l))]$$

# Narrowing

Example: Interval bounds of integer variable  $x$

{locations are after}	{functions}
1 $x := 1;$	$x_1 = [1, 1]$
2 while ( $x \leq 100$ ) {	$x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$
3 $x := x + 1;$	$x_3 = x_2 + [1, 1]$
4 }	$x_4 = (x_1 \cup x_3) \cap [101, \infty]$

Narrowing operator  $\Delta$ :

$[i, j] \Delta [k, l] = [ite(i = -\infty, k, \min(i, k)), ite(j = \infty, l, \max(j, l))]$

{no widening}	{widening}
$x_1 = [1, 1]$	$x_1 = [1, 1]$
$x_2 = [1, 100]$	$x_2 = [1, 100]$
$x_3 = [2, 101]$	$x_3 = [2, \infty]$
$x_4 = [101, 101]$	$x_4 = [101, \infty]$
100 iterations	2 iteration

# Narrowing

Example: Interval bounds of integer variable  $x$

{locations are after}	{functions}
1 $x := 1;$	$x_1 = [1, 1]$
2 while ( $x \leq 100$ ) {	$x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$
3 $x := x + 1;$	$x_3 = x_2 + [1, 1]$
4 }	$x_4 = (x_1 \cup x_3) \cap [101, \infty]$

Narrowing operator  $\Delta$ :

$[i, j] \Delta [k, l] = [ite(i = -\infty, k, \min(i, k)), ite(j = \infty, l, \max(j, l))]$

{no widening}	{widening}	{ $x_3 = x_3 \Delta (x_2 + [1, 1])$ }
$x_1 = [1, 1]$	$x_1 = [1, 1]$	$x_1 = [1, 1]$
$x_2 = [1, 100]$	$x_2 = [1, 100]$	$x_2 = [1, 100]$
$x_3 = [2, 101]$	$x_3 = [2, \infty]$	$x_3 = [2, 101]$
$x_4 = [101, 101]$	$x_4 = [101, \infty]$	$x_4 = [101, 101]$
100 iterations	2 iteration	+1 iteration

# Narrowing

Example: Interval bounds of integer variable  $x$

<code>{locations are after}</code>	<code>{functions}</code>
<code>1 x := 1;</code>	$x_1 = [1, 1]$
<code>2 while (x &lt;= 100) {</code>	$x_2 = (x_1 \cup x_3) \cap [-\infty, 100]$
<code>3     x := x + 1;</code>	$x_3 = x_2 + [1, 1]$
<code>4 }</code>	$x_4 = (x_1 \cup x_3) \cap [101, \infty]$

Narrowing operator  $\Delta$ :

$[i, j] \Delta [k, l] = [ite(i = -\infty, k, \min(i, k)), ite(j = \infty, l, \max(j, l))]$

<code>{no widening}</code>	<code>{widening}</code>	<code>{<math>x_3 = x_3 \Delta (x_2 + [1, 1])</math>}</code>
$x_1 = [1, 1]$	$x_1 = [1, 1]$	$x_1 = [1, 1]$
$x_2 = [1, 100]$	$x_2 = [1, 100]$	$x_2 = [1, 100]$
$x_3 = [2, 101]$	$x_3 = [2, \infty]$	$x_3 = [2, 101]$
$x_4 = [101, 101]$	$x_4 = [101, \infty]$	$x_4 = [101, 101]$
<b>100 iterations</b>	<b>2 iteration</b>	<b>+1 iteration</b>



## Shape Analysis via 3-Valued Logic

- Static analysis of dynamic memory.
- It can detect NULL dereferences, memory leaks, etc.
- Applicable to real code.