CS 453/698: Software and Systems Security

Module: Bug Finding Tools and Practices
Lecture: Static analysis

Meng Xu *(University of Waterloo)*

Spring 2025

Outline

1. Declarative programming
2. Introduction to abstraction interpretation
3. Reaching fixedpoint: joining, widening, and narrowing

Why this topic?

A significant portion of software security research is based on the following observation:

> *If the program contains some specific code pattern, that program is more likely to be vulnerable.*
>
> - *e.g.,* `malloc` *with* `strlen` *as size*
> - *e.g.,* `strcpy` *taking a user-supplied* `src` *argument*

Why this topic?

A significant portion of software security research is based on the following observation:

> *If the program contains some specific code pattern, that program is more likely to be vulnerable.*
>
> - *e.g.,* `malloc` *with* `strlen` *as size*
> - *e.g.,* `strcpy` *taking a user-supplied* `src` *argument*

Q: How do you precisely define and express this code pattern?

Why this topic?

A significant portion of software security research is based on the following observation:

If the program contains some specific code pattern, that program is more likely to be vulnerable.

- *e.g.,* `malloc` *with* `strlen` *as size*
- *e.g.,* `strcpy` *taking a user-supplied* `src` *argument*

Q: How do you precisely define and express this code pattern?

A: Declarative programming, e.g., Datalog and CodeQL, is an option

Programming paradigm: imperative vs declarative

Programming paradigm: imperative vs declarative

Declarative programming is a paradigm describing WHAT the program knows and does, without explicitly specifying its algorithm.

Imperative programming is a paradigm describing HOW the program should do something by explicitly specifying each instruction (or state transition) step by step.

Baking a chocolate cake

The imperative way

1. mix flour, sugar, cocoa powder, baking soda, and salt
2. add milk, vegetable oil, eggs, and vanilla to form the batter
3. preheat the oven at 180°C
4. put the batter in a cake pan and bake for 30 minutes

The declarative way

- cake = batter + 180°C oven + 30 minutes backing
- batter = solid ingredients + liquid ingredients
- solid ingredients = flour, sugar, cocoa powder, baking soda, and salt
- fluid ingredients = milk, vegetable oil, eggs, and vanilla

Finding a vulnerability

The imperative way

1. for each function in the program, search for a `strcpy` call in the function body
2. trace back how the `src` argument in the `strcpy` call is derived (via def-use analysis)
3. for any ancestor in the trace, if it comes from untrusted user-controlled input, mark the `strcpy` call as vulnerable

The declarative way

- program = [function]
- function = [instruction] (per each function)
- *defines*(var, instruction)
- *uses*(instruction, var)
- *is_user_controlled*(var)
- *is_strcpy_vuln* =
  `strcpy(..., src)`
  \+ *defines*(src, i_src)
  \+ *uses*(i_src, x)
  \+ *defines*(x, i_x)
  \+ *uses*(i_x, var)
  \+ *is_user_controlled*(var)

A new trend: declarative vulnerability finding

Recent years have observed a new trend in applying declarative-alike tooling in finding security vulnerabilities.

A new trend: declarative vulnerability finding

Recent years have observed a new trend in applying declarative-alike tooling in finding security vulnerabilities.

The (arguably) most promininent example is CodeQL, a commercial tool developed by Semmle, which was acquired by GitHub in 2019.

A new trend: declarative vulnerability finding

Recent years have observed a new trend in applying declarative-alike tooling in finding security vulnerabilities.

The (arguably) most promininent example is CodeQL, a commercial tool developed by Semmle, which was acquired by GitHub in 2019.

Other use cases include:

- Gigahorse
- Vandle
- Securify 2.0

CodeQL example

```
import cpp
import semmle.code.cpp.controlflow.SSA

class MallocCall extends FunctionCall
{
    MallocCall() { this.getTarget().hasGlobalName("malloc") }

    Expr getAllocatedSize() {
        if this.getArgument(0) instanceof VariableAccess then
            exists(LocalScopeVariable v, SsaDefinition ssaDef |
                result = ssaDef.getAnUltimateDefiningValue(v)
                and this.getArgument(0) = ssaDef.getAUse(v))
        else
            result = this.getArgument(0)
    }
}

from MallocCall malloc
where malloc.getAllocatedSize() instanceof StrlenCall
select malloc, "This allocation does not include space to null-terminate."
```

Other areas of program analysis

Declarative programming, especially Datalog, has also been widely used in other program analysis areas, including

- DOOP points-to analysis (for Java)
- cclyzer++ points-to analysis (for LLVM)
- DDisasm disassembler

Reasons to use declarative programming for static analysis

Precise definition of bug patterns can be beneficial:

- e.g., compare with another code pattern
- e.g., inter-op / composite with code patterns
- e.g., scale to more codebases
- e.g., argue for soundness / completeness

Outline

Why this topic?

A significant portion of software security research is related to program analysis:

- derive properties which hold for program P (i.e., inference)
- prove that some property holds for program P (i.e., verification)
- given a program P, generate a program P' which is
  - in most ways equivalent to P
  - behaves better than P w.r.t some criteria

  (i.e., transformation)

Why this topic?

A significant portion of software security research is related to program analysis:

- derive properties which hold for program P (i.e., inference)
- prove that some property holds for program P (i.e., verification)
- given a program P, generate a program P' which is
  - in most ways equivalent to P
  - behaves better than P w.r.t some criteria

  (i.e., transformation)

Abstract interpretation provides a formal framework for developing program analysis tools.

What is abstract interpretation?

Consider detecting that one branch will not be taken in:

$\textbf{int}\ x, y, z; \quad y := read(file); \quad x := y * y;$

$\textbf{if}\ x \ \geq\ 0\ \textbf{then}\ z := 1\ \textbf{else}\ z := 0$

What is abstract interpretation?

Consider detecting that one branch will not be taken in:
int $x, y, z; \quad y := read(file); \quad x := y * y;$
if $x \geq 0$ **then** $z := 1$ **else** $z := 0$

- Exhaustive analysis in the standard domain: non-termination
- Human reasoning about programs – uses abstractions: signs, order of magnitude, odd/even, ...

What is abstract interpretation?

Consider detecting that one branch will not be taken in:
int $x, y, z; \quad y := read(file); \quad x := y * y;$
if $x \;\geq\; 0$ **then** $z := 1$ **else** $z := 0$

- Exhaustive analysis in the standard domain: non-termination
- Human reasoning about programs – uses abstractions: signs, order of magnitude, odd/even, ...

Basic idea: use approximate (generally finite) representations of computational objects to make the problem of program dataflow analysis tractable.

What is abstract interpretation?

Abstract interpretation is a formalization of the above procedure:

- define a non-standard semantics which can approximate the meaning (or behaviour) of the program in a finite way
- expressions are computed over an approximate (abstract) domain rather than the concrete domain (i.e., meaning of operators has to be reconsidered w.r.t. this new domain)

Example: integer sign arithmetic

Consider the domain $D = Z$ (integers)
and the multiplication operator: $* : Z^2 \to Z$

We define an "abstract domain:" $D_\alpha = \{[-], [+]\}$
and abstract multiplication: $*_\alpha : D_\alpha^2 \to D_\alpha$ defined by:

$*_\alpha$	$[-]$	$[+]$
$[-]$	$[+]$	$[-]$
$[+]$	$[-]$	$[+]$

Example: integer sign arithmetic

Consider the domain $D = Z$ (integers)
and the multiplication operator: $* : Z^2 \to Z$

We define an "abstract domain:" $D_\alpha = \{[-], [+]\}$
and abstract multiplication: $*_\alpha : D_\alpha^2 \to D_\alpha$ defined by:

$*_\alpha$	$[-]$	$[+]$
$[-]$	$[+]$	$[-]$
$[+]$	$[-]$	$[+]$

This allows us to conclude, for example, that $y = x^2 = x * x$ is never negative.

Some observations

- The basis is that whenever we have $z = x * y$ then:
  if $x, y \in Z$ are approximated by $x_\alpha, y_\alpha \in D_\alpha$
  then $z \in Z$ is approximated by $z_\alpha = x_\alpha *_\alpha y_\alpha$
  - Essentially, we map from an unbounded domain to a finite domain.
- It is important to formalize this notion of approximation, in order to be able to reason/prove that the analysis is correct.
- Approximate computation is generally less precise but faster (hence the tradeoff).

Example: integer sign arithmetic (refined)

Again, $D = Z$ (integers)
and: $* : Z^2 \rightarrow Z$

We can define a more refined "abstract domain"
$D'_\alpha = \{[-], [0], [+]\}$

and the corresponding abstract multiplication: $*_\alpha : {D'}^2_\alpha \rightarrow D'_\alpha$

$*_\alpha$	$[-]$	$[0]$	$[+]$
$[-]$	$[+]$	$[0]$	$[-]$
$[0]$	$[0]$	$[0]$	$[0]$
$[+]$	$[-]$	$[0]$	$[+]$

Example: integer sign arithmetic (refined)

Again, $D = Z$ (integers)
and: $* : Z^2 \to Z$

We can define a more refined "abstract domain"
$D'_\alpha = \{[-], [0], [+]\}$

and the corresponding abstract multiplication: $*_\alpha : {D'}_\alpha^2 \to D'_\alpha$

$*_\alpha$	$[-]$	$[0]$	$[+]$
$[-]$	$[+]$	$[0]$	$[-]$
$[0]$	$[0]$	$[0]$	$[0]$
$[+]$	$[-]$	$[0]$	$[+]$

This allows us to conclude, for example, that $z = y * (0 * x)$ is zero.

More observations

- There is a degree of freedom in defining different abstract operators and domains.
- The minimal requirement is that they be “safe” or “correct”.
- Different “safe” definitions result in different kinds of analysis.

Example: integer sign arithmetic (with addition)

Again, $D = Z$ (integers)
and now we want to define the *addition* operator $+ : Z^2 \to Z$

Example: integer sign arithmetic (with addition)

Again, $D = Z$ (integers)
and now we want to define the *addition* operator $+ : Z^2 \rightarrow Z$

We cannot use $D'_\alpha = \{[-], [0], [+]\}$ because we wouldn't know how to represent the result of $[+] +_\alpha [-]$, (i.e., the abstract addition would not be closed).

Example: integer sign arithmetic (with addition)

Again, $D = Z$ (integers)
and now we want to define the *addition* operator $+ : Z^2 \to Z$

We cannot use $D'_\alpha = \{[-], [0], [+]\}$ because we wouldn't know how to represent the result of $[+] +_\alpha [-]$, (i.e., the abstract addition would not be closed).

Solution: introduce a new element "$\top$" in the abstract domain as an approximation of any integer.

Example: integer sign arithmetic (with addition)

New "abstract domain": $D'_\alpha = \{[-], [0], [+], \top\}$

Abstract $+_\alpha : D'^2_\alpha \to D'_\alpha$

$+_\alpha$	$[-]$	$[0]$	$[+]$	$\top$
$[-]$	$[-]$	$[-]$	$\top$	$\top$
$[0]$	$[-]$	$[0]$	$[+]$	$\top$
$[+]$	$\top$	$[+]$	$[+]$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Abstract $*_\alpha : D'^2_\alpha \to D'_\alpha$

$*_\alpha$	$[-]$	$[0]$	$[+]$	$\top$
$[-]$	$[+]$	$[0]$	$[-]$	$\top$
$[0]$	$[0]$	$[0]$	$[0]$	$[0]$
$[+]$	$[-]$	$[0]$	$[+]$	$\top$
$\top$	$\top$	$[0]$	$\top$	$\top$

Example: integer sign arithmetic (with addition)

New "abstract domain": $D'_\alpha = \{[-], [0], [+], \top\}$

Abstract $+_\alpha : D'^2_\alpha \to D'_\alpha$

$+_\alpha$	$[-]$	$[0]$	$[+]$	$\top$
$[-]$	$[-]$	$[-]$	$\top$	$\top$
$[0]$	$[-]$	$[0]$	$[+]$	$\top$
$[+]$	$\top$	$[+]$	$[+]$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Abstract $*_\alpha : D'^2_\alpha \to D'_\alpha$

$*_\alpha$	$[-]$	$[0]$	$[+]$	$\top$
$[-]$	$[+]$	$[0]$	$[-]$	$\top$
$[0]$	$[0]$	$[0]$	$[0]$	$[0]$
$[+]$	$[-]$	$[0]$	$[+]$	$\top$
$\top$	$\top$	$[0]$	$\top$	$\top$

We can now reason that $z = x^2 + y^2$ is never negative

More observations

- In addition to the imprecision due to the coarseness of D_α, the abstract versions of the operations (dependent on D_α) may introduce further imprecision
- Thus, the choice of *abstract domain* and the definition of the *abstract operators* are crucial.

Concerns in abstract interpretation

- Required:
  - Correctness – safe approximations: the analysis should be "conservative" and errs on the "safe side"
  - Termination – compilation should definitely terminate

  (note: not always the case in everyday program analysis tools!)

- Desirable – "practicality":
  - Efficiency – in practice finite analysis time is not enough: finite *and* small is the requirement.
  - Accuracy – too many false alarms is harmful to the adoption of the analysis tool ("the boy who cried wolf").
  - Usefulness – determines which information is worth collecting.

Outline

1. Declarative programming
2. Introduction to abstraction interpretation
3. Reaching fixedpoint: joining, widening, and narrowing

Abstract domain example: intervals

Consider the following abstract domain for $x \in Z$ (integers):
$x = [a, b]$ where

- a can be either a constant or $-\infty$ and
- b can be either a constant or ∞.

Abstract domain example: intervals

Consider the following abstract domain for $x \in Z$ (integers): $x = [a, b]$ where

- a can be either a constant or $-\infty$ and
- b can be either a constant or ∞.

Example:

$\{x^{\#} = [0, 3], y^{\#} = [0, 2]\}$

```
z = 2 * x + 4 * y
```

$\{z^{\#} = 2 \times^{\#} [0, 3] +^{\#} 4 \times^{\#} [0, 2] = [0, 14]\}$

Abstract domain example: intervals

Consider the following abstract domain for $x \in Z$ (integers):
$x = [a, b]$ where

- a can be either a constant or $-\infty$ and
- b can be either a constant or ∞.

Example:

$\{x^{\#} = [0, 3],\ y^{\#} = [0, 2]\}$

```
z = 2 * x + 4 * y
```

$\{z^{\#} = 2 \times^{\#} [0, 3] +^{\#} 4 \times^{\#} [0, 2] = [0, 14]\}$

Q: Why $z^{\#}$ is an abstraction of z?

Join operator

The join operator $\sqcup$ merges two or more abstract states into one abstract state.

Joining operator example

$\{x^\# = [0, 10]\}$

```
if (x < 0) then
  s := -1
else if (x > 0) then
  s := 1
else
  s := 0
```

Joining operator example

$\{x^\# = [0, 10]\}$

```
if (x < 0) then
```
$\{x^\# = \emptyset\}$
```
  s := -1
```
$\{x^\# = \emptyset,\ s^\# = \emptyset\}$
```
else if (x > 0) then

  s := 1

else

  s := 0
```

Joining operator example

$\{x^\# = [0, 10]\}$

```
if (x < 0) then
```

$\{x^\# = \emptyset\}$

```
  s := -1
```

$\{x^\# = \emptyset,\ s^\# = \emptyset\}$

```
else if (x > 0) then
```

$\{x^\# = [1, 10]\}$

```
  s := 1
```

$\{x^\# = [1, 10],\ s^\# = [1, 1]\}$

```
else


  s := 0
```

Joining operator example

$\{x^{\#} = [0, 10]\}$

```
if (x < 0) then
```

$\{x^{\#} = \emptyset\}$

```
   s := -1
```

$\{x^{\#} = \emptyset,\ s^{\#} = \emptyset\}$

```
else if (x > 0) then
```

$\{x^{\#} = [1, 10]\}$

```
   s := 1
```

$\{x^{\#} = [1, 10],\ s^{\#} = [1, 1]\}$

```
else
```

$\{x^{\#} = [0, 0]\}$

```
   s := 0
```

$\{x^{\#} = [0, 0],\ s^{\#} = [0, 0]\}$

Joining operator example

$\{x^\# = [0, 10]\}$

```
if (x < 0) then
```

$\{x^\# = \emptyset\}$

```
  s := -1
```

$\{x^\# = \emptyset,\ s^\# = \emptyset\}$

```
else if (x > 0) then
```

$\{x^\# = [1, 10]\}$

```
  s := 1
```

$\{x^\# = [1, 10],\ s^\# = [1, 1]\}$

```
else
```

$\{x^\# = [0, 0]\}$

```
  s := 0
```

$\{x^\# = [0, 0],\ s^\# = [0, 0]\}$

$\{x^\# = \emptyset \sqcup [1, 10] \sqcup [0, 0] = [0, 10],\ s^\# = \emptyset \sqcup [1, 1] \sqcup [0, 0] = [0, 1]\}$

What about loops?

$\{x^{\#} = \emptyset\}$

```
x := 0
while (x < 100) {
  x := x + 2
}
```

What about loops?

$\{x^\# = \emptyset\}$

```
x := 0
```

$\{x^\# = \langle even \rangle\}$

```
while (x < 100) {

  x := x + 2

}
```

What about loops?

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = \langle even \rangle\}$
```
while (x < 100) {
```
$\{x^{\#} = \langle even \rangle\}_1$
```
  x := x + 2
```
$\{x^{\#} = \langle even \rangle\}_1$
```
}
```

What about loops?

$\{x^{\#} = \emptyset\}$

```
x := 0
```

$\{x^{\#} = \langle even \rangle\}$

```
while (x < 100) {
```

$\{x^{\#} = \langle even \rangle\}_1$ $\{x^{\#} = \langle even \rangle \sqcup \langle even \rangle = \langle even \rangle\}_2$

```
  x := x + 2
```

$\{x^{\#} = \langle even \rangle\}_1$

```
}
```

What about loops?

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = \langle even \rangle\}$
```
while (x < 100) {
```
$\{x^{\#} = \langle even \rangle\}_1$ $\{x^{\#} = \langle even \rangle \sqcup \langle even \rangle = \langle even \rangle\}_2$
```
  x := x + 2
```
$\{x^{\#} = \langle even \rangle\}_1$
```
}
```
$\{x^{\#} = \langle even \rangle\}$

Two iterations to reach fixedpoint (i.e., none of the abstract states changes).

Collecting semantics

$\{x^{\#} = \emptyset\}$

```
x := 0

while (x < 100) {

   x := x + 2

}
```

Collecting semantics

$\{x^{\#} = \emptyset\}$

```
x := 0
```

$\{x^{\#} = [0, 0]\}$

```
while (x < 100) {

   x := x + 2

}
```

Collecting semantics

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = [0, 0]\}$
```
while (x < 100) {
```
$\{x^{\#} = [0, 0]\}_1$
```
    x := x + 2
```
$\{x^{\#} = [2, 2]\}_1$
```
}
```

Collecting semantics

$\{x^\# = \emptyset\}$

x := 0

$\{x^\# = [0, 0]\}$

while (x < 100) {

$\{x^\# = [0, 0]\}_1$ $\{x^\# = [0, 0] \sqcup [2, 2] = [0, 2]\}_2$

x := x + 2

$\{x^\# = [2, 2]\}_1$ $\{x^\# = [2, 2] \sqcup [2, 4] = [2, 4]\}_2$

}

Collecting semantics

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = [0, 0]\}$
```
while (x < 100) {
```
$\{x^{\#} = [0, 0]\}_1$ $\{x^{\#} = [0, 2] \sqcup [2, 4] = [0, 4]\}_3$
```
  x := x + 2
```
$\{x^{\#} = [2, 2]\}_1$ $\{x^{\#} = [2, 4] \sqcup [2, 6] = [2, 6]\}_3$
```
}
```

Collecting semantics

$\{x^\# = \emptyset\}$

```
x := 0
```

$\{x^\# = [0,0]\}$

```
while (x < 100) {
```

$\{x^\# = [0,0]\}_1 \qquad \{\cdots\}_4, \{\cdots\}_5, \cdots$

```
  x := x + 2
```

$\{x^\# = [2,2]\}_1 \qquad \{\cdots\}_4, \{\cdots\}_5, \cdots$

```
}
```

Collecting semantics

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = [0, 0]\}$
```
while (x < 100) {
```
$\{x^{\#} = [0, 0]\}_1$ $\{x^{\#} = [0, 96] \sqcup [2, 98] = [0, 98]\}_{50}$
```
   x := x + 2
```
$\{x^{\#} = [2, 2]\}_1$ $\{x^{\#} = [2, 98] \sqcup [2, 100] = [2, 100]\}_{50}$
```
}
```

Collecting semantics

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = [0, 0]\}$
```
while (x < 100) {
```
$\{x^{\#} = [0, 0]\}_1$ $\{x^{\#} = [0, 96] \sqcup [2, 98] = [0, 98]\}_{50}$
```
  x := x + 2
```
$\{x^{\#} = [2, 2]\}_1$ $\{x^{\#} = [2, 98] \sqcup [2, 100] = [2, 100]\}_{50}$
```
}
```
$\{x^{\#} = [100, 100]\}$

50 iterations to reach fixedpoint (i.e., none of the abstract states changes).

Collecting semantics

$\{x^\# = \emptyset\}$

```
x := 0
```

$\{x^\# = [0, 0]\}$

```
while (x < 100) {
```

$\{x^\# = [0, 0]\}_1$ $\{x^\# = [0, 96] \sqcup [2, 98] = [0, 98]\}_{50}$

```
  x := x + 2
```

$\{x^\# = [2, 2]\}_1$ $\{x^\# = [2, 98] \sqcup [2, 100] = [2, 100]\}_{50}$

```
}
```

$\{x^\# = [100, 100]\}$

50 iterations to reach fixedpoint (i.e., none of the abstract states changes).

Q: can we reach the fixedpoint faster?

Widening operator

We compute the limit of the following sequence:

$$X_0 = \bot$$
$$X_{i+1} = X_i \nabla F^{\#}(X_i)$$

where ∇ denotes the widening operator.

Widening operator example

$\{x^{\#} = \emptyset\}$

```
x := 0

while (x < 100) {

  x := x + 2

}
```

Widening operator example

$\{x^{\#} = \emptyset\}$

```
x := 0
```

$\{x^{\#} = [0, 0]\}$

```
while (x < 100) {

  x := x + 2

}
```

Widening operator example

$\{x^\# = \emptyset\}$

```
x := 0
```
$\{x^\# = [0, 0]\}$
```
while (x < 100) {
```
  $\{x^\# = [0, 0]\}_1$
```
  x := x + 2
```
  $\{x^\# = [2, 2]\}_1$
```
}
```

Widening operator example

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = [0, 0]\}$
```
while (x < 100) {
```
$\{x^{\#} = [0, 0]\}_1$ $\{x^{\#} = [0, 0] \triangledown [2, 2] = [0, +\infty]\}_2$
```
   x := x + 2
```
$\{x^{\#} = [2, 2]\}_1$ $\{x^{\#} = [2, +\infty]\}_2$
```
}
```

Widening operator example

$\{x^\# = \emptyset\}$

```
x := 0
```

$\{x^\# = [0,0]\}$

```
while (x < 100) {
```

$\{x^\# = [0,0]\}_1$ $\{x^\# = [0,+\infty]\nabla[2,+\infty] = [0,+\infty]\}_3$

```
    x := x + 2
```

$\{x^\# = [2,2]\}_1$ $\{x^\# = [2,+\infty]\}_3$

```
}
```

Widening operator example

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = [0, 0]\}$
```
while (x < 100) {
```
$\{x^{\#} = [0, 0]\}_1$ $\{x^{\#} = [0, +\infty] \triangledown [2, +\infty] = [0, +\infty]\}_3$
```
  x := x + 2
```
$\{x^{\#} = [2, 2]\}_1$ $\{x^{\#} = [2, +\infty]\}_3$
```
}
```
$\{x^{\#} = [100, +\infty]\}$

3 iterations to reach fixedpoint (i.e., none of the abstract states changes).

Narrowing operator

We compute the limit of the following sequence:

$$X_0 = \bot$$
$$X_{i+1} = X_i \triangle F^{\#}(X_i)$$

where $\triangle$ denotes the narrowing operator.

Narrowing operator example

$\{x^\# = \emptyset\}$

```
x := 0
```
$\{x^\# = [0, 0]\}$
```
while (x < 100) {
```
$\{x^\# = [0, +\infty]\}$
```
  x := x + 2
```
$\{x^\# = [2, +\infty]\}$
```
}
```
$\{x^\# = [100, 101]\}$

Narrowing operator example

$\{x^{\#} = \emptyset\}$

```
x := 0
```

$\{x^{\#} = [0, 0]\}$

```
while (x < 100) {
```

$\{x^{\#} = [0, +\infty]\}$ $\{x^{\#} = [0, +\infty] \triangle [0, 99] = [0, 99]\}_1$

```
  x := x + 2
```

$\{x^{\#} = [2, +\infty]\}$ $\{x^{\#} = [2, 101]\}_1$

```
}
```

$\{x^{\#} = [100, 101]\}$

Narrowing operator example

$\{x^\# = \emptyset\}$

```
x := 0
```
$\{x^\# = [0, 0]\}$
```
while (x < 100) {
```
$\{x^\# = [0, +\infty]\}$ $\{x^\# = [2, 101] \triangle [0, 99] = [0, 99]\}_2$
```
  x := x + 2
```
$\{x^\# = [2, +\infty]\}$ $\{x^\# = [2, 101]\}_2$
```
}
```
$\{x^\# = [100, 101]\}$

Narrowing operator example

$\{x^\# = \emptyset\}$

```
x := 0
```

$\{x^\# = [0, 0]\}$

```
while (x < 100) {
```

$\{x^\# = [0, +\infty]\}$ $\{x^\# = [2, 101] \triangle [0, 99] = [0, 99]\}_2$

```
x := x + 2
```

$\{x^\# = [2, +\infty]\}$ $\{x^\# = [2, 101]\}_2$

```
}
```

$\{x^\# = [100, 101]\}$

2 iterations to reach fixedpoint (i.e., none of the abstract states changes).

$\langle$ **End** $\rangle$