Lesson 4

Abstract Interpretation

Static Analysis of Programs and Constraint Solving (2019-20)

Master's Degree in Formal Methods in Computer Science

Manuel Montenegro (montenegro@fdi.ucm.es)

Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Despacho 219
Universidad Complutense de Madrid

Outline

Outline

Introduction

- Abstract Interpretation is a formal framework suitable to the design of static analyses that approximate the executions of a program.
- It can be seen as a symbolic execution in which variables hold program properties, instead of concrete values.

- This framework was developed by Patrick Cousot and Radhia Cousot in the 70s, and it is still one of the most relevant approaches to static analysis.

  P. Cousot, R. Cousot
  Static determination of dynamic properties of programs.
  2nd International Symposium on Programming, pp. 106–130, 1976.

- Assume, for example that we want to know whether the variables of a program can take positive values, negative values, or zero.

Concrete execution

```
[]
x := 5;
[x ↦ 5]
y := 2;
[x ↦ 5,y ↦ 2]
x := x + y;
[x ↦ 7,y ↦ 2]
y := y - x
[x ↦ 7,y ↦ -5]
```

Abstract execution

```
[]
x := 5;
[x ↦ +]
y := 2;
[x ↦ +,y ↦ +]
x := x + y;
[x ↦ +,y ↦ +]
y := y - x
[x ↦ +,y ↦ ??]
```

- Abstract interpretation is formal approach to analysis.
- In order to know how to build an abstract execution we have to:

1. Define the concrete semantics of a language.
   - 😊 We already know how to do that! (TLP)
   - 😕 Kind of...
2. Set up a correspondence between concrete and abstract values.
   - 😎 This is given by a Galois connection.

- However, there are some issues with the abstract execution.
  - Conditionals: Which branch does abstract execution take?
  - Loops: How many iterations does the abstract interpretation do?
- To address these programs we need some tools for computing fixed points in monotonic functions.

  😊 Yes... again.
- Moreover, we will introduce two new tools: widening and narrowing.

- Variables $x \in \textbf{Var}$.
- Arithmetic expressions $e \in \textbf{AExp}$:

$$e ::= n \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2$$

- Boolean expressions $b \in \textbf{BExp}$:

$$b ::= \mathit{true} \mid \mathit{false} \mid e_1 = e_2 \mid e_1 \leq e_2 \mid \neg b \mid b_1 \wedge b_2$$

- Programas $S \in \textbf{Stm}$:

$$S ::= \texttt{skip} \mid x := e \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S$$

Denotational semantics of arithmetic expressions

- We denote by **State** the set of functions **Var** $\to \mathbb{Z}$.
- The semantics of an arithmetic expression is given by a function $\mathcal{A}$: **AExp** $\to$ **State** $\to \mathbb{Z}$.

$$
\begin{aligned}
\mathcal{A}\,[\![n]\!] &= \lambda\sigma.n \\
\mathcal{A}\,[\![x]\!] &= \lambda\sigma.\sigma(x) \\
\mathcal{A}\,[\![e_1 + e_2]\!] &= \lambda\sigma.\,(\mathcal{A}\,[\![e_1]\!]\ \sigma) + (\mathcal{A}\,[\![e_2]\!]\ \sigma) \\
\mathcal{A}\,[\![e_1 * e_2]\!] &= \lambda\sigma.\,(\mathcal{A}\,[\![e_1]\!]\ \sigma) * (\mathcal{A}\,[\![e_2]\!]\ \sigma) \\
\mathcal{A}\,[\![e_1 - e_2]\!] &= \lambda\sigma.\,(\mathcal{A}\,[\![e_1]\!]\ \sigma) - (\mathcal{A}\,[\![e_2]\!]\ \sigma)
\end{aligned}
$$

- Similarly we define $\mathcal{B} : \mathbf{BExp} \to \mathbf{State} \to \{\mathit{true}, \mathit{false}\}$

$$
\begin{aligned}
\mathcal{B}\,[\![\mathit{true}]\!] &= \lambda\sigma.\mathit{true} \\
\mathcal{B}\,[\![\mathit{false}]\!] &= \lambda\sigma.\mathit{false} \\
\mathcal{B}\,[\![e_1 = e_2]\!] &= \lambda\sigma.\begin{cases} \mathit{true} & \text{if } \mathcal{A}\,[\![e_1]\!]\,\sigma = \mathcal{A}\,[\![e_2]\!]\,\sigma \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathcal{B}\,[\![e_1 \leq e_2]\!] &= \lambda\sigma.\begin{cases} \mathit{true} & \text{if } \mathcal{A}\,[\![e_1]\!]\,\sigma \leq \mathcal{A}\,[\![e_2]\!]\,\sigma \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathcal{B}\,[\![\neg b]\!] &= \lambda\sigma.\begin{cases} \mathit{true} & \text{if } \mathcal{B}\,[\![b]\!]\,\sigma = \mathit{false} \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathcal{B}\,[\![b_1 \wedge b_2]\!] &= \lambda\sigma.\begin{cases} \mathit{true} & \text{if } \mathcal{B}\,[\![b_1]\!]\,\sigma = \mathit{true} \text{ and } \mathcal{B}\,[\![b_2]\!]\,\sigma = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

- In this case, $\mathcal{S} : \textbf{Stm} \to \textbf{State} \to \textbf{State}_\perp$

$$
\begin{array}{lcl}
\mathcal{S}[\![\texttt{skip}]\!] & = & id \\
\mathcal{S}[\![x := e]\!] & = & \lambda\sigma.\sigma\,[x \mapsto (\mathcal{A}[\![e]\!]\,\sigma)] \\
\mathcal{S}[\![S_1; S_2]\!] & = & \mathcal{S}[\![S_2]\!] \circ \mathcal{S}[\![S_1]\!] \\
\mathcal{S}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & cond(\mathcal{B}[\![b]\!], \mathcal{S}[\![S_1]\!], \mathcal{S}[\![S_2]\!]) \\
\mathcal{S}[\![\texttt{while } b \texttt{ do } S]\!] & = & lfp\ \ (\lambda f.cond(\mathcal{B}[\![b]\!], f \circ \mathcal{S}[\![S]\!], id))
\end{array}
$$

- The least fixed point of $\lambda f.cond(\mathcal{B}[\![b]\!], f \circ \mathcal{S}[\![S]\!], id)$ is, in general, not computable.

Semantics of statements

- In the following we shall leave out the $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{S}$ when there is no ambiguity.

$$
\begin{array}{lcl}
[\![\mathtt{skip}]\!] & = & id \\
[\![x := e]\!] & = & \lambda\sigma.\sigma\,[x \mapsto ([\![e]\!]\ \sigma)] \\
[\![S_1; S_2]\!] & = & [\![S_2]\!] \circ [\![S_1]\!] \\
[\![\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2]\!] & = & cond([\![b]\!], [\![S_1]\!], [\![S_2]\!]) \\
[\![\mathtt{while}\ b\ \mathtt{do}\ S]\!] & = & lfp\ (\lambda f.cond([\![b]\!], f \circ [\![S]\!], id))
\end{array}
$$

- The semantic definitions shown so far specify how programs manage concrete integer values.

Example

According to $\mathcal{S}[\![\_]\!]$, the statement $x := 0 - x$ transforms the concrete state $[x \mapsto 8]$ into another concrete state: $[x \mapsto -8]$.

- But we are interested in **properties** of the values, not the values themselves.

- Considering properties as values leads us to a world of abstract values and abstract states.

Example

A sign analysis would say that $x := 0 - x$ transforms the abstract state $[x \mapsto +]$ into $[x \mapsto -]$ and vice versa.

An **abstract interpretation** is a semantic definition that deals with **abstract states** instead of concrete states.

- For example, let S be the following program:

```
n := 1;
while m > 0 do {
  n := n * m;
  m := m - 1
}
```

- Concrete semantics (values in $\mathbb{Z}$):

$$[\![S]\!]\ [m \mapsto 5] = [n \mapsto 120, m \mapsto 0]$$

- Abstract interpretation (values in $\{+, -, 0\}$):

$$[\![S]\!]^{\sharp}\ [m \mapsto +] = [n \mapsto +, m \mapsto 0]$$

Abstract interpretation by example: sign analysis

2. Abstract interpretation by example: sign analysis

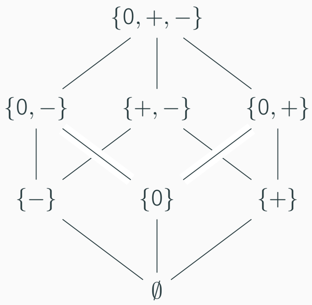
- Arithmetic expressions with a single variable
- Arithmetic expressions with several variables
- Boolean expressions
- Statements

- A sign analysis determines whether a variable takes a positive, negative or zero value in a given execution point.
- Hence we consider the set **Sign** $= \{+, -, 0\}$.
- And abstract state would map variables to subsets of **Sign**.

$\{0\}$	The value of a variable is 0
$\{+\}$	The value of a variable is > 0
$\{-\}$	The value of a variable is < 0
$\{0, +\}$	The value of a variable is ≥ 0
$\{0, -\}$	The value of a variable is ≤ 0
$\{+, -\}$	The value of a variable is > 0 or < 0 (that is, $\neq 0$)
$\{0, +, -\}$	The value of a variable can be any number in $\mathbb{Z}$

Some abstract values seem to be more accurate than others. Can you define an order relation among these values?

- The theoretical framework requires abstract domains to be lattices.
- We add a bottommost element to our ordered set, so it becomes a lattice. This element is the empty set.
- Our abstract domain is, therefore, the lattice $(\mathcal{P}(\mathsf{Sign}), \subseteq)$.



- We shall design our analysis in several steps of increasing complexity.
  1. Arithmetic expressions depending on a fixed variable.
  2. Arithmetic expressions depending on several variables.
  3. Boolean expressions.
  4. *While* programs.

- Let us consider the set **AExp** of arithmetic expressions generated by the following grammar:

$$e ::= n \mid \mathtt{x} \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2$$

where $n \in \mathbb{Z}$ and $\mathtt{x}$ is a fixed variable.

- The value which an expression e is evaluated to depends on the value of x.
- Therefore, the concrete semantics of an expression e, denoted by $[\![e]\!]$ is a function $\mathbb{Z} \to \mathbb{Z}$.

$$
\begin{aligned}
[\![n]\!] &= \lambda x.n \\
[\![x]\!] &= \lambda x.x \\
[\![e_1 + e_2]\!] &= \lambda x.\,([\![e_1]\!]\; x) + ([\![e_2]\!]\; x) \\
[\![e_1 * e_2]\!] &= \lambda x.\,([\![e_1]\!]\; x) * ([\![e_2]\!]\; x) \\
[\![e_1 - e_2]\!] &= \lambda x.\,([\![e_1]\!]\; x) - ([\![e_2]\!]\; x)
\end{aligned}
$$

Example

$$
\begin{aligned}
[\![(2 * \mathsf{x}) + 6 + \mathsf{x}]\!]\ 3 &= [\![2 * \mathsf{x}]\!]\ 3 + [\![6]\!]\ 3 + [\![\mathsf{x}]\!]\ 3 \\
&= ([\![2]\!]\ 3 * [\![\mathsf{x}]\!]\ 3) + [\![6]\!]\ 3 + + [\![\mathsf{x}]\!]\ 3 \\
&= ((\lambda x.\, 2)\ \ 3 * (\lambda x.x)\ \ 3) + (\lambda x.\, 6)\ \ 3 + (\lambda x.x)\ \ 3 \\
&= (2 * 3) + 6 + 3 \\
&= 15
\end{aligned}
$$

In general, $[\![(2 * \mathsf{x}) + 6 + \mathsf{x}]\!]\ x = 3x + 6$, that is,

$$[\![(2 * \mathsf{x}) + 6 + \mathsf{x}]\!] = \lambda x\,.\, 3x + 6$$

- Collecting semantics is the most precise semantics that can be used to describe a given class of properties.
- In our case we want to know whether an arithmetic expression evaluates to a value of:
    - The set of strictly positive integers: $\mathbb{Z}^+$.
    - The set of strictly negative integers: $\mathbb{Z}^-$.
    - The set $\{0\}$.
- Therefore, our collecting semantics has to deal with sets of integers.
    - To which values may an expression be evaluated?

Why is collecting semantics necessary?

- We have to deal with those cases in which x has an unknown value.
- For example, assume the expression $2 * \mathsf{x} + 3$.
  - If we know that x takes the value 2, the collecting semantics of e would be the singleton set $\{7\}$.
  - However, if the only thing we know about x is that it may take either the value 2 or the value 5, the collecting semantics of e would yield the set $\{7, 13\}$.
  - Moreover, if we do not have a hint on the value of x (that is, its value is a member of $\mathbb{Z}$, the most accurate claim about e is that it evaluates to an element of $\mathbb{Z}$.

What would the collecting semantics of $2 * x + x$ be if we knew that x belongs to the set $\{3, 4\}$?

- We use $[\![e]\!]^*$ to denote the collecting semantics of e.
- For any e, $[\![e]\!]^*$ is a function $\mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$.
  - The collecting semantics of e is a function that receives the set of possible values for x and returns the set of possible values that the expression may be evaluated to.
- It can be defined as follows:

$$[\![e]\!]^* = \lambda X.\{[\![e]\!]\ z \mid z \in X\}$$

Examples

$$
\begin{array}{llll}
[\![(2 * x) + x]\!]^* & \{-2, 0, 5\} & = & \{-6, 0, 15\} \\
[\![(2 * x) + x]\!]^* & \{0\} & = & \{0\} \\
[\![(2 * x) + x]\!]^* & \{1, 2, 3, \ldots\} & = & \{3, 6, 9, \ldots\} \\
[\![(2 * x) + x]\!]^* & \{-1, -2, -3, \ldots\} & = & \{-3, -6, -9, \ldots\} \\
[\![(2 * x) + x]\!]^* & \mathbb{Z} & = & \{3x \mid x \in \mathbb{Z}\}
\end{array}
$$

Is this definition compositional?

$$[\![e]\!]^* = \lambda X.\{[\![e]\!]\ z \mid z \in X\}$$

The following alternative definition of $[\![e]\!]^*$ is compositional. It is equivalent to the previous one?

$$
\begin{aligned}
[\![n]\!]^* &= \lambda X.\{n\} \\
[\![\mathsf{x}]\!]^* &= \lambda X.X \\
[\![e_1 + e_2]\!]^* &= \lambda X.\{x + y \mid x \in [\![e_1]\!]^*\, X, y \in [\![e_2]\!]^*\, X\} \\
[\![e_1 - e_2]\!]^* &= \lambda X.\{x - y \mid x \in [\![e_1]\!]^*\, X, y \in [\![e_2]\!]^*\, X\} \\
[\![e_1 * e_2]\!]^* &= \lambda X.\{x * y \mid x \in [\![e_1]\!]^*\, X, y \in [\![e_2]\!]^*\, X\}
\end{aligned}
$$

Still… could we define $[\![e]\!]^*$ compositionally in order to obtain a definition equivalent to the first one?

Which definition should we use?

- The first and the third definitions are more precise, but awkward to work with.
- The second definition is compositional, but involves loss of accuracy.
    - Anyway, sign analysis is going to incur the same loss of precision.
- For the technical development that follows, any of these definitions are OK, but as we progress through the lesson, we would prefer definitions like the second one.

Is $[\![e]\!]^* \, X$ computable for any e and X?

- An abstract interpretation is a (usually computable) semantic definition that deals with abstract elements in $\mathcal{P}(\mathsf{Sign})$ instead of concrete elements $\mathcal{P}(\mathbb{Z})$.
- We denote the abstract interpretation of e by $[\![e]\!]^\sharp$.
- For any e, $[\![e]\!]^\sharp$ is a function $\mathcal{P}(\mathsf{Sign}) \to \mathcal{P}(\mathsf{Sign})$.

- For any e, $[\![e]\!]^\sharp$ is defined as follows:

$$
\begin{aligned}
[\![n]\!]^\sharp &= \lambda S.\ \begin{cases} \{+\} & \text{if } n > 0 \\ \{-\} & \text{if } n < 0 \\ \{0\} & \text{if } n = 0 \end{cases} \\
[\![\mathsf{x}]\!]^\sharp &= \lambda S.\ S \\
[\![e_1 + e_2]\!]^\sharp &= \lambda S.\ ([\![e_1]\!]^\sharp\ S) \oplus ([\![e_2]\!]^\sharp\ S) \\
[\![e_1 * e_2]\!]^\sharp &= \lambda S.\ ([\![e_1]\!]^\sharp\ S) \otimes ([\![e_2]\!]^\sharp\ S) \\
[\![e_1 - e_2]\!]^\sharp &= \lambda S.\ ([\![e_1]\!]^\sharp\ S) \ominus ([\![e_2]\!]^\sharp\ S)
\end{aligned}
$$

where $\oplus$, $\otimes$, and $\ominus$ are functions $(\mathcal{P}(\mathbf{Sign}) \times \mathcal{P}(\mathbf{Sign})) \to \mathcal{P}(\mathbf{Sign})$ that will be defined in the following tables.

Definition of $\oplus$

$\oplus$	$\emptyset$	$\{0\}$	$\{-\}$	$\{+\}$	$\{0,-\}$	$\{0,+\}$	$\{-,+\}$	$\{0,-,+\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{0\}$	$\emptyset$	$\{0\}$	$\{-\}$	$\{+\}$	$\{0,-\}$	$\{0,+\}$	$\{-,+\}$	$\{0,-,+\}$
$\{-\}$	$\emptyset$	$\{-\}$	$\{-\}$	$\{0,-,+\}$	$\{-\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$
$\{+\}$	$\emptyset$	$\{+\}$	$\{0,-,+\}$	$\{+\}$	$\{0,-,+\}$	$\{+\}$	$\{0,-,+\}$	$\{0,-,+\}$
$\{0,-\}$	$\emptyset$	$\{0,-\}$	$\{-\}$	$\{0,-,+\}$	$\{0,-\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$
$\{0,+\}$	$\emptyset$	$\{0,+\}$	$\{0,-,+\}$	$\{0,+\}$	$\{0,-,+\}$	$\{0,+\}$	$\{0,-,+\}$	$\{0,-,+\}$
$\{-,+\}$	$\emptyset$	$\{-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{-,+\}$	$\{0,-,+\}$
$\{0,-,+\}$	$\emptyset$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$

Definition of $\otimes$

$\otimes$	$\emptyset$	$\{0\}$	$\{-\}$	$\{+\}$	$\{0,-\}$	$\{0,+\}$	$\{-,+\}$	$\{0,-,+\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{0\}$	$\emptyset$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$
$\{-\}$	$\emptyset$	$\{0\}$	$\{+\}$	$\{-\}$	$\{0,+\}$	$\{0,-\}$	$\{-,+\}$	$\{0,-,+\}$
$\{+\}$	$\emptyset$	$\{0\}$	$\{-\}$	$\{+\}$	$\{0,-\}$	$\{0,+\}$	$\{-,+\}$	$\{0,-,+\}$
$\{0,-\}$	$\emptyset$	$\{0\}$	$\{0,+\}$	$\{0,-\}$	$\{0,+\}$	$\{0,-\}$	$\{0,-,+\}$	$\{0,-,+\}$
$\{0,+\}$	$\emptyset$	$\{0\}$	$\{0,-\}$	$\{0,+\}$	$\{0,-\}$	$\{0,+\}$	$\{0,-,+\}$	$\{0,-,+\}$
$\{-,+\}$	$\emptyset$	$\{0\}$	$\{-,+\}$	$\{-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{-,+\}$	$\{0,-,+\}$
$\{0,-,+\}$	$\emptyset$	$\{0\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$	$\{0,-,+\}$

Definition of $\ominus$

$X \ominus Y = X \oplus \overline{Y}$, where $\overline{Y}$ is defined as follows:‘

Y	$\overline{Y}$
$\emptyset$	$\emptyset$
$\{0\}$	$\{0\}$
$\{-\}$	$\{+\}$
$\{+\}$	$\{-\}$
$\{0, -\}$	$\{0, +\}$
$\{0, +\}$	$\{0, -\}$
$\{-, +\}$	$\{-, +\}$
$\{0, -, +\}$	$\{0, -, +\}$

Is $[\![e]\!]^{\sharp}\ S$ computable for every expression e and each $S \in \mathcal{P}(\mathsf{Sign})$?

Example

$$\begin{aligned}[\![(3 * x) - 3]\!]^\sharp \{-\} &= \left(\left([\![3]\!]^\sharp \{-\}\right) \otimes \left([\![x]\!]^\sharp \{-\}\right)\right) \ominus \left([\![3]\!]^\sharp \{-\}\right) \\ &= (\{+\} \otimes \{-\}) \ominus \{+\} \\ &= \{-\} \ominus \{+\} \\ &= \{-\}\end{aligned}$$

That is, $(3 * x - 3)$ always evaluates to a negative number provided x contains a negative value.

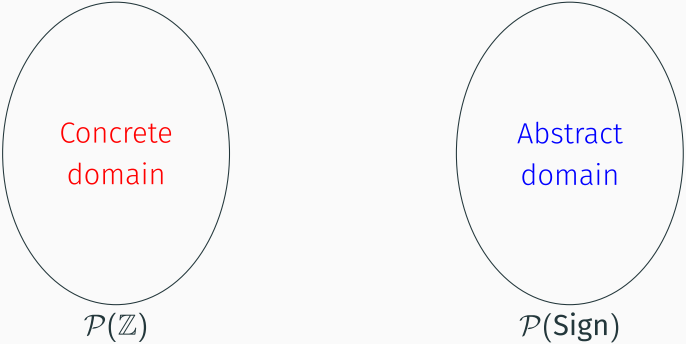
Example

However, for any $S \in \mathcal{P}(\mathbf{Sign})$:

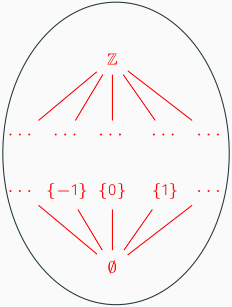
$$\begin{aligned}
[\![3 + (-4)]\!]^\sharp\, S &= \left([\![3]\!]^\sharp\, S\right) \oplus \left([\![-4]\!]^\sharp\, S\right) \\
&= \{+\} \oplus \{-\} \\
&= \{0, -, +\}
\end{aligned}$$

Abstract interpretation involves loss of precision.

Assume that $[\![e_1]\!]^* = [\![e_2]\!]^*$. Does it hold that $[\![e_1]\!]^\sharp = [\![e_2]\!]^\sharp$?

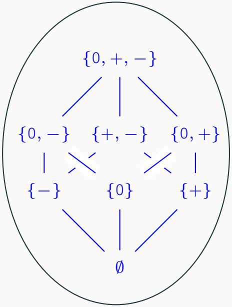


- The elements of $\mathcal{P}(\mathbb{Z})$ that take part in the collecting semantics define a **concrete domain**.
- The elements of $\mathcal{P}(\mathsf{Sign})$ that take part in the abstract interpretation define an **abstract domain**.



$(\mathcal{P}(\mathbb{Z}), \subseteq)$

Are these domains related in some way?



$(\mathcal{P}(\mathsf{Sign}), \subseteq)$

- Considering the inclusion relation ($\subseteq$), they are *posets*.
- In fact, they are **lattices**.
    - Least element $\Rightarrow$ most accurate description.
    - Greatest element $\Rightarrow$ less accurate description.

Assume an expression such that its collecting semantics yields $\{-1, 3, 4\}$. Which of the following abstract values soundly approximates this concrete set?

- $\{+\}$
- $\{+, -\}$
- $\{0, +, -\}$

What about an expression such that its collecting semantics yields $\{-3, -5\}$?

- $\{-\}$
- $\{+, -\}$
- $\emptyset$

Abstraction function α

- For every member of the concrete domain $\mathcal{P}(\mathbb{Z})$ there are one or several elements in the abstract domain that correctly approximate this member.
  - For example, $\{-3, -5\}$ is approximated by:

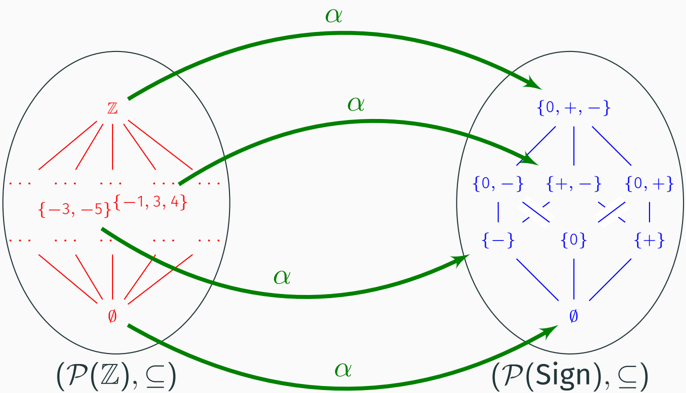
$$\{-\} \qquad \{+, -\} \qquad \{0, -\} \qquad \{0, +, -\}$$

- The greatest lower bound of all those values is the **best approximation** to this element.
  - The best approximation of $\{-3, -5\}$ is:

$$\{-\} \cap \{+, -\} \cap \{0, -\} \cap \{0, +, -\} = \{-\}$$

- The **abstraction function** α is a function $\mathcal{P}(\mathbb{Z}) \rightarrow \mathcal{P}(\mathsf{Sign})$ that maps every element of the concrete domain to the element of the abstract domain that approximates it in the most accurate way.

ABSTRACTION FUNCTION α



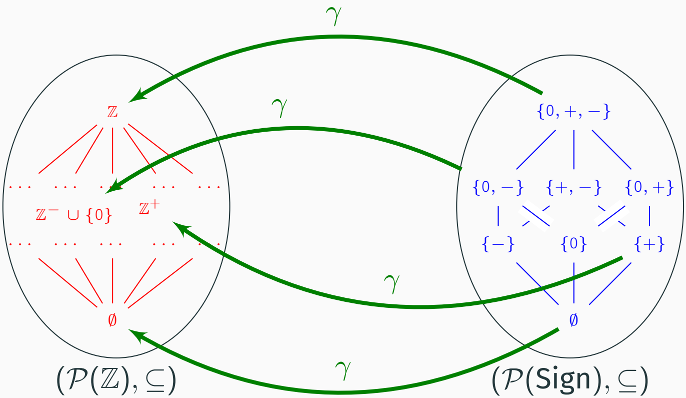
$$\alpha(\{-3,-5\}) = \{-\} \qquad \alpha(\{-1,3,4\}) = \{+,-\}$$

$$\alpha(\mathbb{Z}) = \{0,+,-\} \qquad \alpha(\emptyset) = \emptyset$$

Concretization function γ

- Conversely, each element in the abstract domain represents one or several elements of the concrete domain.
- The **concretization function** γ is a function $\mathcal{P}(\mathsf{Sign}) \to \mathcal{P}(\mathbb{Z})$ that maps every element of the abstract domain to the biggest element in the concrete domain represented by it.
- In our example:

$$
\begin{array}{lcl@{\qquad}lcl}
\gamma(\{0,+,-\}) & = & \mathbb{Z} & \gamma(\{-\}) & = & \mathbb{Z}^- \\
\gamma(\{0,-\}) & = & \mathbb{Z}^- \cup \{0\} & \gamma(\{0\}) & = & \{0\} \\
\gamma(\{0,+\}) & = & \mathbb{Z}^+ \cup \{0\} & \gamma(\{+\}) & = & \mathbb{Z}^+ \\
\gamma(\{+,-\}) & = & \mathbb{Z}^- \cup \mathbb{Z}^+ & \gamma(\emptyset) & = & \emptyset
\end{array}
$$


γ
γ
$\mathbb{Z}$
$\mathbb{Z}^- \cup \{0\}$
$\mathbb{Z}^+$
$\emptyset$
$\{0, +, -\}$
$\{0, -\}$
$\{+, -\}$
$\{0, +\}$
$\{-\}$
$\{0\}$
$\{+\}$
$\emptyset$
γ
γ
$(\mathcal{P}(\mathbb{Z}), \subseteq)$
$(\mathcal{P}(\mathsf{Sign}), \subseteq)$

Deriving abstract interpreters

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\mathcal{P}(\mathbb{Z}) \arrow[r, "\llbracket e \rrbracket^*"] \arrow[d, shift left=2, "\alpha"] & \mathcal{P}(\mathbb{Z}) \arrow[d, shift left=2, "\alpha"] & \text{Concrete domain} \\
\mathcal{P}(\mathsf{Sign}) \arrow[r, "\llbracket e \rrbracket^\sharp"] \arrow[u, shift left=2, "\gamma"] & \mathcal{P}(\mathsf{Sign}) \arrow[u, shift left=2, "\gamma"] & \text{Abstract domain}
\end{tikzcd}$$

- We have started by defining a collecting semantics that deals with sets of integers (concrete domain).
- We have separately given another semantic definition dealing with signs (abstract domain).
- But the abstract and concrete domains are related by means of the α and γ functions.

$$\begin{tikzcd}
\mathcal{P}(\mathbb{Z}) \arrow[rr, "\llbracket e \rrbracket^*"] \arrow[d, shift left, "\alpha"] & & \mathcal{P}(\mathbb{Z}) \arrow[d, shift left, "\alpha"] & \text{Concrete domain} \\
\mathcal{P}(\mathsf{Sign}) \arrow[rr, "\llbracket e \rrbracket^\sharp"] \arrow[u, shift left, "\gamma"] & & \mathcal{P}(\mathsf{Sign}) \arrow[u, shift left, "\gamma"] & \text{Abstract domain}
\end{tikzcd}$$

This theoretical framework allows us to derive the abstract interpretation function $\llbracket e \rrbracket^\sharp$ from the following ingredients:

- Collecting semantics: $\llbracket e \rrbracket^*$.
- Abstraction and concretization functions: α and γ.

$$\begin{tikzcd} \mathcal{P}(\mathbb{Z}) \arrow[rr, "[\![e]\!]^*"] & & \mathcal{P}(\mathbb{Z}) \arrow[d, "\alpha"] & \text{Concrete domain} \\ \mathcal{P}(\mathsf{Sign}) \arrow[u, "\gamma"] \arrow[rr, "???"] & & \mathcal{P}(\mathsf{Sign}) & \text{Abstract domain} \end{tikzcd}$$

- Assume we want to define $[\![e]\!]^\sharp\ S$ for a given $S \in \mathcal{P}(\mathsf{Sign})$. We have to:
  1. Make S concrete (γ).
  2. Apply the concrete semantics ($[\![e]\!]^*$).
  3. Make the result abstract (α).

$$
\begin{tikzcd}
\mathcal{P}(\mathbb{Z}) \arrow[rr, "[\![e]\!]^*"] & & \mathcal{P}(\mathbb{Z}) \arrow[d, "\alpha"] & \text{Concrete domain} \\
\mathcal{P}(\mathbf{Sign}) \arrow[u, "\gamma"] \arrow[rr, "???"] & & \mathcal{P}(\mathbf{Sign}) & \text{Abstract domain}
\end{tikzcd}
$$

- Therefore, $[\![e]\!]^\sharp\ S$ has to yield something that is greater or equal than $\alpha([\![e]\!]^*\ (\gamma(S)))$.
- Equivalently, for any S, $[\![e]\!]^\sharp\ S$ has to be greater or equal than $(\alpha \circ [\![e]\!]^* \circ \gamma)(S)$.
- Equivalently we have to define $[\![e]\!]^\sharp$ such that it is greater or equal than $\alpha \circ [\![e]\!]^* \circ \gamma$.

Why cannot we just define the interpreter as follows?

$$[\![e]\!]^\sharp = \alpha \circ [\![e]\!]^* \circ \gamma$$

- Let us put this into practice with a simple function $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ defined as $f(x, y) = x + y$ for each $x, y \in \mathbb{Z}$.
- The collecting semantics $f^* : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ would be defined as follows:

$$f^*(X, Y) = \{x + y \mid x \in X, y \in Y\}$$

for every $X, Y \in \mathcal{P}(\mathbb{Z})$.

- How would be derive an abstract semantics $f^\sharp$ that deals with signs instead of sets of integers?

$$f^\sharp : \mathcal{P}(\mathbf{Sign}) \times \mathcal{P}(\mathbf{Sign}) \to \mathcal{P}(\mathbf{Sign})$$

A simple example

$$\begin{tikzcd}
\mathcal{P}(\mathbb{Z}) \arrow[r, "f^*"] & \mathcal{P}(\mathbb{Z}) \arrow[d, "\alpha"] \\
\mathcal{P}(\textsf{Sign}) \arrow[u, "\gamma"] \arrow[r, "f^\sharp"] & \mathcal{P}(\textsf{Sign})
\end{tikzcd}$$

Concrete domain

Abstract domain

- For example, what would $f^\sharp(\{0,+\},\{+\})$ be?
  1. Concrete parameters: $(\{0,+\},\{+\}) \xrightarrow{\gamma} (\mathbb{Z}^+ \cup \{0\}, \mathbb{Z}^+)$.
  2. Apply
     $f^*(\mathbb{Z}^+ \cup \{0\}, \mathbb{Z}^+) = \{x + y \mid x \in \mathbb{Z}^+ \cup \{0\}, y \in \mathbb{Z}^+\} = \mathbb{Z}^+$.
  3. Make the result abstract: $\mathbb{Z}^+ \cup \{0\} \xrightarrow{\alpha} \{+\}$
- Therefore, $f^\sharp(\{0,+\},\{+\}) = \{+\}$, which coincides with $\{0,+\} \oplus \{+\}$ as defined before.

Arithmetic expressions with several variables

- Let us change the definition of **AExp**:

$$e ::= n \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2$$

  where $n \in \mathbb{Z}$ and x is no longer a fixed variable, but a variable in a set **Var**.
- Now we group the values of the variables in a state σ, which is a function **Var** $\to \mathbb{Z}$.
- Let us denote by **State** the set of states.
- Recall that, for any e, $[\![e]\!]$ is a function **State** $\to \mathbb{Z}$.

$$
\begin{aligned}
[\![n]\!] &= \lambda\sigma.n \\
[\![x]\!] &= \lambda\sigma.\sigma(x) \\
[\![e_1 + e_2]\!] &= \lambda\sigma.\left([\![e_1]\!]\ \sigma\right) + \left([\![e_2]\!]\ \sigma\right) \\
[\![e_1 * e_2]\!] &= \lambda\sigma.\left([\![e_1]\!]\ \sigma\right) * \left([\![e_2]\!]\ \sigma\right) \\
[\![e_1 - e_2]\!] &= \lambda\sigma.\left([\![e_1]\!]\ \sigma\right) - \left([\![e_2]\!]\ \sigma\right)
\end{aligned}
$$

Example

$$[\![(3 * x) + y]\!] = \lambda\sigma.\, 3\sigma(x) + \sigma(y)$$

In particular:

$$[\![(3 * x) + y]\!]\ [x \mapsto 5, y \mapsto -3] = 12$$

- Collecting semantics has to manage sets of values.
- Here we have two possibilities:
    1. Manage functions $\mathsf{Var} \to \mathcal{P}(\mathbb{Z})$
        - That is the state specifies, for every variable, which values may take. For example:

$$\sigma = [x \mapsto \{1,5\}, y \mapsto \{3\}, z \mapsto \{-2,0\}]$$

    2. Manage sets of states, so we have $\mathcal{P}(\mathsf{Var} \to \mathbb{Z})$
        - That is, manage all possible states. For example:

$$\{[x \mapsto 1, y \mapsto 3, z \mapsto -2], [x \mapsto 5, y \mapsto 3, z \mapsto 0]\}$$

Which one is better?

$$[\![e]\!]^* : (\mathsf{Var} \to \mathcal{P}(\mathbb{Z})) \to \mathcal{P}(\mathbb{Z})$$

$$[\![e]\!]^* : \mathcal{P}(\mathsf{State}) \to \mathcal{P}(\mathbb{Z})$$

- Which of the following is the most accurate?
  1. **Var** $\to \mathcal{P}(\mathbb{Z})$

$$\sigma = [x \mapsto \{1,5\}, y \mapsto \{3\}, z \mapsto \{-2,0\}]$$

  2. $\mathcal{P}(\textbf{State})$, that is, $\mathcal{P}(\textbf{Var} \to \mathbb{Z})$

$$\{[x \mapsto 1, y \mapsto 3, z \mapsto -2], [x \mapsto 5, y \mapsto 3, z \mapsto 0]\}$$

- The first one includes all the states specified by the second one, plus some additional states:

$$[x \mapsto 5, y \mapsto 3, z \mapsto 0] \quad [x \mapsto 1, y \mapsto 3, z \mapsto -2]$$

- Therefore, by using $\mathcal{P}(\textbf{State})$ we get more accurate results.
- But analyses **become more costly**, due to the combinatorial explosion.

- In the following we shall use **Var** $\to \mathcal{P}(\mathbb{Z})$. Let us denote by **State*** the set of these functions.
- Therefore:

$$[\![e]\!]^* : \textbf{State}^* \to \mathcal{P}(\mathbb{Z})$$

- If σ denotes an alement of **State***, the collecting semantics is defined as follows:

$$\begin{aligned}
[\![n]\!]^* &= \lambda\sigma.\ \{n\} \\
[\![x]\!]^* &= \lambda\sigma.\ \sigma(x) \\
[\![e_1 + e_2]\!]^* &= \lambda\sigma.\ \{y + z \mid y \in [\![e_1]\!]^*\ \sigma, z \in [\![e_2]\!]^*\ \sigma\} \\
[\![e_1 * e_2]\!]^* &= \lambda\sigma.\ \{y * z \mid y \in [\![e_1]\!]^*\ \sigma, z \in [\![e_2]\!]^*\ \sigma\} \\
[\![e_1 - e_2]\!]^* &= \lambda\sigma.\ \{y - z \mid y \in [\![e_1]\!]^*\ \sigma, z \in [\![e_2]\!]^*\ \sigma\}
\end{aligned}$$

- This semantic definition involves loss of precision.

Example
Let $\sigma = [x \mapsto \{3,5\}]$:

$$\begin{aligned} [\![x + x]\!]^* \ \sigma \quad &= \{y + z \mid y \in \{3,5\}, z \in \{3,5\}\} \\ &= \{9, 8, 10\} \end{aligned}$$

- We could have given a more precise definition:

$$[\![e]\!]^* \ \sigma' = \{[\![e]\!] \ \sigma \mid \sigma \in \mathbf{State}, \sigma \preceq \sigma'\}$$

  where $\sigma \preceq \sigma'$ means that $\sigma(x) \in \sigma'(x)$ for all x.
  - However, we stick to the definition of the previous slide.
  - If we wanted to apply this modification, it would be better to use $\mathcal{P}(\mathbf{State})$.

If we start from the following state:

$$\sigma = [x \mapsto \{1, 7\}, y \mapsto \{3\}, z \mapsto \{-2, 1\}]$$

then

$$[\![2 * x + y - z]\!]^* \sigma =$$

- Our states in the abstract domain have to associate variables with signs, instead of integers.
- Again, we have two possibilities:
  1. Manage functions $\mathsf{Var} \to \mathcal{P}(\mathsf{Sign})$
     - That is, states specify, for every variable, the set of possible signs. For example:

$$\sigma = [x \mapsto \{+\}, y \mapsto \{+\}, z \mapsto \{0, -\}]$$

  2. Manage sets of "abstract" states, that is, $\mathcal{P}(\mathsf{Var} \to \mathsf{Sign})$
     - For example:

$$\{[x \mapsto +, y \mapsto +, z \mapsto -], [x \mapsto +, y \mapsto +, z \mapsto 0]\}$$

- We use the first option, because:
  - It is computationally simpler, although less accurate.
  - It is consistent with our choice for the concrete domain.

- Therefore, an abstract state is a function that maps each variable to a set of signs.
  - That is, a function $\sigma^\sharp : \mathbf{Var} \to \mathcal{P}(\mathbf{Sign})$.
  - We denote by $\mathbf{State}^\sharp$ the set of abstract states.
- Here we get: $[\![e]\!]^\sharp : \mathbf{State}^\sharp \to \mathcal{P}(\mathbf{Sign})$.

$$
\begin{aligned}
[\![n]\!]^\sharp &= \lambda\sigma^\sharp.\begin{cases}\{+\} & \text{si } \mathcal{N}(n) > 0\\ \{-\} & \text{si } \mathcal{N}(n) < 0\\ \{0\} & \text{si } \mathcal{N}(n) = 0\end{cases}\\
[\![x]\!]^\sharp &= \lambda\sigma^\sharp.\,\sigma^\sharp(x)\\
[\![e_1 + e_2]\!]^\sharp &= \lambda\sigma^\sharp.\,([\![e_1]\!]^\sharp\,\sigma^\sharp) \oplus ([\![e_2]\!]^\sharp\,\sigma^\sharp)\\
[\![e_1 * e_2]\!]^\sharp &= \lambda\sigma^\sharp.\,([\![e_1]\!]^\sharp\,\sigma^\sharp) \otimes ([\![e_2]\!]^\sharp\,\sigma^\sharp)\\
[\![e_1 - e_2]\!]^\sharp &= \lambda\sigma^\sharp.\,([\![e_1]\!]^\sharp\,\sigma^\sharp) \ominus ([\![e_2]\!]^\sharp\,\sigma^\sharp)
\end{aligned}
$$

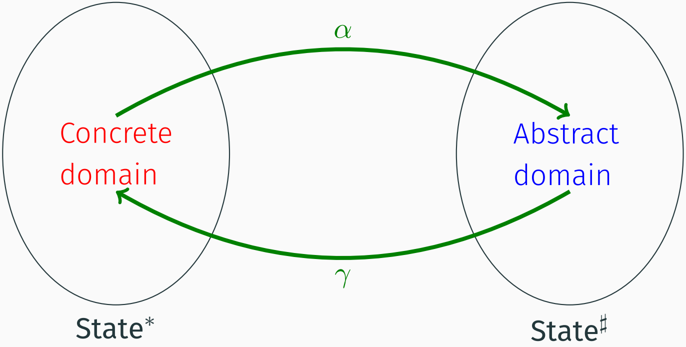
Example

- Assume the following abstract state $\sigma^\sharp$:

$$\sigma^\sharp = [x \mapsto \{0\}, y \mapsto \{-,+\}, z \mapsto \{+\}]$$

- We want to compute the sign of the expression $(x * y) + z$ under $\sigma^\sharp$:

$$\begin{aligned}
[\![(x * y) + z]\!]^\sharp\,\sigma^\sharp &= \left([\![x]\!]^\sharp\,\sigma^\sharp \otimes [\![y]\!]^\sharp\,\sigma^\sharp\right) \oplus [\![z]\!]^\sharp\,\sigma^\sharp \\
&= (\{0\} \otimes \{-,+\}) \oplus \{+\} \\
&= \{0\} \oplus \{+\} \\
&= \{+\}
\end{aligned}$$



- We have introduced a new concrete domain (State^*) and a new abstract domain and a new abstract domain ($\mathsf{State}^\sharp$).
- Both $\mathsf{Var} \to \mathcal{P}(\mathbb{Z})$ and $\mathsf{Var} \to \mathcal{P}(\mathsf{Sign})$ are lattices.
- Both are related by means of abstraction and concretization functions α and γ.

- Assume the following collecting semantics state:

$$\sigma = [x \mapsto \{3\}, y \mapsto \{0, 3, -6\}, z \mapsto \{-4, 5\}]$$

- We can transform it in an abstract state by replacing each set by the signs of the integers contained within:

$$\sigma^\sharp = [x \mapsto \{+\}, y \mapsto \{0, -, +\}, z \mapsto \{-, +\}]$$

- Therefore α receives a collecting semantics state and applies this transformation to get an abstract state.

- Assume the following abstract state:

$$\sigma^\sharp = [x \mapsto \{+\}, y \mapsto \{0\}, z \mapsto \{-, 0\}]$$

- This state represents all the concrete states that map x to the empty set or a set of positive numbers, y to an empty set or the set $\{0\}$, and z to an empty set or a set containing negative numbers and/or zero.
    - In particular, it approximates the following state:

$$[x \mapsto \{4, 7, 1\}, y \mapsto \{0\}, z \mapsto \{-3, -4\}]$$

- Among all of them, the greatest one is:

$$\left[x \mapsto \mathbb{Z}^+, y \mapsto \{0\}, z \mapsto \mathbb{Z}^- \cup \{0\}\right]$$

- The concretization function γ receives an abstract state $\sigma^\sharp$ and computes the greatest collecting state approximated by $\sigma^\sharp$.

Deriving abstract interpreters

$$
\begin{tikzcd}
\mathsf{State}^{*} \arrow[rr, "\llbracket e \rrbracket^{*}"] \arrow[d, shift left, "\alpha"] & & \mathcal{P}(\mathbb{Z}) \arrow[d, shift left, "\alpha"] & \text{Concrete domain} \\
\mathsf{State}^{\sharp} \arrow[rr, "\llbracket e \rrbracket^{\sharp}"] \arrow[u, shift left, "\gamma"] & & \mathcal{P}(\mathsf{Sign}) \arrow[u, shift left, "\gamma"] & \text{Abstract domain}
\end{tikzcd}
$$

- We have a collecting semantics dealing with concrete states and concrete numbers.
- We have an abstract semantics dealing with abstract states and signs.
- Both domains are related by means of α y γ.

Deriving abstract interpreters

$$
\begin{tikzcd}
\mathsf{State}^* \arrow[rr, "[\![e]\!]^*"] & & \mathcal{P}(\mathbb{Z}) \arrow[d, "\alpha"] & \text{Concrete domain} \\
\mathsf{State}^\sharp \arrow[u, "\gamma"] \arrow[rr, "???"] & & \mathcal{P}(\mathsf{Sign}) & \text{Abstract domain}
\end{tikzcd}
$$

- We can derive the abstract interpreter by:
    1. Apply γ to the input abstract state.
    2. Apply the concrete semantics ($[\![e]\!]^*$).
    3. Apply α to make the result abstract.

- Let **BExp** denote the set of boolean expressions:

$$b ::= true \mid false \mid e_1 = e_2 \mid e_1 \leq e_2 \mid \neg b \mid b_1 \wedge b_2$$

- The semantics of an expression b is a function **State** $\rightarrow \mathbb{B}$, where $\mathbb{B} = \{true, false\}$.
- The collecting semantics of an expression b, denoted by $[\![b]\!]^*$ is a function that:
  - Receives a state that maps each variable to a set of possible values.
  - Returns a set of boolean values to which it may evaluate.
- Therefore, $[\![b]\!]^*$ is a function:

$$[\![b]\!]^* : \textbf{State}^* \rightarrow \mathcal{P}(\mathbb{B})$$

Collecting semantics

$$
\begin{aligned}
[\![true]\!]^* &= \lambda\sigma.\,\{true\} \\
[\![false]\!]^* &= \lambda\sigma.\,\{false\} \\
[\![e_1 = e_2]\!]^* &= \lambda\sigma.\,\{eq(x,y) \mid x \in ([\![e_1]\!]^*\ \sigma)\,, y \in ([\![e_2]\!]^*\ \sigma)\} \\
[\![e_1 \leq e_2]\!]^* &= \lambda\sigma.\,\{le(x,y) \mid x \in ([\![e_1]\!]^*\ \sigma)\,, y \in ([\![e_2]\!]^*\ \sigma)\} \\
[\![\neg b]\!]^* &= \lambda\sigma.\,\{\neg v \mid v \in ([\![b]\!]^*\ \sigma)\} \\
[\![b_1 \wedge b_2]\!]^* &= \lambda\sigma.\,\{v_1 \wedge v_2 \mid v_1 \in ([\![b_1]\!]^*\ \sigma)\,, v_2 \in ([\![b_2]\!]^*\ \sigma)\}
\end{aligned}
$$

where

$$
eq(x,y) = \begin{cases} true & \text{if } x = y \\ false & \text{otherwise} \end{cases} \qquad le(x,y) = \begin{cases} true & \text{if } x \leq y \\ false & \text{otherwise} \end{cases}
$$

Example

- If σ is the following collecting state:

$$\sigma = [x \mapsto \{0, 9\}, y \mapsto \{-3\}]$$

- Let us evaluate the following boolean expressions:

$$\begin{array}{lcl}
[\![x \leq 10]\!]^* \ \sigma & = & \{\mathit{true}\} \\
[\![x \leq 10 \land \neg y \leq 0]\!]^* \ \sigma & = & \{\mathit{false}\} \\
[\![x \leq 10 \land y \leq 0]\!]^* \ \sigma & = & \{\mathit{true}\} \\
[\![x \leq 1]\!]^* \ \sigma & = & \{\mathit{true}, \mathit{false}\}
\end{array}$$

What would an abstract interpreter for boolean expressions receive? What would it return?

$$[\![b]\!]^\sharp : ??? \rightarrow ???$$

- The abstract semantics receives abstract states ($\mathsf{State}^\sharp$) instead of concrete states.
- The collecting semantics returns sets of boolean values, but this set is simple enough. There is no need for an abstract domain describing the result.
- Therefore, the abstract semantics of a boolean expression is a function:

$$[\![b]\!]^\sharp : \mathsf{State}^\sharp \to \mathcal{P}(\mathbb{B})$$

$$
\begin{aligned}
[\![true]\!]^\sharp &= \lambda\sigma^\sharp.\,\{true\}\\
[\![false]\!]^\sharp &= \lambda\sigma^\sharp.\,\{false\}\\
[\![e_1 = e_2]\!]^\sharp &= \lambda\sigma^\sharp.eq^\sharp([\![e_1]\!]^\sharp\ \sigma^\sharp, [\![e_2]\!]^\sharp\ \sigma^\sharp)\\
[\![e_1 \leq e_2]\!]^\sharp &= \lambda\sigma^\sharp.le^\sharp([\![e_1]\!]^\sharp\ \sigma^\sharp, [\![e_2]\!]^\sharp\ \sigma^\sharp)\\
[\![\neg b]\!]^\sharp &= \lambda\sigma^\sharp.not^\sharp([\![b]\!]^\sharp\ \sigma^\sharp)\\
[\![b_1 \wedge b_2]\!]^\sharp &= \lambda\sigma^\sharp.and^\sharp([\![b_1]\!]^\sharp\ \sigma^\sharp, [\![b_2]\!]^\sharp\ \sigma^\sharp)
\end{aligned}
$$

where the $eq^\sharp$, $le^\sharp$, $not^\sharp$, $and^\sharp$ functions are the abstract variants of the concrete operators *eq*, *le*, $\neg$ and $\wedge$, respectively. They are described in the following tables.

Function $eq^\sharp$

$eq^\sharp(\downarrow, \rightarrow)$	$\emptyset$	$\{0\}$	$\{-\}$	$\{+\}$	$\{0,-\}$	$\{0,+\}$	$\{-,+\}$	$\{0,-,+\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{0\}$	$\emptyset$	$\{true\}$	$\{false\}$	$\{false\}$	$\mathbb{B}$	$\mathbb{B}$	$\{false\}$	$\mathbb{B}$
$\{-\}$	$\emptyset$	$\{false\}$	$\mathbb{B}$	$\{false\}$	$\mathbb{B}$	$\{false\}$	$\mathbb{B}$	$\mathbb{B}$
$\{+\}$	$\emptyset$	$\{false\}$	$\{false\}$	$\mathbb{B}$	$\{false\}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
$\{0,-\}$	$\emptyset$	$\mathbb{B}$	$\mathbb{B}$	$\{false\}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
$\{0,+\}$	$\emptyset$	$\mathbb{B}$	$\{false\}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
$\{-,+\}$	$\emptyset$	$\{false\}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
$\{0,-,+\}$	$\emptyset$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$

Function $le^\sharp$

$le^\sharp(\downarrow,\rightarrow)$	$\emptyset$	$\{0\}$	$\{-\}$	$\{+\}$	$\{0,-\}$	$\{0,+\}$	$\{-,+\}$	$\{0,-,+\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{0\}$	$\emptyset$	$\{true\}$	$\{false\}$	$\{true\}$	$\mathbb{B}$	$\{true\}$	$\mathbb{B}$	$\mathbb{B}$
$\{-\}$	$\emptyset$	$\{true\}$	$\mathbb{B}$	$\{true\}$	$\mathbb{B}$	$\{true\}$	$\mathbb{B}$	$\mathbb{B}$
$\{+\}$	$\emptyset$	$\{false\}$	$\{false\}$	$\mathbb{B}$	$\{false\}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
$\{0,-\}$	$\emptyset$	$\{true\}$	$\mathbb{B}$	$\{true\}$	$\mathbb{B}$	$\{true\}$	$\mathbb{B}$	$\mathbb{B}$
$\{0,+\}$	$\emptyset$	$\mathbb{B}$	$\{false\}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
$\{-,+\}$	$\emptyset$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$
$\{0,-,+\}$	$\emptyset$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$	$\mathbb{B}$

Functions $not^\sharp$ and $and^\sharp$

B	$not^\sharp(B)$
$\emptyset$	$\emptyset$
$\{true\}$	$\{false\}$
$\{false\}$	$\{true\}$
$\mathbb{B}$	$\mathbb{B}$

$and^\sharp$	$\emptyset$	$\{true\}$	$\{false\}$	$\mathbb{B}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{true\}$	$\emptyset$	$\{true\}$	$\{false\}$	$\mathbb{B}$
$\{false\}$	$\emptyset$	$\{false\}$	$\{false\}$	$\{false\}$
$\mathbb{B}$	$\emptyset$	$\mathbb{B}$	$\{false\}$	$\mathbb{B}$

Deriving abstract interpreters

$$
\begin{tikzcd}
\mathsf{State}^* \arrow[r, "{[\![b]\!]^*}"] \arrow[d, shift left, "\alpha"] & \mathcal{P}(\mathbb{B}) & \text{Concrete domain} \\
\mathsf{State}^\sharp \arrow[r, "{[\![b]\!]^\sharp}"] \arrow[u, shift left, "\gamma"] & \mathcal{P}(\mathbb{B}) & \text{Abstract domain}
\end{tikzcd}
$$

- We have a collecting semantics that deals with concrete states and sets of booleans.
- We have an abstract semantics that deals with abstract states and sets of booleans.
- Concrete and abstract states are related by means of α and γ.

Deriving abstract interpreters

$$
\begin{tikzcd}
\mathrm{State}^* \arrow[rr, "{[\![b]\!]^*}"] & & \mathcal{P}(\mathbb{B}) \arrow[d, "id"] & \text{Concrete domain} \\
\mathrm{State}^\sharp \arrow[u, "\gamma"] \arrow[rr, "???"] & & \mathcal{P}(\mathbb{B}) & \text{Abstract domain}
\end{tikzcd}
$$

- We can derive $[\![b]\!]^\sharp$ by composing the following operations:
    1. Make input state concrete (γ).
    2. Apply collecting semantics ($[\![b]\!]^*$).
    3. Leave the result as is, since both domains coincide in the result.
- Summarizing, $[\![b]\!]^\sharp$ must be greater or equal than $[\![b]\!]^* \circ \gamma$.

- *While* syntax:

$$S ::= \texttt{skip} \mid x := e \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S$$

- For a given S, its semantics $[\![S]\!]$ is a function $\textbf{State} \to \textbf{State}$.
- Let us define a collecting semantics that maps every $\sigma \in \textbf{State}^*$ denoting an initial collecting state to another $\sigma \in \textbf{State}^*$ denoting another set of collecting states.
- So let us denote by $[\![S]\!]^*$ the collecting semantics:

$$[\![S]\!]^* : \textbf{State}^* \to \textbf{State}^*$$

- The `skip` does not change the state. Therefore:

$$[\![\texttt{skip}]\!]^* = \lambda\sigma.\,\sigma$$

where σ is an element from **State***.

- The collecting semantics for `skip` can also be expressed as follows:

$$[\![\texttt{skip}]\!]^* = \mathit{id}$$

where *id* is the identity function.

- Assume we want to execute $x := e$ under a collecting state $\sigma \in \mathsf{State}^*$.
    - We have to evaluate all possible values of e under σ.
    - We return the collecting state that results from updating x with these values.

$$[\![x := e]\!]^* = \lambda\sigma.\sigma\,[x \mapsto [\![e]\!]^*\,\sigma]$$

- The collecting semantics for $S_1; S_2$ is similar to the standard denotational semantics:

$$[\![S_1; S_2]\!]^* = [\![S_2]\!]^* \circ [\![S_1]\!]^*$$

- Assume we execute `if` b `then` S_1 `else` S_2 under $\sigma \in \mathbf{State}^*$.
- We evaluate $[\![b]\!]\ \sigma$. This may yield $\emptyset$, $\{true\}$, $\{false\}$, or $\{true, false\}$.
  - If it yields $\emptyset$, then we return the $\bot$ of $\mathbf{State}^*$.
  - If it yields $\{true\}$ or $\{false\}$, then we return the result of executing the corresponding branch.
  - If it returns $\{true, false\}$, we have to execute both branches and combine (join) the results.

$$[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]^* \quad = \lambda\sigma. \begin{cases} \bot & \text{if } [\![b]\!]^*\ \sigma = \emptyset \\ [\![S_1]\!]^*\ \sigma & \text{if } [\![b]\!]^*\ \sigma = \{true\} \\ [\![S_2]\!]^*\ \sigma & \text{if } [\![b]\!]^*\ \sigma = \{false\} \\ ([\![S_1]\!]^*\ \sigma) \sqcup ([\![S_2]\!]^*\ \sigma) & \text{if } [\![b]\!]^*\ \sigma = \mathbb{B} \end{cases}$$

Let us shorten the notation:

- Assume a complete lattice $(L, \sqsubseteq)$.
  - Let f denote a function $L \to \mathcal{P}(\mathbb{B})$.
  - Let g denote a function $L \to L$.
  - Let h denote a function $L \to L$.
- We denote by $cond_L(f, g, h)$ the function $L \to L$ defined as follows:

$$cond_L(f, g, h) = \lambda x. \begin{cases} \bot_L & \text{if } f(x) = \emptyset \\ g(x) & \text{if } f(x) = \{true\} \\ h(x) & \text{if } f(x) = \{false\} \\ g(x) \sqcup h(x) & \text{if } f(x) = \mathbb{B} \end{cases}$$

- Therefore:

$$[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]^* = cond_{\mathsf{State}^*}([\![b]\!]^*, [\![S_1]\!]^*, [\![S_2]\!]^*)$$

- For loops $\texttt{while}\ b\ \texttt{do}\ S$ we have an expression similar to that of standard semantics:

$$[\![\texttt{while}\ b\ \texttt{do}\ S]\!]^* = lfp\ \lambda f.cond_{\mathsf{State}^*}([\![b]\!]^*, f \circ [\![S]\!]^*, id)$$

- The function $\lambda f.cond_{\mathsf{State}^*}([\![b]\!]^*, f \circ [\![S]\!]^*, id)$ is monotonically increasing and continuous for every b and S, so the existence of the lfp is guaranteed.

Could we compute the least fixed point by using Kleene's ascending chain?

Collecting semantics: a summary

$$
\begin{array}{lcl}
[\![\texttt{skip}]\!]^* & = & id \\
[\![x := e]\!]^* & = & \lambda\sigma.\,\sigma\,[x \mapsto [\![e]\!]^*\,\sigma] \\
[\![S_1; S_2]\!]^* & = & [\![S_2]\!]^* \circ [\![S_1]\!]^* \\
[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]^* & = & cond_{\mathsf{State}^*}([\![b]\!]^*, [\![S_1]\!]^*, [\![S_2]\!]^*) \\
[\![\texttt{while}\ b\ \texttt{do}\ S]\!]^* & = & lfp\ (\lambda f.cond_{\mathsf{State}^*}([\![b]\!]^*, f \circ [\![S]\!]^*, id))
\end{array}
$$

COMPUTE $[\![S]\!]^* \sigma_i$ UNDER THE FOLLOWING STATES:

$$S \equiv \texttt{if}\ x \geq 0\ \texttt{then}\ y := 1\ \texttt{else}\ y := y + 1$$

$$\begin{aligned}
\sigma_1 &= [x \mapsto \{1, 3\}, y \mapsto \{3, 4\}] \\
\sigma_2 &= [x \mapsto \{-4\}, y \mapsto \{-2\}] \\
\sigma_3 &= [x \mapsto \{1, -5\}, y \mapsto \{0\}]
\end{aligned}$$

Compute $[\![S]\!]^* \sigma_i$ under the following state:

$$S \equiv \texttt{while}\ x \geq 0\ \textbf{do}\ (m := m * x; x := x - 1)$$

$$\sigma = [x \mapsto \{5, -3\}, m \mapsto \{1\}]$$

Here is how we did in Theory of Programming Languages:

1. Build the first elements of the ascending chain:

$$\perp \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \dots$$

2. Conjecture a generic f_i for each $i \in \mathbb{N}$, and prove that the conjecture is correct.
3. Infer the value of $\bigsqcup_i f_i$, which is the *lfp* by Fixed Point Theorem.

But now *cond* is more complicated!

A SHORTCUT: *computeWhile*

- The following semialgorithm allows us to compute the final collecting state of a `while` b `do` S loop assuming that the initial state σ_0 is known.
- It receives an input state σ and a set V of visited states.
- Initially, $\sigma = \sigma_0$, and $V = \emptyset$.

A SHORTCUT: *computeWhile*

```
method computeWhile(σ: State*, V: set<State*>)
returns State*
{
  if (σ ∈ V) then return ⊥;
  σ′ := ⟦S⟧* σ;
  case ⟦b⟧* σ {
    ∅             → return ⊥
    {false}       → return σ
    {true}        → return computeWhile(σ', V ∪ {σ})
    {true,false} → return σ ⊔ computeWhile(σ', V ∪ {σ})
  }
}
```

- Given the following program, we start from $\sigma_0 = [m \mapsto \{1, 2\}, n \mapsto \mathbb{Z}]$.

```
[m↦{1,2}, n↦ℤ]
n := 1;
[m↦{1,2}, n↦{1}]
while m > 0 do {  Condition: {true}
  [m↦{1,2}, n↦{1}]
  n := n * m;
  [m↦{1,2}, n↦{1,2}]
  m := m - 1
  [m↦{0,1}, n↦{1,2}]
}
```

- Given the following program, we start from $\sigma_0 = [m \mapsto \{1,2\}, n \mapsto \mathbb{Z}]$.

```
[m↦{1,2}]
n := 1;
[m↦{0,1}, n↦{1,2}]
while m > 0 do {  Cond: {true,false}
  [m↦{0,1}, n↦{1,2}]
  n := n * m;
  [m↦{0,1}, n↦{0,1,2}]
  m := m - 1
  [m↦{-1,0}, n↦{0,1,2}]
}
[m↦{0,1}, n↦{1,2}]⊔...
```

- Given the following program, we start from $\sigma_0 = [m \mapsto \{1,2\}, n \mapsto \mathbb{Z}]$.

```
[m↦{1,2}]
n := 1;
[m↦{-1,0}, n↦{0,1,2}]
while m > 0 do {  Condition: {false}

  n := n * m;

  m := m - 1

}
[m↦{0,1}, n↦{1,2}]⊔[m↦{-1,0}, n↦{0,1,2}]
= [m↦{-1,0,1}, n↦{0,1,2}]
```

- The abstract semantics of a program S, denoted by $[\![S]\!]^\sharp$ is a function that manages abstract states instead of concrete states.

$$\begin{array}{ll} \text{Collecting semantics:} & [\![S]\!]^* : \mathbf{State}^* \rightarrow \mathbf{State}^* \\ \text{Abstract semantics:} & [\![S]\!]^\sharp : \mathbf{State}^\sharp \rightarrow \mathbf{State}^\sharp \end{array}$$

- Recall that an abstract state is a function from variables to sets of signs.

$$\mathbf{State}^\sharp = \mathbf{Var} \rightarrow \mathcal{P}(\mathbf{Sign})$$

where $\mathbf{Sign} = \{0, +, -\}$.

Collecting semantics

$$
\begin{aligned}
[\![\texttt{skip}]\!]^* &= id \\
[\![x := e]\!]^* &= \lambda\sigma.\, \sigma\,[x \mapsto [\![e]\!]^*\, \sigma] \\
[\![S_1; S_2]\!]^* &= [\![S_2]\!]^* \circ [\![S_1]\!]^* \\
[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]^* &= cond_{\mathsf{State}^*}([\![b]\!]^*, [\![S_1]\!]^*, [\![S_2]\!]^*) \\
[\![\texttt{while } b \texttt{ do } S]\!]^* &= lfp\ \ (\lambda f.cond_{\mathsf{State}^*}([\![b]\!]^*, f \circ [\![S]\!]^*, id))
\end{aligned}
$$

Abstract semantics

$$
\begin{aligned}
[\![\texttt{skip}]\!]^\sharp &= id \\
[\![x := e]\!]^\sharp &= \lambda\sigma.\, \sigma\,\left[x \mapsto [\![e]\!]^\sharp\, \sigma\right] \\
[\![S_1; S_2]\!]^\sharp &= [\![S_2]\!]^\sharp \circ [\![S_1]\!]^\sharp \\
[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]^\sharp &= cond_{\mathsf{State}^\sharp}([\![b]\!]^\sharp, [\![S_1]\!]^\sharp, [\![S_2]\!]^\sharp) \\
[\![\texttt{while } b \texttt{ do } S]\!]^\sharp &= lfp\ \ \left(\lambda f.cond_{\mathsf{State}^\sharp}([\![b]\!]^\sharp, f \circ [\![S]\!]^\sharp, id)\right)
\end{aligned}
$$

Does $\mathsf{State}^\sharp$ satisfy the ascending chain condition?

Does it matter?

Is $[\![S]\!]^\sharp$ computable?

- Despite the above, using Kleene's ascending chain is rather costly.
- Recall that we are computing the least fixed point of the following function F:

$$F(f) = cond_{\mathsf{State}^\sharp}([\![b]\!]^\sharp, f \circ [\![S]\!]^\sharp, id)$$

- It turns out that F is a function from state transformers into state transformers:

$$F : (\mathsf{State}^\sharp \to \mathsf{State}^\sharp) \to (\mathsf{State}^\sharp \to \mathsf{State}^\sharp)$$

- In order to check that f_0 is a fixed point of F, we have to compare two state transformers.
    - That means that we have to check whether that $f_0(\sigma) = F(f_0)(\sigma)$ for every $\sigma \in \mathsf{State}^\sharp$!

- However, we can apply `computeWhile` if we know the initial abstract state under which the loop is going to be analysed.
- In this case, the `computeWhile` semialgorithm becomes an algorithm, because it always terminates.

Why does it terminate?

- Let S be the program below. We start from $\sigma^\sharp = [m \mapsto \{+\}, n \mapsto \{+\}]$:

```
[n↦{+}, m↦{+}]
z := 1;
[n↦{+}, m↦{+}, z↦{+}]
while m > 0 do { Cond: {true}
  [n↦{+}, m↦{+}, z↦{+}]
  z := z * n;
  [n↦{+}, m↦{+}, z↦{+}]
  m := m - 1
  [n↦{+}, m↦{0,-,+}, z↦{+}]
}
```

- Let S be the program below. We start from $\sigma^\sharp = [m \mapsto \{+\}, n \mapsto \{+\}]$:

```
[n↦{+}, m↦{+}]
z := 1;
[n↦{+}, m↦{0,-,+}, z↦{+}]
while m > 0 do { Cond: {true,false}
  [n↦{+}, m↦{0,-,+}, z↦{+}]
  z := z * n;
  [n↦{+}, m↦{0,-,+}, z↦{+}]
  m := m - 1
  [n↦{+}, m↦{0,-,+}, z↦{+}]
}
[n↦{+}, m↦{0,-,+}, z↦{+}]⊔...
```

- Let S be the program below. We start from $\sigma^\sharp = [m \mapsto \{+\}, n \mapsto \{+\}]$:

```
[n↦{+}, m↦{+}]
z := 1;
[n↦{+}, m↦{0,-,+}, z↦{+}] Visited
while m > 0 do {

  z := z * n;

  m := m - 1

}
[n↦{+}, m↦{0,-,+}, z↦{+}]⊔⊥
= [n↦{+}, m↦{0,-,+}, z↦{+}]
```

- Now another program under $\sigma^\sharp = [m \mapsto \{+\}]$.

```
[m↦{+}]
n := 1;
[m↦{+},n↦{+}]
while m > 0 do { Cond: {true}
  [m↦{+},n↦{+}]
  n := n * m;
  [m↦{+},n↦{+}]
  m := m - 1
  [m↦{0,-,+},n↦{+}]
}
```

- Now another program under $\sigma^\sharp = [m \mapsto \{+\}]$.

```
[m↦{+}]
n := 1;
[m↦{0,-,+},n↦{+}]
while m > 0 do { Cond: {true,false}
  [m↦{0,-,+},n↦{+}]
  n := n * m;
  [m↦{0,-,+},n↦{0,-,+}]
  m := m - 1
  [m↦{0,-,+},n↦{0,-,+}]
}
[m↦{0,-,+},n↦{+}]⊔...
```

Example

- Now another program under $\sigma^\sharp = [m \mapsto \{+\}]$.

```
[m↦{+}]
n := 1;
[m↦{0,-,+},n↦{0,-,+}]
while m > 0 do { Cond: {true,false}
  [m↦{0,-,+},n↦{0,-,+}]
  n := n * m;
  [m↦{0,-,+},n↦{0,-,+}]
  m := m - 1
  [m↦{0,-,+},n↦{0,-,+}]
}
[m↦{0,-,+},n↦{+}]⊔[m↦{0,-,+},n↦{0,-,+}]
```

- Now another program under $\sigma^\sharp = [m \mapsto \{+\}]$.

```
[m↦{+}]
n := 1;
[m↦{0,-,+},n↦{0,-,+}] Visited
while m > 0 do {

  n := n * m;

  m := m - 1

}
[m↦{0,-,+},n↦{+}]⊔[m↦{0,-,+},n↦{0,-,+}]⊔⊥
= [m↦{0,-,+},n↦{0,-,+}]
```

- The result given by $\llbracket S \rrbracket^\sharp\ \sigma^\sharp$ in the previous example is rather inaccurate.
- The analysis cannot prove that the factorial of a positive number computed in this way is positive.
- There are refinements that allow one to obtain a more refined result:

$$[m \mapsto \{0, -\}, n \mapsto \{+\}]$$

- They consist in discarding the information that is not compatible with the conditions of the loop.

H.R. Nielson, F. Nielson
Semantics with applications: an appetizer
Springer, 2007

- Back to the previous program with $\sigma^\sharp = [m \mapsto \{+\}]$:

```
[m↦{+}]
n := 1;
[m↦{+},n↦{+}]
while m > 0 do { Cond: {true}
  [m↦{+},n↦{+}]
  n := n * m;
  [m↦{+},n↦{+}]
  m := m - 1
  [m↦{0,-,+},n↦{+}]
}
```

- Back to the previous program with $\sigma^\sharp = [m \mapsto \{+\}]$:

```
[m↦{+}]
n := 1;
[m↦{0,-,+},n↦{+}]
while m > 0 do { Cond: {true,false}
  [m↦{0,-,+},n↦{+}][m↦{+},n↦{+}]
  n := n * m;
  [m↦{+},n↦{+}]
  m := m - 1
  [m↦{0,-,+},n↦{+}]
}
[m↦{0,-,+},n↦{+}]⊔...[m↦{0,-},n↦{+}]⊔...
```

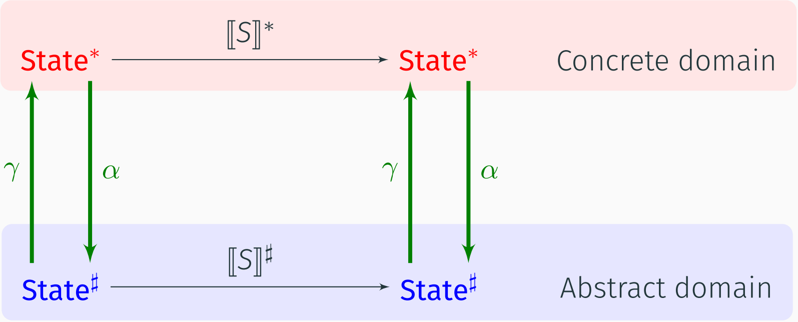
- Back to the previous program with $\sigma^\sharp = [m \mapsto \{+\}]$:

```
[m↦{+}]
n := 1;
[m↦{0,-,+},n↦{+}] Visited
while m > 0 do {

  n := n * m;

  m := m - 1

}
[m↦{0,-},n↦{+}]⊔⊥
= [m↦{0,-},n↦{+}]
```



- We have a semantics that deals with concrete states.
- We have another semantics that deals with abstract states.
- Both are related by means of α and γ.

Deriving abstract interpreters

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
\text{State}^{*} \arrow[r, "\llbracket S \rrbracket^{*}"] & \text{State}^{*} \arrow[d, "\alpha"] & \text{Concrete domain} \\
\text{State}^{\sharp} \arrow[u, "\gamma"] \arrow[r, "???"] & \text{State}^{\sharp} & \text{Abstract domain}
\end{tikzcd}
$$

- We can derive the abstract interpretation as follows:
    1. Make input state concrete (γ).
    2. Apply concrete semantics ($\llbracket S \rrbracket^{*}$).
    3. Make the result abstract (α).

Application to data-flow analyses

3. Application to data-flow analyses

Review: monotone frameworks

Concrete monotone frameworks

Deriving abstract monotone frameworks

Motivation

- We can apply abstract interpretation to data flow analysis.
- This is particularly useful if we are interested in properties **at each program point**.
- It also provides an alternative way of dealing with loops, without having to use *computeWhile*.

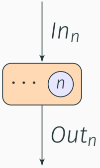
In order to get a monotone framework instance for a given program we need the following ingredients:

- A lattice $(L, \sqsubseteq)$ of **properties**.
  - In this lesson we do not require ascending chain condition.
- An **extremal value** ι.
- A monotonically increasing **transfer function** $f_n : L \to L$ for each block.
  - where $n \in \mathtt{Lab}$, which is the set of block labels.

Recall that:

- We assume that there is a single entry block in our CFG (add a `skip` block if necessary).

- For each block $n \in \textsf{Lab}$ we want to compute $In_n, Out_n \in L$ indicating the values of the property we want to study before and after each block.



- This yields the following system of equations:

$$In_n = \begin{cases} \iota & \text{if } n \text{ is initial block} \\ \bigsqcup \{Out_m \mid m \in pred(n)\} & \text{otherwise} \end{cases}$$

$$Out_n = f_n\left(In_n\right)$$

- We can replace the system of equations by a system of constraints:

$$In_n \sqsupseteq \begin{cases} \iota & \text{if } n \text{ is initial block} \\ \bigsqcup\{Out_m \mid m \in pred(n)\} & \text{otherwise} \end{cases}$$

$$Out_n \sqsupseteq f_n\,(In_n)$$

- If **Lab** $= \{1..m\}$, a solution is a tuple of $2m$ components

$$(In_1, In_2, \ldots, In_m, Out_1, Out_2, \ldots, Out_m) \in L^{2m}$$

satisfying all the constraints.

Which of the following statements hold?

- Every solution of the system of constraints $\sqsupseteq$ is a solution of the system of equations $(=)$.

  -

- Every solution of the system of equations $=$ is a solution of the system of constraints $(\sqsupseteq)$.

  -

- The system of constraints $\sqsupseteq$ always has a solution.

  -

- The solution of the system of equations $=$ is the least solution of the system of constraints $\sqsupseteq$.

  -

- We have a function:

$$F : L^{2m} \to L^{2m}$$

that returns, for each component, the right-hand side of the corresponding constraint:

$$F\begin{pmatrix} In_1 \\ \vdots \\ In_m \\ Out_1 \\ \vdots \\ Out_m \end{pmatrix} = \begin{pmatrix} \vdots \\ \\ \\ \vdots \\ \end{pmatrix}$$

- We know that $\mathsf{X} \in L^{2m}$ is a solution of the system of equations $(=)$ iff F is a fixed point of F:

$$\mathsf{X} = F(\mathsf{X})$$

- In the case of the system of constraints $(\sqsubseteq)$ a vector is a solution iff:

$$\mathsf{X} \sqsupseteq F(\mathsf{X})$$

  that is, F is **reductive** in X.

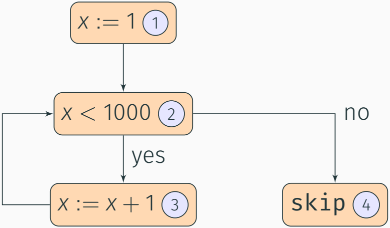
- Assume that we have a concrete domain L and an abstract domain $L^\sharp$.
- We can build a simple "collecting" data-flow analysis, as we did with our collecting semantics.
- From this collecting analysis, our abstraction function α and a concretization function γ we can derive an analysis in the abstract domain.

- Assume we want to compute a collecting state at each program point.
- Our lattice L is $\textbf{State}^* = \textsf{Var} \to \mathcal{P}(\mathbb{Z})$.
- The extreme value ι is σ_0, which contains the information of each variable at the beginning.
- For each block n, f_n is defined as follows:
  - If the n-th block contains $x := e$, then:

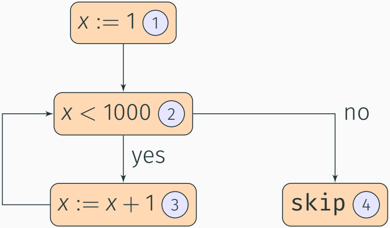
$$f_n(\sigma) = \sigma[x \mapsto [\![e]\!]^* \ \sigma]$$

  - Otherwise, $f_n(\sigma) = \sigma$.

Example



$$
\begin{aligned}
In_1 &\sqsupseteq \sigma_0 & Out_1 &\sqsupseteq In_1\,[x \mapsto \{1\}] \\
In_2 &\sqsupseteq Out_1 \sqcup Out_3 & Out_2 &\sqsupseteq In_2 \\
In_3 &\sqsupseteq Out_2 & Out_3 &\sqsupseteq In_3\,[x \mapsto \{y+1 \mid y \in In_3(x)\}] \\
In_4 &\sqsupseteq Out_2 & Out_4 &\sqsupseteq In_4
\end{aligned}
$$



$$
\begin{aligned}
In_1 &\sqsupseteq \sigma_0 \\
In_2 &\sqsupseteq \sigma_0\,[x \mapsto \{1\}] \sqcup In_3\,[x \mapsto \{y+1 \mid y \in In_3(x)\}] \\
In_3 &\sqsupseteq \sigma_0\,[x \mapsto \{1\}] \sqcup In_3\,[x \mapsto \{y+1 \mid y \in In_3(x)\}] \\
In_4 &\sqsupseteq \sigma_0\,[x \mapsto \{1\}] \sqcup In_3\,[x \mapsto \{y+1 \mid y \in In_3(x)\}]
\end{aligned}
$$

Example

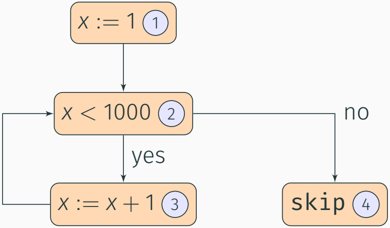
- Let us iterate assuming that $\sigma_0 = \lambda y.\mathbb{Z}$:

$$\underbrace{\begin{pmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{pmatrix}}_{\perp} \rightarrow \underbrace{\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \{1\}] \\ \sigma_0[x \mapsto \{1\}] \\ \sigma_0[x \mapsto \{1\}] \end{pmatrix}}_{F(\perp)} \rightarrow \underbrace{\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \{1,2\}] \\ \sigma_0[x \mapsto \{1,2\}] \\ \sigma_0[x \mapsto \{1,2\}] \end{pmatrix}}_{F^2(\perp)} \rightarrow \underbrace{\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \{1,2,3\}] \\ \sigma_0[x \mapsto \{1,2,3\}] \\ \sigma_0[x \mapsto \{1,2,3\}] \end{pmatrix}}_{F^3(\perp)} \rightarrow \cdots$$

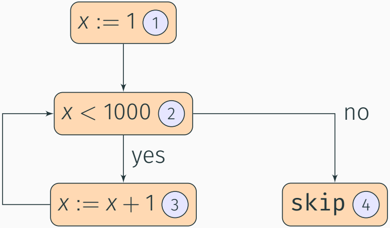
- It does not stabilize, but it converges to:

$$\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \mathbb{Z}^+] \\ \sigma_0[x \mapsto \mathbb{Z}^+] \\ \sigma_0[x \mapsto \mathbb{Z}^+] \end{pmatrix}$$

There is room for improvement!

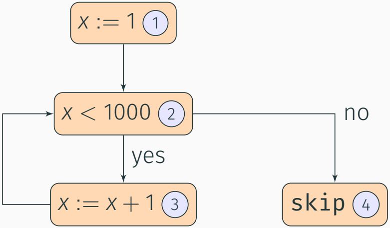


- Notice that we generate the constraint $In_3 \sqsupseteq Out_2$.
- That is, In_3 gathers all the outgoing states from block 2.
- But not all the states coming from block 2 lead to block 3.
- Only the states from Out_2 for which the condition $x < 1000$ holds, lead to block 3.



- We know that the state $[x \mapsto \{y \in \mathbb{Z} \mid y < 1000\}]$ covers all cases in which the condition of block 2 is satisfied.
  - Note: $[x \mapsto \{y \in \mathbb{Z} \mid y < 1000\}]$ denotes the state σ in which $\sigma(x) = \{y \in \mathbb{Z} \mid y < 1000\}$ and $\sigma(z) = \mathbb{Z}$ for any other $z \neq x$.
- Therefore, we can generate:

$$In_3 \sqsupseteq Out_2 \sqcap [x \mapsto \{y \in \mathbb{Z} \mid y < 1000\}]$$



- Analogously, we can have

$$In_4 \sqsupseteq Out_2 \sqcap [x \mapsto \{y \in \mathbb{Z} \mid y \geq 1000\}]$$

Adding path sensitivity

- If we iterate under this new system of constraints we get:

$$\underbrace{\begin{pmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{pmatrix}}_{\perp} \rightarrow \underbrace{\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \{1\}] \\ \sigma_0[x \mapsto \{1\}] \\ \sigma_0[x \mapsto \emptyset] \end{pmatrix}}_{F(\perp)} \rightarrow \underbrace{\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \{1,2\}] \\ \sigma_0[x \mapsto \{1,2\}] \\ \sigma_0[x \mapsto \emptyset] \end{pmatrix}}_{F^2(\perp)} \rightarrow \underbrace{\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \{1,2,3\}] \\ \sigma_0[x \mapsto \{1,2,3\}] \\ \sigma_0[x \mapsto \emptyset] \end{pmatrix}}_{F^3(\perp)} \rightarrow \cdots$$

- But now it does stabilize after 1000 iterations:

$$\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \{1,2,\ldots,1000\}] \\ \sigma_0[x \mapsto \{1,2,\ldots,999\}] \\ \sigma_0[x \mapsto \{1000\}] \end{pmatrix}$$

- Given a concrete data-flow analysis consisting of:
  - A lattice L with the concrete values.
  - An extreme value ι.
  - A set of transfer functions $f_i : L \to L$, one for each block.
- We can build an abstract analysis with:
  - A lattice $L^\sharp$ with the abstract values.
  - An extreme value $\iota^\sharp = \alpha(\iota)$.
  - A set of transfer functions $f_i^\sharp : L^\sharp \to L^\sharp$ such that $f_i^\sharp \sqsupseteq \alpha \circ f_i \circ \gamma$.

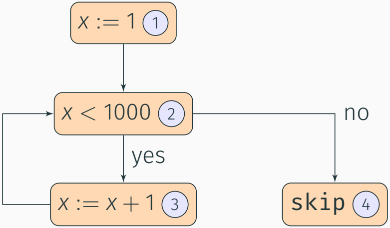
- Our collecting data-flow analysis consists of:
  - A lattice $\mathbf{State}^* = \mathbf{Var} \to \mathcal{P}(\mathbb{Z})$ with the concrete values.
  - An extreme value σ_0.
  - A set of transfer functions:

$$f_i(\sigma) = \sigma[x \mapsto [\![e]\!]^* \ \sigma] \quad \text{if the } i\text{-th block is } x := e$$

    and $f_i(\sigma) = \sigma$ otherwise.
- Our sign analysis consists of:
  - A lattice $\mathbf{State}^\sharp = \mathbf{Var} \to \mathcal{P}(\mathbf{Sign})$ with the abstract values.
  - An extreme value $\sigma_0^\sharp = \alpha(\sigma_0)$.
  - A set of transfer functions:

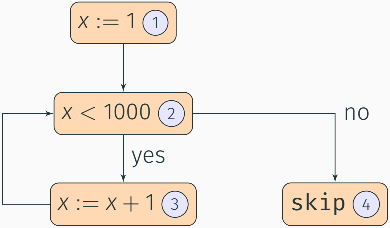
$$f_i^\sharp(\sigma^\sharp) = \sigma^\sharp[x \mapsto [\![e]\!]^\sharp \ \sigma^\sharp] \quad \text{if the } i\text{-th block is } x := e$$

    and $f_i^\sharp(\sigma^\sharp) = \sigma^\sharp$ otherwise.



$$
\begin{aligned}
In_1^\sharp &\sqsupseteq \sigma_0^\sharp & \qquad Out_1^\sharp &\sqsupseteq In_1^\sharp[x \mapsto \{+\}] \\
In_2^\sharp &\sqsupseteq Out_1^\sharp \sqcup Out_3^\sharp & \qquad Out_2^\sharp &\sqsupseteq In_2^\sharp \\
In_3^\sharp &\sqsupseteq Out_2^\sharp & \qquad Out_3^\sharp &\sqsupseteq In_3^\sharp[x \mapsto In_3^\sharp(x) \oplus \{+\}] \\
In_4^\sharp &\sqsupseteq Out_2^\sharp & \qquad Out_4^\sharp &\sqsupseteq In_4^\sharp
\end{aligned}
$$

Example



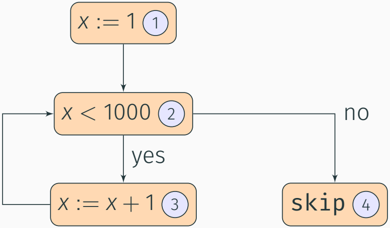
$$
\begin{aligned}
In_1^\sharp &\sqsupseteq \sigma_0^\sharp \\
In_2^\sharp &\sqsupseteq \sigma_0^\sharp[x \mapsto \{+\}] \sqcup In_3^\sharp[x \mapsto In_3(x)^\sharp \oplus \{+\}] \\
In_3^\sharp &\sqsupseteq \sigma_0^\sharp[x \mapsto \{+\}] \sqcup In_3^\sharp[x \mapsto In_3(x)^\sharp \oplus \{+\}] \\
In_4^\sharp &\sqsupseteq \sigma_0^\sharp[x \mapsto \{+\}] \sqcup In_3^\sharp[x \mapsto In_3(x)^\sharp \oplus \{+\}]
\end{aligned}
$$

- The abstract domain satisfies the ascending chain condition:
- Let us iterate:

$$\underbrace{\begin{pmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{pmatrix}}_{\perp} \rightarrow \underbrace{\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \{+\}] \\ \sigma_0[x \mapsto \{+\}] \\ \sigma_0[x \mapsto \{+\}] \end{pmatrix}}_{F(\perp)} \rightarrow \underbrace{\begin{pmatrix} \sigma_0 \\ \sigma_0[x \mapsto \{+\}] \\ \sigma_0[x \mapsto \{+\}] \\ \sigma_0[x \mapsto \{+\}] \end{pmatrix}}_{F^2(\perp)}$$

- The chain stabilizes at the second iteration.

Example



$$
\begin{aligned}
In_1^\sharp &= \sigma_0^\sharp & \quad Out_1^\sharp &= \sigma_0^\sharp[x \mapsto \{+\}] \\
In_2^\sharp &= \sigma_0^\sharp[x \mapsto \{+\}] & \quad Out_2^\sharp &= \sigma_0^\sharp[x \mapsto \{+\}] \\
In_3^\sharp &= \sigma_0^\sharp[x \mapsto \{+\}] & \quad Out_3^\sharp &= \sigma_0^\sharp[x \mapsto \{+\}] \\
In_4^\sharp &= \sigma_0^\sharp[x \mapsto \{+\}] & \quad Out_4^\sharp &= \sigma_0^\sharp[x \mapsto \{+\}]
\end{aligned}
$$

Galois connections

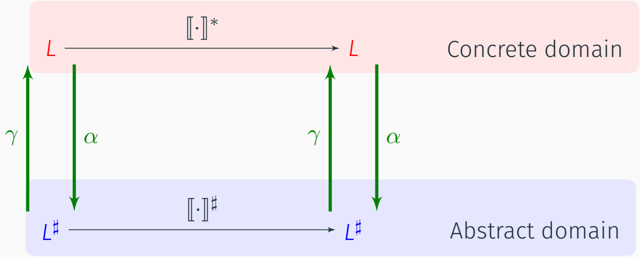
4. Galois connections

Definition

Correctness of the derived abstract interpretation

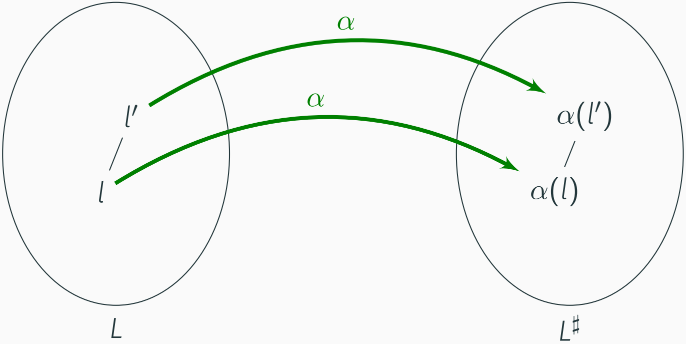
Systematic combination of analyses

Galois connections



- Abstract interpretation is based on the idea of connecting the concrete and abstract domains by means of an abstraction function $\alpha : L \to L^\sharp$ and a concretization function $\gamma : L^\sharp \to L$.
- Not every pair of functions α, γ leads to a sound analysis. Abstraction and concretization functions must satisfy certain conditions.

- Given:
  - A concrete domain L, which is a lattice.
  - An abstract domain $L^\sharp$, which is also a lattice.
  - An abstraction function $\alpha : L \to L^\sharp$.
  - An concretization function $\gamma : L^\sharp \to L$.
- We say that $(L, \alpha, \gamma, L^\sharp)$ form a **Galois connection** between L and $L^\sharp$ if the following conditions hold:
  - α and γ are monotonically increasing.
  - $\gamma \circ \alpha \sqsupseteq id$
  - $\alpha \circ \gamma \sqsubseteq id$



- If l is more accurate than l' in the concrete domain (that is, $l \sqsubseteq l'$), their abstract counterparts must preserve the same relationship: $\alpha(l) \sqsubseteq \alpha'(l)$.

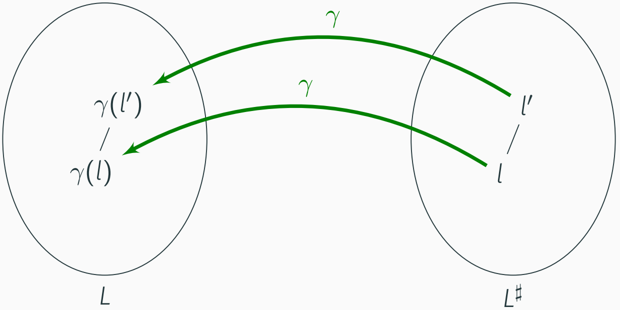
- For example, when $L = \mathcal{P}(\mathbb{Z})$ and $L^\sharp = \mathcal{P}(Sign)$ we have

$$\{2, 4\} \subseteq \{-3, 1, 2, 3, 4\}$$

and abstracting both sides:

$$\{+\} \subseteq \{-, +\}$$

α AND γ MUST BE MONOTONICALLY INCREASING



- Similarly with the concretization function. If:

$$\{+\} \subseteq \{-, +\}$$

- Applying γ to both sides:

$$\mathbb{Z}^+ \subseteq \mathbb{Z}^- \cup \mathbb{Z}^+$$

$\gamma \circ \alpha \sqsupseteq id$

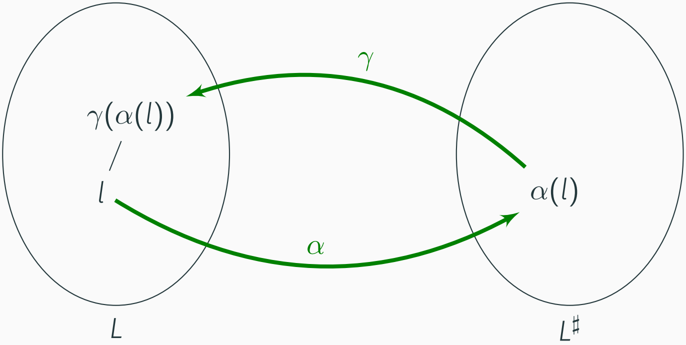
- The condition $\gamma \circ \alpha \sqsupseteq id$ can be rewritten as follows:

$$\forall x \in L. \quad (\gamma \circ \alpha)(x) \sqsupseteq id(x)$$

- or, equivalently:

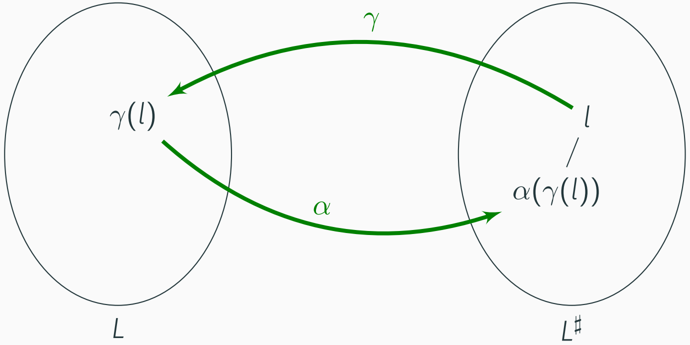
$$\forall x \in L. \quad \gamma(\alpha(x)) \sqsupseteq x$$

$\gamma \circ \alpha \sqsupseteq id$



- If we start from $l \in L$, we abstract it, and make the result concrete again, we obtain another element in L greater or equal than the original one.
- For example: $\{-3, 1, 2, 3, 4\} \xrightarrow{\alpha} \{-, +\} \xrightarrow{\gamma} \mathbb{Z}^- \cup \mathbb{Z}^+$.

$\alpha \circ \gamma \sqsubseteq id$



- If we start from an abstract element $l \in L^\sharp$, make it concrete, and abstract it again, we get another abstract element lower or equal than the original one.
- For example: $\{-\} \xrightarrow{\gamma} \mathbb{Z}^- \xrightarrow{\alpha} \{-\}$.

- In many cases, it holds that $\alpha \circ \gamma = id$ instead of the weaker condition $\alpha \circ \gamma \sqsubseteq id$.
- If this happens, we say that $(L, \alpha, \gamma, L^\sharp)$ is a **Galois insertion**.
- An equivalent characterization of Galois insertions is the following:

Theorem
A Galois connection $(L, \alpha, \gamma, L^\sharp)$ is a Galois insertion iff γ is injective. That is, if there are no pair of abstract values $l, l' \in L^\sharp$ representing the same concrete element.

This means that, in a Galois insertion, the abstract domain $L^\sharp$ does not contain **superfluous elements**.

Theorem (Informal statement)
If:

- *The collecting semantics* $[\![\cdot]\!]^* : L \to L$ *reflects the behaviour of a programming language.*
- $(L, \alpha, \gamma, L^\sharp)$ *is a Galois connection.*

Then:

- *Any analysis (or abstract interpreter)* $[\![\cdot]\!]^\sharp : L^\sharp \to L^\sharp$ *such that* $[\![\cdot]\!]^\sharp \sqsupseteq \alpha \circ [\![\cdot]\!]^* \circ \gamma$ *soundly approximates the behaviour of that language.*

- What does it mean that $[\![\cdot]\!]^*$ reflects the behaviour of the programming language?
- It depends on the language, and the concrete domain.
- For example, in our language we had:

$$\begin{aligned} [\![S]\!] \quad &: \quad \mathbf{State} \rightarrow \mathbf{State} \\ [\![S]\!]^* \quad &: \quad \mathbf{State}^* \rightarrow \mathbf{State}^* \end{aligned}$$

  where $\mathbf{State} = \mathbf{Var} \rightarrow \mathbb{Z}$ and $\mathbf{State}^* = \mathbf{Var} \rightarrow \mathcal{P}(\mathbb{Z})$.
- Our collecting semantics is correct because for every $S \in \mathbf{Stm}, \sigma \in \mathbf{State}, \sigma^* \in \mathbf{State}^*$ such that $\sigma \preceq \sigma^*$ we get $[\![S]\!]\ \sigma \preceq [\![S]\!]^*\ \sigma^*$.

Assuming that the sign analysis on expressions $[\![e]\!]^\sharp$ is correct, prove that the data flow-based sign analysis is correct.

- Galois connections allow us to systematically combine several analyses in order to get another one.
- Analyses can be combined in several ways:

1. Parallel composition
2. Extension to functions
3. Sequential composition

- We have developed a sign analysis for arithmetic expressions.
  - Concrete domain: $\mathcal{P}(\mathbb{Z})$. Abstract domain: $\mathcal{P}(\mathsf{Sign})$.
- Assume we have also developed an analysis for determining parity:
  - Concrete domain: $\mathcal{P}(\mathbb{Z})$. Abstract domain: $\mathcal{P}(\mathsf{Parity})$, where $\mathsf{Parity} = \{odd, even\}$.
- How can we combine both analysis?
- There are two ways:
  1. Direct product: Simple, less accurate.
  2. Direct tensor product: More accurate, but limited to some domains, namely those of the form $\mathcal{P}(X)$ for some set X.

- With direct product, our combined abstract domain is $\mathcal{P}(\mathsf{Sign}) \times \mathcal{P}(\mathsf{Parity})$.
- That is, the analysis yields pairs such as $(\{+\}, \{odd\})$, $(\{0, -\}, \{odd, even\})$, etc.
- For example, we have:

$$
\begin{array}{rcl}
\alpha_{\mathsf{Sign}}(\{0, 3, -5, 1, 9\}) & = & \{0, -, +\} \\
\alpha_{\mathsf{Parity}}(\{0, 3, -5, 1, 9\}) & = & \{odd, even\}
\end{array}
$$

- In the resulting analysis:

$$\alpha(\{0, 3, -5, 1, 9\}) = (\{0, -, +\}, \{odd, even\})$$

- In general, if we have two Galois connections over the same concrete domain:

$$(L, \alpha_1, \gamma_1, L_1^\sharp) \qquad (L, \alpha_2, \gamma_2, L_2^\sharp)$$

- Then $(L, \alpha, \gamma, L_1^\sharp \times L_2^\sharp)$ is a Galois connection, with α and γ defined as follows:

$$\begin{array}{rcl} \alpha(l) & = & (\alpha_1(l), \alpha_2(l)) \\ \gamma(l^\sharp) & = & (\gamma_1(l^\sharp), \gamma_2(l^\sharp)) \end{array}$$

Parallel composition: direct tensor product

- Tensor product can only be applied when the concrete and abstract domains are sets of sets.
- With tensor product, our combined abstract domain is $\mathcal{P}(\textsf{Sign} \times \textsf{Parity})$.
- That is, the analysis yields sets of pairs such as $\{(+, \mathit{odd}), (-, \mathit{even})\}$, $\{(0, \mathit{even})\}$.
- For example, given:

$$\begin{aligned} \alpha_{\textsf{Sign}}(\{0, 3, -5, 1, 9\}) &= \{0, -, +\} \\ \alpha_{\textsf{Parity}}(\{0, 3, -5, 1, 9\}) &= \{\mathit{odd}, \mathit{even}\} \end{aligned}$$

- In the resulting analysis:

$$\alpha(\{0, 3, -5, 1, 9\}) = \{(0, \mathit{even}), (+, \mathit{odd}), (-, \mathit{odd})\}$$

Which result is more accurate?

$$\alpha(\{0, 3, -5, 1, 9\}) = (\{0, -, +\}, \{\mathit{odd}, \mathit{even}\})$$

$$\alpha(\{0, 3, -5, 1, 9\}) = \{(0, \mathit{even}), (+, \mathit{odd}), (-, \mathit{odd})\}$$

- In general, if we have two Galois connections over the same concrete domain:

$$(\mathcal{P}(X), \alpha_1, \gamma_1, \mathcal{P}(X_1)) \qquad (\mathcal{P}(X), \alpha_2, \gamma_2, \mathcal{P}(X_2))$$

- Then $(\mathcal{P}(X), \alpha, \gamma, \mathcal{P}(X_1 \times X_2))$ is a Galois connection, with α and γ defined as follows:

$$\begin{array}{rcl} \alpha(X) & = & \bigcup\{\alpha_1(\{x\}) \times \alpha_2(\{x\}) \mid x \in X\} \\ \gamma(D) & = & \{x \mid \alpha_1(\{x\}) \times \alpha_2(\{x\}) \subseteq D\} \end{array}$$

- Assume we have a sign analysis on expressions with a single variable:

$$[\![e]\!]^\sharp : \mathcal{P}(\mathsf{Sign}) \to \mathcal{P}(\mathsf{Sign})$$

  This analysis receives the signs of the variable, and returns the signs of the expression.
- Now we want to extend our analysis to several variables:

$$[\![e]\!]^\sharp : \underbrace{(\mathsf{Var} \to \mathcal{P}(\mathsf{Sign}))}_{\mathsf{State}^\sharp} \to \mathcal{P}(\mathsf{Sign})$$

  We can do this in a systematic way.

- Given a Galois connection $(L_1, \alpha_1, \gamma_1, L_1^\sharp)$ and a set S, we can build another Galois connection:

$$(S \to L_1, \alpha, \gamma, S \to L_1^\sharp)$$

in which α and γ are defined as follows:

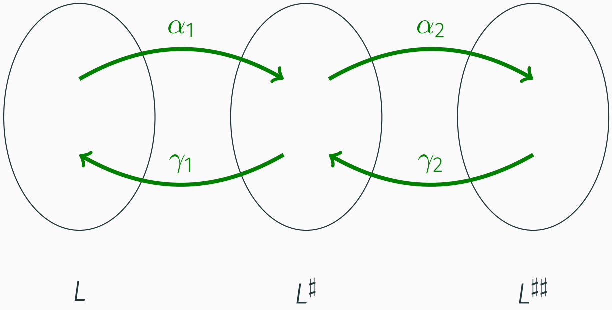
$$\begin{array}{rcl} \alpha(f) & = & \lambda s.\alpha_1(f(s)) \\ \gamma(f^\sharp) & = & \lambda s.\gamma_1(f^\sharp(s)) \end{array} \quad \text{or, equivalently:} \quad \begin{array}{rcl} \alpha(f) & = & \alpha_1 \circ f \\ \gamma(f^\sharp) & = & \gamma_1 \circ f^\sharp \end{array}$$

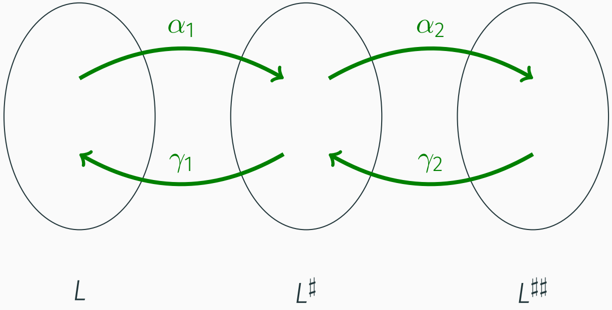
Example
Given $(\mathcal{P}(\mathbb{Z}), \alpha_1, \gamma_1, \mathcal{P}(\mathsf{Sign}))$, we can define $(\mathsf{State}^*, \alpha, \gamma, \mathsf{State}^\sharp)$ such that, for example:

$$\alpha([x \mapsto \{1, -4\}, y \mapsto \{0, 4\}]) = [x \mapsto \{+, -\}, y \mapsto \{0, +\}]$$

$$\gamma([x \mapsto \{-\}, y \mapsto \{0\}]) = [x \mapsto \mathbb{Z}^-, y \mapsto \{0\}]$$

- Sometimes it is useful to design analysis step by step, in successive layers of abstraction:
    - Concrete domain L: close to language semantics.
    - Abstract domain $L^\sharp$: simpler than language semantics.
    - More abstract domain $L^{\sharp\sharp}$: simpler than $L^\sharp$.
    - etc.





- Given two Galois connections:

$$(L, \alpha_1, \gamma_1, L^\sharp) \qquad (L^\sharp, \alpha_2, \gamma_2, L^{\sharp\sharp})$$

- The following is also a Galois connection:

$$(L, \alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2, L^{\sharp\sharp})$$

$$
\begin{tikzcd}
L \arrow[r, "{[\![\cdot]\!]^*}"] & L \arrow[d, "\alpha_1"] & \text{Concrete domain} \\
L^\sharp \arrow[u, "\gamma_1"] & L^\sharp \arrow[d, "\alpha_2"] & \text{Abstract domain} \\
L^{\sharp\sharp} \arrow[u, "\gamma_2"] \arrow[r, "???"] & L^{\sharp\sharp} & \text{Simpler abstract domain}
\end{tikzcd}
$$

$$
\begin{tikzcd}
L & & L & \text{Concrete domain} \\
L^\sharp \arrow[rr, "{[\![\cdot]\!]^\sharp}"] & & L^\sharp \arrow[d, "\alpha_2"] & \text{Abstract domain} \\
L^{\sharp\sharp} \arrow[u, "\gamma_2"] \arrow[rr, "???"] & & L^{\sharp\sharp} & \text{Simpler abstract domain}
\end{tikzcd}
$$

Forcing stabilization of ascending chains

5. Forcing stabilization of ascending chains

- Case study: interval analysis
- Tarski's fixed point theorem
- Widening operators
- Narrowing operators

- We want to determine the range of values that a variable may take as a closed interval $[a, b]$.
- **Application:** check that array accesses are within bounds.
- Example:

```
x := 1
{ x ∈ [1,1] }
while x < 1000 do
    { x ∈ [1,999] }
    x := x + 1
    { x ∈ [2,1000] }
{ x ∈ [1000,1000] }
```

- Our analysis will be data-flow based, combined with abstract interpretation.
- Concrete domain:

$$\textsf{State}^* = \textsf{Var} \to \mathcal{P}(\mathbb{Z})$$

- Abstract domain:

$$\textsf{State}^\sharp = \textsf{Var} \to \textsf{Interval}$$

What is **Interval**?

Interval LATTICE

- The **Interval** lattice is defined as follows:

$$\textbf{Interval} = \{\bot\} \cup \{[a, b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \in \{+\infty\}, a \leq b\}$$

  where $\bot$ represents an empty interval.

- Let us define an order relation $\sqsubseteq$ on **Interval**:

$$\begin{aligned} \bot \sqsubseteq \mathit{int} \qquad & \text{for every } \mathit{int} \in \textbf{Interval} \\ [a_1, b_1] \sqsubseteq [a_2, b_2] \quad \iff & \quad a_2 \leq a_1 \text{ y } b_1 \leq b_2 \end{aligned}$$

  that is, $[a_1, b_1] \sqsubseteq [a_2, b_2]$ iff the set of numbers defined by $[a_1, b_1]$ is contained within the set defined by $[a_2, b_2]$.

Interval LATTICE

- If int_1 e int_2 are two intervals, their least upper bound is the smallest interval containing both.

Interval lattice

- If int_1 e int_2 are two intervals, their **least upper bound** is the smallest interval containing both.

- That is:

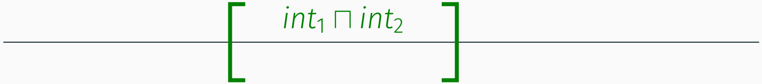
$$\bot \sqcup int = int \sqcup \bot = int \qquad \text{for every } int \in \textbf{Interval}$$

$$[a_1, b_1] \sqcup [a_2, b_2] = [min(a_1, b_1), max(a_2, b_2)]$$

- If int_1 e int_2 are two intervales, their greatest lower bound $int_1 \sqcap int_2$ is their intersection.

Interval LATTICE

- If int_1 e int_2 are two intervales, their **greatest lower bound** $int_1 \sqcap int_2$ is their intersection.



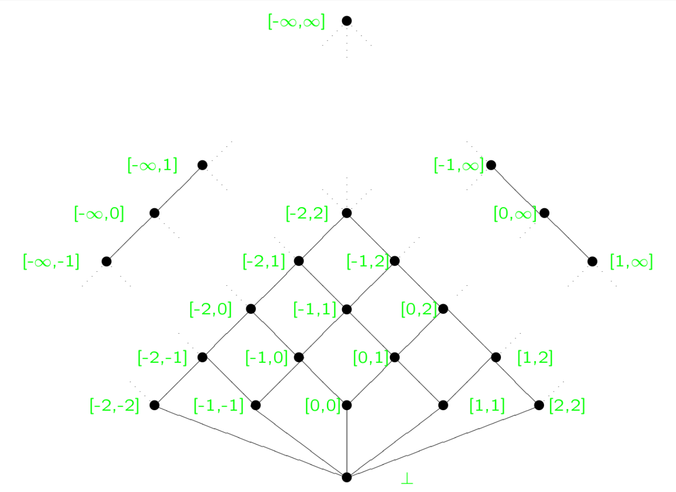
- That is:

$$\bot \sqcap int = int \sqcap \bot = \bot \qquad \text{for every } int \in \textbf{Interval}$$

$$[a_1, b_1] \sqcap [a_2, b_2] = \begin{cases} \bot & \text{if } b_1 < a_2 \text{ or } b_2 < a_1 \\ [max(a_1, b_1), min(a_2, b_2)] & \text{otherwise} \end{cases}$$

Is Interval a complete lattice? Which are its topmost and bottommost elements?

Interval LATTICE



Source: F. Nielson & H. Riis Nielson.
`http://www.imm.dtu.dk/~hrni/PPA/slides4.pdf`

- Recall that if $(L, \sqsubseteq)$ is a complete lattice, then we can build a complete lattice $S \to L$ for any finite S.
- In particular, $\textsf{State}^\sharp = \textsf{Var} \to \textsf{Interval}$ is a lattice.
- The corresponding order relation $\sqsubseteq$ is defined as follows:

$$\sigma_1^\sharp \sqsubseteq \sigma_2^\sharp \Leftrightarrow \quad \forall x \in \textsf{Var} : \; \sigma_1^\sharp(x) \sqsubseteq_{\textsf{Interval}} \sigma_2^\sharp(x)$$

  and the $\sqcap$ and $\sqcup$ operators are defined accordingly.

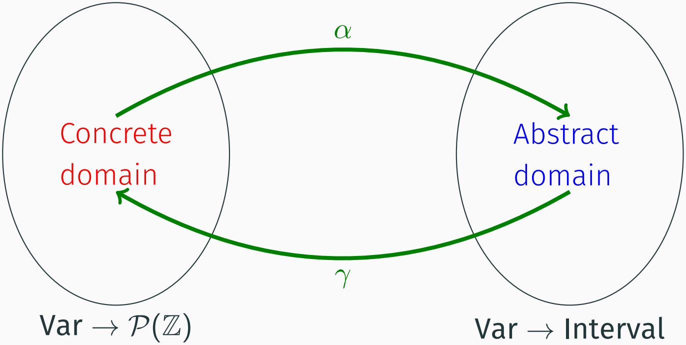
Example

Given:

$$\sigma_1^\sharp = [x \mapsto [1, +\infty], y \mapsto [-3, 4]]$$
$$\sigma_2^\sharp = [x \mapsto [-\infty, -2], y \mapsto [-10, 0]]$$

then:

$$\sigma_1^\sharp \sqcup \sigma_2^\sharp = [x \mapsto [-\infty, +\infty], y \mapsto [-10, 4]]$$
$$\sigma_1^\sharp \sqcap \sigma_2^\sharp = [x \mapsto \bot, y \mapsto [-3, 0]]$$



- Let us bind these domains by a Galois connection.

Which is the least element of $\mathsf{State}^\sharp$ representing this concrete state?

$$\sigma = [x \mapsto \{0, -2, 5\}, y \mapsto \{3, 5, 6\}]$$

We start relating $\mathcal{P}(\mathbb{Z})$ and **Interval**.

- Abstraction function $\alpha' : \mathcal{P}(\mathbb{Z}) \to$ **Interval**:

$$\alpha'(X) = \begin{cases} \bot & \text{if } X = \emptyset \\ [\mathit{inf}\ X, \mathit{sup}\ X] & \text{otherwise} \end{cases}$$

- Concretization function: $\gamma' :$ **Interval** $\to \mathcal{P}(\mathbb{Z})$:

$$\begin{array}{rcl} \gamma'(\bot) & = & \emptyset \\ \gamma'([a, b]) & = & \{x \in \mathbb{Z} \mid a \leq x \leq b\} \end{array}$$

$(\mathcal{P}(\mathbb{Z}), \alpha', \gamma',$ **Interval**$)$ is a Galois Connection

- Since $(\mathcal{P}(\mathbb{Z}), \alpha', \gamma', \mathsf{Interval})$ is a Galois connection, so is $(\mathsf{State}^*, \alpha, \gamma, \mathsf{State}^\sharp)$, with α and γ defined as follows:

$$\alpha(\sigma) = \lambda x.\ \alpha'(\sigma(x))$$

$$\gamma(\sigma^\sharp) = \lambda x.\ \gamma'(\sigma^\sharp(x))$$

- In order to derive an abstract interpretation for arithmetic expressions, we have to define the abstract variants of the $+$, $-$ and $*$ operators:
  - These will be denoted by $\oplus$, $\ominus$ and $\otimes$, respectively.
- Given the collecting versions of these operators:

$$f_+, f_-, f_* : \mathcal{P}(\mathbb{Z}) \rightarrow \mathcal{P}(\mathbb{Z})$$

$$\begin{aligned} f_+(X, Y) &= \{x + y \mid x \in X, y \in Y\} \\ f_-(X, Y) &= \{x - y \mid x \in X, y \in Y\} \\ f_*(X, Y) &= \{x * y \mid x \in X, y \in Y\} \end{aligned}$$

- In order to derive an abstract interpretation for arithmetic expressions, we have to define the abstract variants of the $+$, $-$ and $*$ operators:
  - These will be denoted by $\oplus$, $\ominus$ and $\otimes$, respectively.
- We have to define abstract operations such that:

$$\oplus, \ominus, \oplus : \textsf{Interval} \rightarrow \textsf{Interval}$$

$$\begin{array}{lcl} \oplus & \sqsupseteq & \alpha' \circ f_{+} \circ \gamma' \\ \ominus & \sqsupseteq & \alpha' \circ f_{-} \circ \gamma' \\ \otimes & \sqsupseteq & \alpha' \circ f_{*} \circ \gamma' \end{array}$$

- Which interval does $\bot \oplus [a, b]$ yield?
  1. Make parameters concrete:

$$\gamma(\bot) = \emptyset \quad \gamma([a,b]) = \{z \in \mathbb{Z} \mid a \leq z \leq b\}$$

  2. Apply f_+:

$$\begin{aligned} & f_+(\emptyset, \{z \in \mathbb{Z} \mid a \leq z \leq b\}) \\ = & \{x + y \mid x \in \emptyset, y \in \{z \in \mathbb{Z} \mid a \leq z \leq b\}\} \\ = & \emptyset \end{aligned}$$

  3. Make result abstract: $\alpha(\emptyset) = \bot$.

Abstract addition: $\oplus$

Now two nonempty intervals: $[a_1, b_1] \oplus [a_2, b_2]$

1. Make parameters concrete:

$$\gamma([a_1, b_1]) = \{z \in \mathbb{Z} \mid a_1 \leq z \leq b_1\} \quad \gamma([a_2, b_2]) = \{z \in \mathbb{Z} \mid a_2 \leq z \leq b_2\}$$

2. Apply f_+:

$$\begin{aligned}
& f_+(\{z \in \mathbb{Z} \mid a_1 \leq z \leq b_1\}, \{z \in \mathbb{Z} \mid a_2 \leq z \leq b_2\}) \\
= \;& \{x + y \mid x \in \{z \in \mathbb{Z} \mid a_1 \leq z \leq b_1\}, y \in \{z \in \mathbb{Z} \mid a_2 \leq z \leq b_2\}\} \\
= \;& \{x + y \mid a_1 \leq x \leq b_1, a_2 \leq y \leq b_2\} \\
= \;& \{x + y \mid a_1 \leq x \leq b_1, a_2 \leq y \leq b_2\} \\
\subseteq \;& \{x + y \mid a_1 + a_2 \leq x + y \leq b_1 + b_2\} \\
= \;& \{z \mid a_1 + a_2 \leq z \leq b_1 + b_2\}
\end{aligned}$$

since $a_1 \leq x \leq b_1, a_2 \leq y \leq b_2$ implies $a_1 + a_2 \leq x + y \leq b_1 + b_2$

3. Make result abstract:

$$\alpha(\{z \mid a_1 + a_2 \leq z \leq b_1 + b_2\}) = [a_1 + a_2, b_1 + b_2]$$

- Therefore:

$$
\begin{aligned}
int_1 \oplus int_2 &= \bot && \text{if } int_1 = \bot \text{ or } int_2 = \bot \\
[a_1, b_1] \oplus [a_2, b_2] &= [a_1 + a_2, b_1 + b_2] && \text{otherwise}
\end{aligned}
$$

- Similarly:

$$
\begin{aligned}
int_1 \ominus int_2 &= \bot && \text{if } int_1 = \bot \text{ or } int_2 = \bot \\
[a_1, b_1] \ominus [a_2, b_2] &= [a_1 - b_2, b_1 - a_2] && \text{otherwise}
\end{aligned}
$$

Abstract multiplication: $\otimes$

- Assume we want to derive $[a_1, b_1] \otimes [a_2, b_2]$:
- **Case 1**: $a_1, a_2 \geq 0$.
  1. Make parameters concrete:

     $$\gamma([a_1, b_1]) = \{z \in \mathbb{Z} \mid a_1 \leq z \leq b_1\} \quad \gamma([a_2, b_2]) = \{z \in \mathbb{Z} \mid a_2 \leq z \leq b_2\}$$

  2. Apply f_*:

     $$\begin{aligned} & f_*(\{z \in \mathbb{Z} \mid a_1 \leq z \leq b_1\}, \{z \in \mathbb{Z} \mid a_2 \leq z \leq b_2\}) \\ = & \{x * y \mid a_1 \leq x \leq b_1, a_2 \leq y \leq b_2\} \\ \subseteq & \{x * y \mid a_1 * a_2 \leq x * y \leq b_1 * b_2\} \\ = & \{z \mid a_1 * a_2 \leq z \leq b_1 * b_2\} \end{aligned}$$

     since $0 \leq a_1 \leq x \leq b_1, 0 \leq a_2 \leq y \leq b_2$ implies $a_1 * b_1 \leq x * y \leq a_2 * b_2$.
  3. Make result abstract:

     $$\alpha(\{z \mid a_1 * a_2 \leq z \leq b_1 * b_2\}) = [a_1 * a_2, b_1 * b_2]$$

- **Case 2:** $b_1 \leq 0, a_2 \geq 0$.
    - Same as before, but now:

$$\begin{aligned}
& f_*(\{z \in \mathbb{Z} \mid a_1 \leq z \leq b_1\}, \{z \in \mathbb{Z} \mid a_2 \leq z \leq b_2\}) \\
= & \{x * y \mid a_1 \leq x \leq b_1 \leq 0, 0 \leq a_2 \leq y \leq b_2\} \\
= & \{x * y \mid 0 \leq -b_1 \leq -x \leq -a_1, 0 \leq a_2 \leq y \leq b_2\} \\
= & \{-v * y \mid 0 \leq -b_1 \leq v \leq -a_1, 0 \leq a_2 \leq y \leq b_2\} \\
\subseteq & \{-v * y \mid -b_1 * a_2 \leq v * y \leq -a_1 * b_2\} \\
= & \{-v * y \mid a_1 * b_2 \leq -v * y \leq b_1 * a_2\} \\
= & \{z \mid a_1 * b_2 \leq z \leq b_1 * a_2\}
\end{aligned}$$

    - Abstraction: $[a_1 * b_2, b_1 * a_2]$.

- **Case 3:** $a_1 \geq 0, b_2 \leq 0$. The same as before.
- **Case 4:** $b_1 \leq 0, b_2 \leq 0$:
  - We get:

$$
\begin{aligned}
& f_*(\{z \in \mathbb{Z} \mid a_1 \leq z \leq b_1\}, \{z \in \mathbb{Z} \mid a_2 \leq z \leq b_2\}) \\
= & \{x * y \mid a_1 \leq x \leq b_1 \leq 0, a_2 \leq y \leq b_2 \leq 0\} \\
= & \{x * y \mid 0 \leq -b_1 \leq -x \leq -a_1, 0 \leq -b_2 \leq -y \leq -a_2\} \\
= & \{-v * (-w) \mid 0 \leq -b_1 \leq v \leq -a_1, 0 \leq -b_2 \leq w \leq -a_2\} \\
\subseteq & \{-v * (-w) \mid -b_1 * (-b_2) \leq v * w \leq -a_1 * (-a_2)\} \\
= & \{v * w \mid b_1 * b_2 \leq v * w \leq a_1 * a_2\} \\
= & \{z \mid b_1 * b_2 \leq z \leq a_1 * a_2\}
\end{aligned}
$$

  - Abstraction: $[b_1 * b_2, a_1 * a_2]$.

- **Case 5:** None of the above
    - We can decompose:

$$
\begin{aligned}
&f_*(\{z \in \mathbb{Z} \mid a_1 \le z \le b_1\}, \{z \in \mathbb{Z} \mid a_2 \le z \le b_2\}) \\
=&f_*(\{z \in \mathbb{Z} \mid a_1 \le z \le 0\} \cup \{z \in \mathbb{Z} \mid 0 \le z \le b_1\}, \\
&\quad \{z \in \mathbb{Z} \mid a_2 \le z \le 0\} \cup \{z \in \mathbb{Z} \mid 0 \le z \le b_2\}) \\
=&f_*(\{z \in \mathbb{Z} \mid a_1 \le z \le 0\}, \{z \in \mathbb{Z} \mid a_2 \le z \le 0\}) \\
&\cup f_*(\{z \in \mathbb{Z} \mid a_1 \le z \le 0\}, \{z \in \mathbb{Z} \mid 0 \le z \le b_2\}) \\
&\cup f_*(\{z \in \mathbb{Z} \mid 0 \le z \le b_1\}, \{z \in \mathbb{Z} \mid a_2 \le z \le 0\}) \\
&\cup f_*(\{z \in \mathbb{Z} \mid 0 \le z \le b_1\}, \{z \in \mathbb{Z} \mid 0 \le z \le b_2\})
\end{aligned}
$$

    - Each subset falls into one of the previous cases:
    - Abstraction:
      $[0, a_1 * a_2] \sqcup [a_1 * b_2, 0] \sqcup [a_2 * b_1, 0] \sqcup [0, b_1 * b_2] =$
      $[min(a_1 * b_2, a_2 * b_1), max(a_1 * a_2, b_1 * b_2)]$.

- Reorganizing all cases:

$$\begin{array}{lll} int_1 \otimes int_2 & = \bot & \text{if } int_1 = \bot \text{ or } int_2 = \bot \\ [a_1, b_1] \otimes [a_2, b_2] & = [min(V), max(V)] & \text{otherwise} \end{array}$$

where $V = \{a_1 * b_2, a_1 * b_2, b_1 * a_2, b_1 * b_2\}$.

- Our abstract interpretation is defined as follows:

$$[\![e]\!]^\sharp : \mathsf{State}^\sharp \to \mathsf{Interval}$$

$$\begin{aligned}
[\![n]\!]^\sharp &= \lambda\sigma^\sharp.[n, n] \\
[\![x]\!]^\sharp &= \lambda\sigma^\sharp.\ \sigma^\sharp(x) \\
[\![e_1 + e_2]\!]^\sharp &= \lambda\sigma^\sharp.\ \left([\![e_1]\!]^\sharp\ \sigma^\sharp\right) \oplus \left([\![e_2]\!]^\sharp\ \sigma^\sharp\right) \\
[\![e_1 * e_2]\!]^\sharp &= \lambda\sigma^\sharp.\ \left([\![e_1]\!]^\sharp\ \sigma^\sharp\right) \otimes \left([\![e_2]\!]^\sharp\ \sigma^\sharp\right) \\
[\![e_1 - e_2]\!]^\sharp &= \lambda\sigma^\sharp.\ \left([\![e_1]\!]^\sharp\ \sigma^\sharp\right) \ominus \left([\![e_2]\!]^\sharp\ \sigma^\sharp\right)
\end{aligned}$$

Example

- Let $\sigma^\sharp = [x \mapsto [-2, 3], y \mapsto [4, 9]]$.
- Let us evaluate $2 * x + y$:

$$
\begin{aligned}
[\![2 * x + y]\!]^\sharp\, \sigma^\sharp &= [\![2 * x]\!]^\sharp\, \sigma^\sharp \oplus [\![y]\!]^\sharp\, \sigma^\sharp \\
&= \left([\![2]\!]^\sharp\, \sigma^\sharp \otimes [\![x]\!]^\sharp\, \sigma^\sharp\right) \oplus [\![y]\!]^\sharp\, \sigma^\sharp \\
&= ([2, 2] \otimes [-2, 3]) \oplus [4, 9] \\
&= [-4, 6] \oplus [4, 9] \\
&= [0, 15]
\end{aligned}
$$

Abstract interpretation for boolean expressions

- We need the abstract version of $\leq$ operator, which will be denoted by $\leq^\sharp$: **Interval** $\times$ **Interval** $\to \mathcal{P}(\mathbb{B})$.

$$
\begin{array}{lll}
int_1 \leq^\sharp int_2 & = \emptyset & \text{if } int_1 = \bot \text{ or } int_2 = \bot \\
[a_1, b_1] \leq^\sharp [a_2, b_2] & = \{true\} & \text{if } b_1 \leq a_2 \\
[a_1, b_1] \leq^\sharp [a_2, b_2] & = \{false\} & \text{if } a_1 > b_2 \\
[a_1, b_1] \leq^\sharp [a_2, b_2] & = \mathbb{B} & \text{otherwise}
\end{array}
$$

- The same with the $=$ operator:

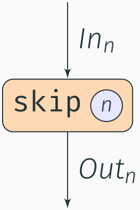
$$
\begin{array}{lll}
int_1 =^\sharp int_2 & = \emptyset & \text{if } int_1 = \bot \text{ or } int_2 = \bot \\
[a_1, b_1] =^\sharp [a_2, b_2] & = \{true\} & \text{si } a_1 = a_2 = b_1 = b_2 \\
[a_1, b_1] =^\sharp [a_2, b_2] & = \{false\} & \text{si } b_1 < a_2 \text{ or } b_2 < a_1 \\
[a_1, b_1] =^\sharp [a_2, b_2] & = \mathbb{B} & \text{otherwise}
\end{array}
$$

$$[\![b]\!]^\sharp \colon \mathsf{State}^\sharp \to \mathcal{P}(\mathbb{B})$$

$$\begin{aligned}
[\![\mathit{true}]\!]^\sharp &= \lambda\sigma^\sharp.\,\{\mathit{true}\}\\
[\![\mathit{false}]\!]^\sharp &= \lambda\sigma^\sharp.\,\{\mathit{false}\}\\
[\![e_1 = e_2]\!]^\sharp &= \lambda\sigma^\sharp.[\![e_1]\!]^\sharp\ \sigma^\sharp =^\sharp [\![e_2]\!]^\sharp\ \sigma^\sharp\\
[\![e_1 \leq e_2]\!]^\sharp &= \lambda\sigma^\sharp.[\![e_1]\!]^\sharp\ \sigma^\sharp \leq^\sharp [\![e_2]\!]^\sharp\ \sigma^\sharp\\
[\![\neg b]\!]^\sharp &= \lambda\sigma^\sharp.\mathit{not}^\sharp([\![b]\!]^\sharp\ \sigma^\sharp)\\
[\![b_1 \wedge b_2]\!]^\sharp &= \lambda\sigma^\sharp.\mathit{and}^\sharp([\![b_1]\!]^\sharp\ \sigma^\sharp, [\![b_2]\!]^\sharp\ \sigma^\sharp)
\end{aligned}$$

- Let us define a monotone framework for intervals:
  - The lattice is our abstract domain: $\mathbf{State}^\sharp = \mathbf{Var} \to \mathbf{Interval}$.
  - If σ_0 is the extreme value of the concrete semantics, then $\iota^\sharp = \alpha(\sigma_0)$.
  - We have to find the abstract transfer functions for each kind of block: `skip`, conditional and $x := e$.

Transfer functions



- Concrete transfer function:

$$f_n(\sigma) = \sigma$$

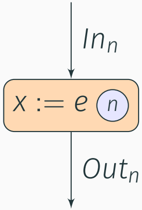
- Abstract transfer function:

$$f_n^\sharp(\sigma^\sharp) = \sigma^\sharp$$

The same applies to conditional blocks. It holds that:

$$f_n^\sharp = \lambda\sigma^\sharp.\sigma^\sharp \sqsupseteq \sigma^\sharp.\alpha(\gamma(\sigma^\sharp)) = \sigma^\sharp.\alpha(f_n(\gamma(\sigma^\sharp))) = \alpha \circ f_n \circ \gamma$$

Transfer functions



- Concrete transfer function:

$$f_n(\sigma) = \sigma[x \mapsto [\![e]\!]^* \, \sigma]$$

- Abstract function:

$$f_n^\sharp(\sigma^\sharp) = \sigma^\sharp \left[x \mapsto [\![e]\!]^\sharp \, \sigma^\sharp\right]$$

- Let us prove that $\alpha \circ f_n \circ \gamma \sqsubseteq f_n^\sharp$.
- Assume we have:

$$\begin{array}{rlcrl} \alpha' : & \mathcal{P}(\mathbb{Z}) \to \textbf{Interval} & & \alpha : & \textbf{State}^* \to \textbf{State}^\sharp \\ \gamma' : & \textbf{Interval} \to \mathcal{P}(\mathbb{Z}) & & \gamma : & \textbf{State}^\sharp \to \textbf{State}^* \end{array}$$

- Assume some $\sigma^\sharp \in \textbf{State}^\sharp$. Let us denote $\gamma(\sigma^\sharp)$ by σ. We get:

$$\begin{array}{rl} & (\alpha \circ f_n \circ \gamma)(\sigma^\sharp) \\ = & \alpha(f_n(\gamma(\sigma^\sharp))) \\ = & \alpha(f_n(\sigma)) \\ = & \alpha(\sigma[x \mapsto [\![e]\!]^*\ \sigma]) \\ = & \alpha\left(\lambda z. \begin{cases} [\![e]\!]^*\ \sigma & \text{if } z = x \\ \sigma(z) & \text{otherwise} \end{cases}\right) \end{array}$$

- By definition of α:

$$\alpha\left(\lambda z.\begin{cases}[\![e]\!]^*\ \sigma & \text{if } z = x\\ \sigma(z) & \text{otherwise}\end{cases}\right) = \lambda z.\begin{cases}\alpha'([\![e]\!]^*\ \sigma) & \text{if } z = x\\ \alpha'(\sigma(z)) & \text{otherwise}\end{cases}$$

- We know that $\sigma = \gamma(\sigma^\sharp)$, so:

$$\lambda z.\begin{cases}\alpha'([\![e]\!]^*\ \sigma) & \text{if } z = x\\ \alpha'(\sigma(z)) & \text{otherwise}\end{cases} = \lambda z.\begin{cases}\alpha'([\![e]\!]^*\gamma(\sigma^\sharp)) & \text{if } z = x\\ \alpha'(\gamma'(\sigma^\sharp(z))) & \text{otherwise}\end{cases}$$

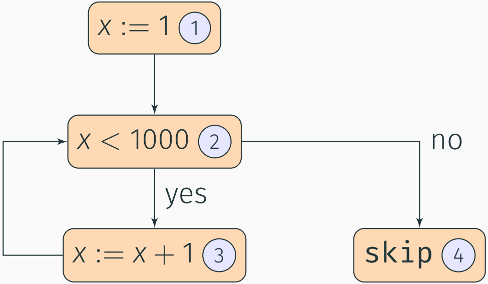
- Since $\alpha' \circ [\![e]\!]^* \circ \gamma \sqsubseteq [\![e]\!]^\sharp$:

$$\lambda z. \begin{cases} \alpha'([\![e]\!]^* \gamma(\sigma^\sharp)) & \text{if } z = x \\ \alpha'(\gamma'(\sigma^\sharp(z))) & \text{otherwise} \end{cases} \sqsubseteq \lambda z. \begin{cases} [\![e]\!]^\sharp\ \sigma^\sharp & \text{if } z = x \\ \alpha'(\gamma'(\sigma^\sharp(z))) & \text{otherwise} \end{cases}$$

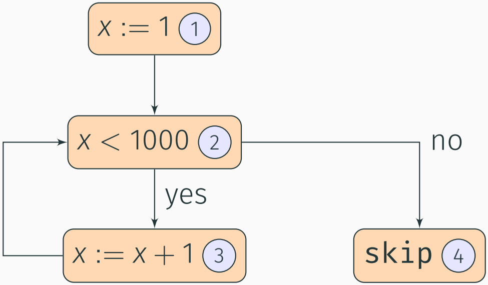
- Since $\alpha' \circ \gamma' \sqsubseteq id$:

$$\lambda z. \begin{cases} [\![e]\!]^\sharp\ \sigma^\sharp & \text{if } z = x \\ \alpha'(\gamma'(\sigma^\sharp(z))) & \text{otherwise} \end{cases} \sqsubseteq \lambda z. \begin{cases} [\![e]\!]^\sharp\ \sigma^\sharp & \text{if } z = x \\ \sigma^\sharp(z) & \text{otherwise} \end{cases}$$

- The latter of which being equal to $\sigma^\sharp[x \mapsto [\![e]\!]^\sharp\ \sigma^\sharp]$, which is $f^\sharp(\sigma^\sharp)$.

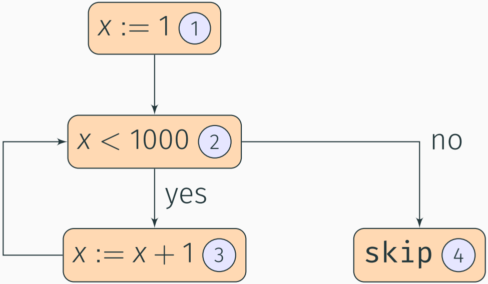


$$
\begin{array}{lll@{\qquad\qquad}lll}
In_1^\sharp & \sqsupseteq & \alpha(\sigma_0) & Out_1^\sharp & \sqsupseteq & In_1^\sharp\,[x \mapsto [1,1]] \\
In_2^\sharp & \sqsupseteq & Out_1^\sharp \sqcup Out_3^\sharp & Out_2^\sharp & \sqsupseteq & In_2^\sharp \\
In_3^\sharp & \sqsupseteq & Out_2^\sharp & Out_3^\sharp & \sqsupseteq & In_3^\sharp \left[x \mapsto In_3^\sharp(x) \oplus [1,1]\right] \\
In_4^\sharp & \sqsupseteq & Out_2^\sharp & Out_4^\sharp & \sqsupseteq & In_4^\sharp
\end{array}
$$



$$
\begin{aligned}
In_1^\sharp &\sqsupseteq \alpha(\sigma_0) \\
In_2^\sharp &\sqsupseteq In_1^\sharp[x \mapsto [1,1]] \sqcup In_3^\sharp\left[x \mapsto In_3^\sharp(x) \oplus [1,1]\right] \\
In_3^\sharp &\sqsupseteq In_2^\sharp \\
In_4^\sharp &\sqsupseteq In_2^\sharp
\end{aligned}
$$

Example



- Adding path sensitivity:

$$
\begin{array}{lcl}
In_1^\sharp & \sqsupseteq & \alpha(\sigma_0) \\
In_2^\sharp & \sqsupseteq & In_1^\sharp\,[x \mapsto [1,1]] \sqcup In_3^\sharp \left[x \mapsto In_3^\sharp(x) \oplus [1,1]\right] \\
In_3^\sharp & \sqsupseteq & In_2^\sharp \;\sqcap\; [x \mapsto [-\infty, 999]] \\
In_4^\sharp & \sqsupseteq & In_2^\sharp \;\sqcap\; [x \mapsto [1000, +\infty]]
\end{array}
$$

Example

- We start iterating from $\lambda x.\bot \in \mathsf{State}^\sharp$.
- Assume that $\sigma_0 = [x \mapsto \{0\}]$.

$$\begin{pmatrix} \lambda x.\bot \\ \lambda x.\bot \\ \lambda x.\bot \\ \lambda x.\bot \end{pmatrix} \to \begin{pmatrix} [x \mapsto [0,0]] \\ [x \mapsto [1,1]] \\ \lambda x.\bot \\ \lambda x.\bot \end{pmatrix} \to \begin{pmatrix} [x \mapsto [0,0]] \\ [x \mapsto [1,1]] \\ [x \mapsto [1,1]] \\ [x \mapsto \bot] \end{pmatrix} \to \begin{pmatrix} [x \mapsto [0,0]] \\ [x \mapsto [1,2]] \\ [x \mapsto [1,1]] \\ [x \mapsto \bot] \end{pmatrix} \to$$

$$\begin{pmatrix} [x \mapsto [0,0]] \\ [x \mapsto [1,2]] \\ [x \mapsto [1,2]] \\ [x \mapsto \bot] \end{pmatrix} \to \begin{pmatrix} [x \mapsto [0,0]] \\ [x \mapsto [1,3]] \\ [x \mapsto [1,2]] \\ [x \mapsto \bot] \end{pmatrix} \to \begin{pmatrix} [x \mapsto [0,0]] \\ [x \mapsto [1,3]] \\ [x \mapsto [1,3]] \\ [x \mapsto \bot] \end{pmatrix} \to \cdots$$

Does $\mathsf{State}^\sharp$ satisfy the ascending chain condition?

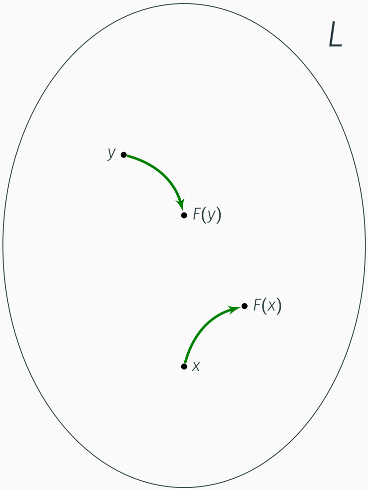
- However, in this case, the chain stabilizes:

$$
\begin{aligned}
In_1^\sharp &= [0,0] \\
In_2^\sharp &= [1,1000] \\
In_3^\sharp &= [1,999] \\
In_4^\sharp &= [1000,1000]
\end{aligned}
$$

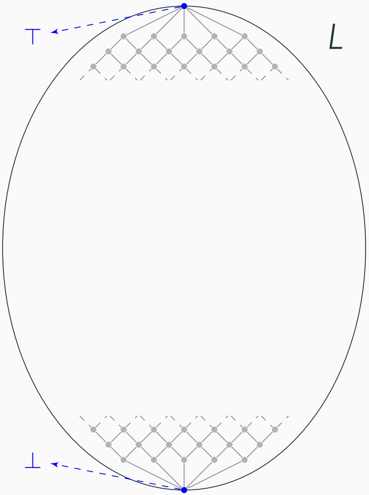
- ...but it does after 2000 iterations!
- In some other cases, the generated chain might not stabilize.

1. Can we force the stabilization of the chain?
2. Even if the chain estabilizes by itself, can we accelerate its convergence?

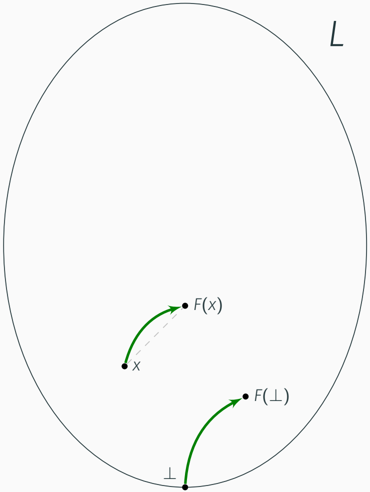
- We assume:
    - A lattice L, which is the property space.
    - A function $F : L \to L$.
- If $x \in L$, we represent the image of x with an arrow to the element $F(x)$ in the lattice.



- Let us assume that L is a complete lattice.
- Therefore there are $\top$ and $\bot$ elements.



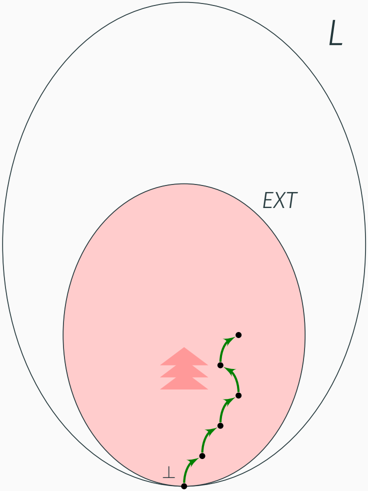
- If $F : L \to L$, F is said to be **extensive** at x if $F(x) \sqsupseteq x$.
- Obviously, F is extensive at $\perp$.



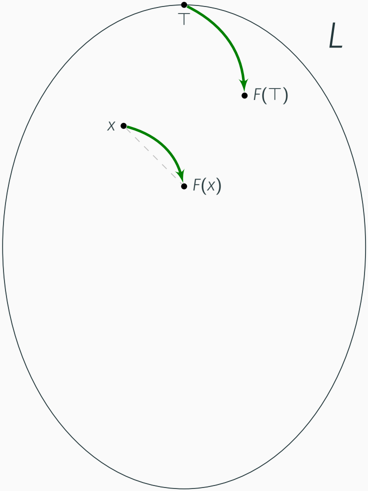
- Let us denote by EXT the set of elements $x \in L$ at which F is extensive:
- Obviously, $\perp \in EXT$.
- If F es monotone, the Kleene ascending chain:

$$\perp \sqsubseteq F(\perp) \sqsubseteq F^2(\perp) \sqsubseteq \cdots$$

does not leave EXT.



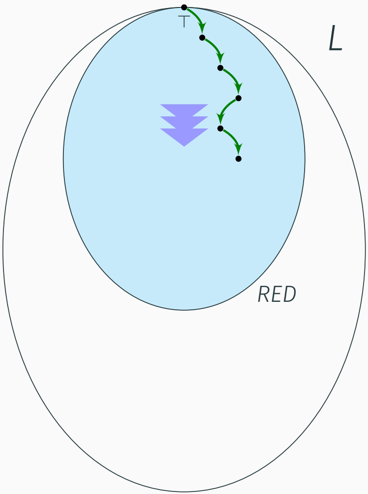
- A function $F : L \to L$ is said to be **reductive** at x if $F(x) \sqsubseteq x$.
- Obviously, F is reductive at $\top$.



- We denote by *RED* the set of elements $x \in L$ at which F is reductive.
- Obviously, $\top \in RED$.
- If F is monotone, Kleene's descending chain:

$$\top \sqsupseteq F(\top) \sqsupseteq F^2(\top) \sqsupseteq \cdots$$

does not leave *RED*.



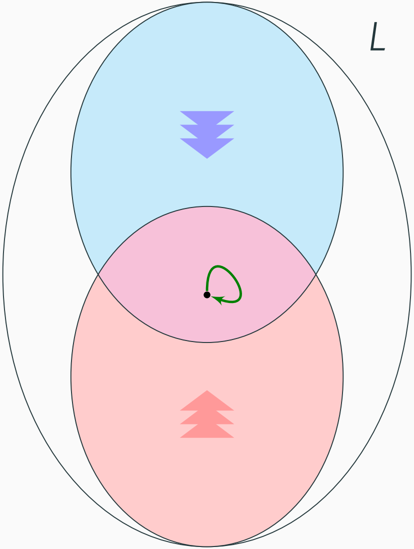
- *EXT* and *RED* can overlap.
- If $x \in EXT \cap RED$ this means that:

$$x \sqsubseteq F(x) \quad x \sqsupseteq F(x)$$

that is,

$$x = F(x)$$

- Therefore, x is a **fixed point** of F.



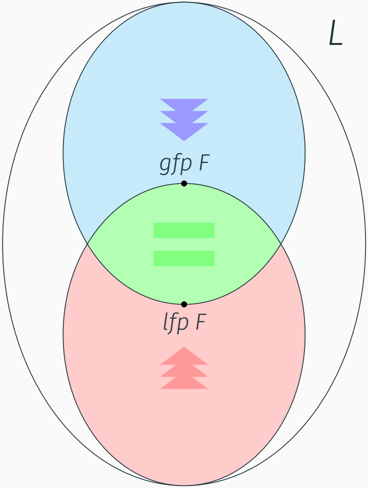
- Let us denote by *FIX* the set of fixed points of *F*.
- That is:

$$FIX = EXT \cap RED$$

- In *FIX* there is a least fixed point (*lfp F*) and a greatest fixed point (*gfp F*).

$$lfp\ F = \bigsqcap FIX$$

$$gfp\ F = \bigsqcup FIX$$

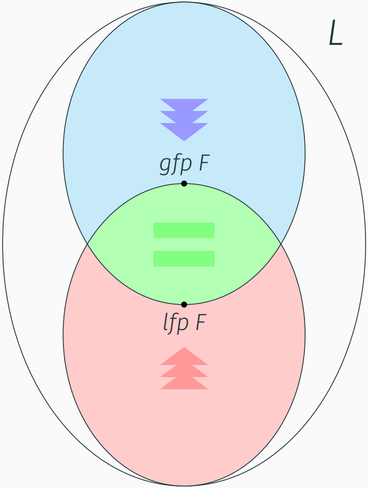


Theorem (Tarski)

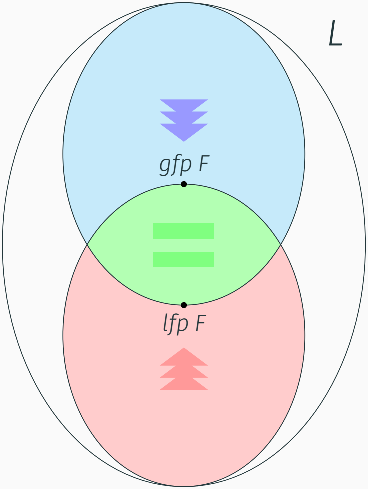
If $F : L \to L$ is monotonically increasing and L is a complete lattice:

$$lfp\ F = \bigsqcap RED$$

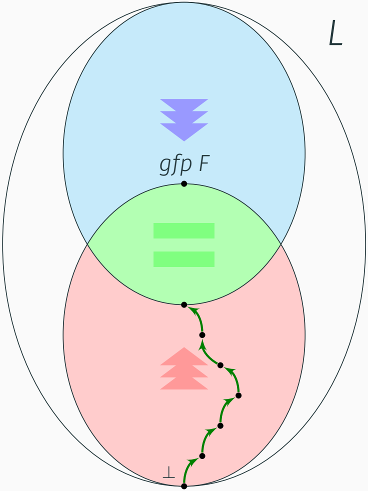
$$gfp\ F = \bigsqcup EXT$$



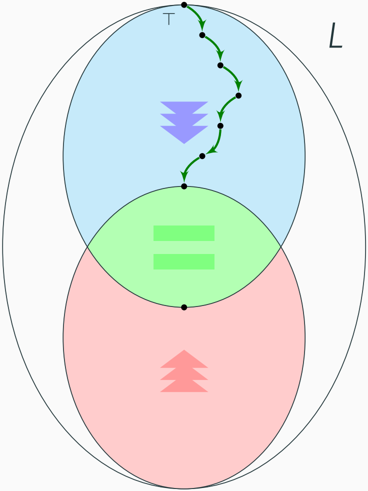
- If F is the function associated to the data-flow constraints, we want to compute *lfp F*.
- All solutions of the system of constraints are in the blue and green zones (*RED*), but the most accurate one is *lfp F*.



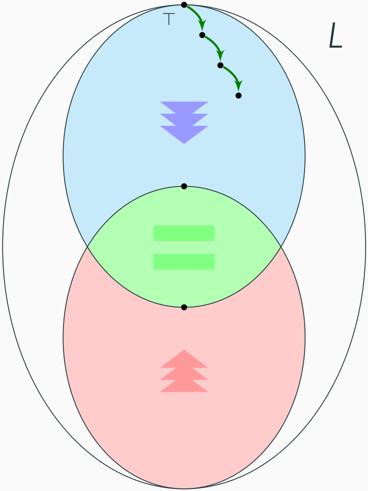
- If Kleene's ascending chain stabilizes, it does at the least fixed point.
- Any intermediate step before stabilization (red zone) is not a solution of the system of constraints.

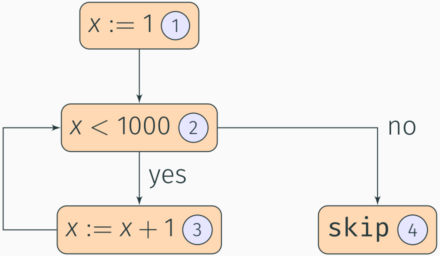


- Alternatively, we can start iterating F from $\top$ in order to get a Kleene's descending chain.
- If the chain stabilizes, it does at the greatest fixed point.



- In this case we could stop at any intermediate point in the chain.
- Any element in this chain is a solution of the system of constraints.
- However, this technique usually leads to inaccurate results.





- If we had applied Kleene's descending chain in our interval analysis, starting from $\top$, the chain would have stabilized in the following solution:

$$
\begin{array}{lcl}
In_1^\sharp & = & [x \mapsto [0, 0]] \\
In_2^\sharp & = & [x \mapsto [-\infty, 1000]] \\
In_3^\sharp & = & [x \mapsto [-\infty, 999]] \\
In_4^\sharp & = & [x \mapsto [1000, +\infty]]
\end{array}
$$

- We know that Kleene's ascending chain

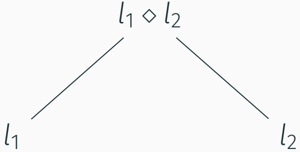
$$\bot \sqsubseteq F(\bot) \sqsubseteq F^2(\bot) \sqsubseteq F^3(\bot) \sqsubseteq \cdots$$

  might not stabilize.
- A widening operator transforms this ascending chain into another chain such that the latter stabilizes.
- The value in which the transformed chain stabilizes is an upper approximation of $\mathit{lfp}\ F$.

- Let $\diamond$ a binary operator in a complete lattice:

$$\diamond : L \times L \to L$$

- We say that $\diamond$ is an **upper bound** operator if it yields a result that is always greater than its operands.



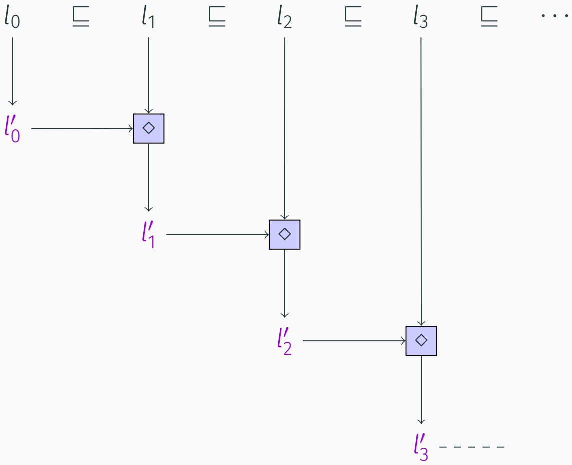
- That is, for every $l_1, l_2 \in L$:

$$l_1 \sqsubseteq l_1 \diamond l_2 \qquad l_2 \sqsubseteq l_1 \diamond l_2$$

Which of the following are upper bound operators?

- $l_1 \diamond l_2 = l_1 \sqcup l_2$
- $l_1 \diamond l_2 = \top$
- $l_1 \diamond l_2 = \begin{cases} l_1 & \text{if } l_2 \sqsubseteq l_1 \\ \top & \text{otherwise} \end{cases}$
- $l_1 \diamond l_2 = \begin{cases} l_2 & \text{if } l_2 \sqsubseteq l_1 \\ \top & \text{otherwise} \end{cases}$

- We can use an upper bound operator to transform an ascending chain into another one.



- The new chain is defined as follows:

$$\begin{array}{rcll} l'_0 & = & l_0 & \\ l'_i & = & l'_{i-1} \diamond l_i & \text{for all } i > 0 \end{array}$$

- If the chain resulting from the transformation always stabilizes regardless of the input chain, we say that $\diamond$ is a **widening operator**.

- Assume a monotonically increasing function $F : L \to L$ on a complete lattice and a widening operator $\diamond$.
- We define a function $F_\diamond : L \to L$ as follows:

$$F_\diamond(x) = \begin{cases} x & \text{if } F(x) \sqsubseteq x \\ x \diamond F(x) & \text{otherwise} \end{cases}$$

- The following ascending chain:

$$\bot \sqsubseteq F_\diamond(x) \sqsubseteq F_\diamond^1(x) \sqsubseteq F_\diamond^2(x) \sqsubseteq F_\diamond^3(x) \sqsubseteq \cdots$$

  always **stabilizes** above the least fixed point of F.

A widening operator in interval analysis

- For example, let us make our chain of intervals stabilize by choosing a set $K \subseteq \mathbb{Z}$ of **stop points** (usually the constants occurring in the program), so that the bounds of the interval are constrained to these.

- If $z' < z$, we push z' until the next point in K, which is

$$max\{k \in K \mid k \leq z'\}$$

- For example, let us make our chain of intervals stabilize by choosing a set $K \subseteq \mathbb{Z}$ of **stop points** (usually the constants occurring in the program), so that the limits of the interval are constrained to these.

- If there were no more points in K to the left, we push it to $-\infty$.

- This idea can be formalized by the *LB* function:

$$LB(z, z') = \begin{cases} z & \text{if } z \leq z' \\ k & \text{if } z' < z \wedge k = max\{k \in K \mid k \leq z'\} \\ -\infty & \text{if } z' < z \wedge \forall k \in K : z' < k \end{cases}$$

- And similarly for the right limit of the interval.

$$UB(z, z') = \begin{cases} z & \text{if } z' \leq z \\ k & \text{if } z < z' \wedge k = min\{k \in K \mid z' \leq k\} \\ +\infty & \text{if } z < z' \wedge \forall k \in K : k < z' \end{cases}$$

- Let us define $\diamond$: **Interval** $\times$ **Interval** $\to$ **Interval** as follows:

$$
\begin{aligned}
\bot \diamond \bot &= \bot \\
[a_1, b_1] \diamond \bot &= [a_1, b_1] \\
\bot \diamond [a_2, b_2] &= [LB(+\infty, a_2), UB(-\infty, b_2)] \\
[a_1, b_1] \diamond [a_2, b_2] &= [LB(a_1, a_2), UB(b_1, b_2)]
\end{aligned}
$$

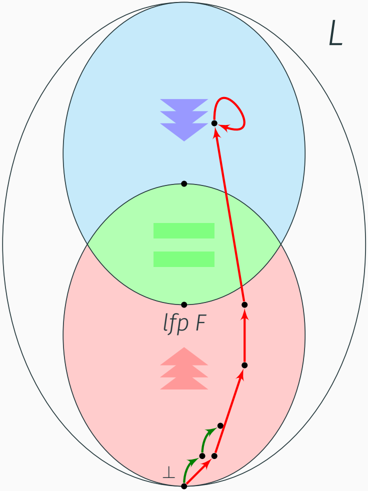
Example

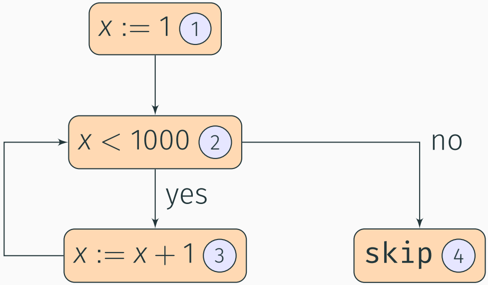
- If $K = \{3, 5\}$, the operator transforms the sequence $[0, 1], [0, 2], [0, 3]$, etc. into:

$$[0, 1], [0, 3], [0, 3], [0, 5], [0, 5], [0, +\infty], [0, +\infty], [0, +\infty], \ldots$$

that stabilizes.

- Assume that Kleene's ascending chain does not stabilize.
- With the widening operator we build another chain that stabilizes in a fixed point or in the reductive zone of the lattice



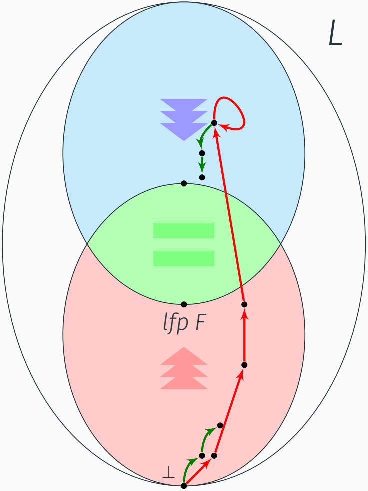


- If $K = \{1, 1000\}$, we get a solution in four iterations:

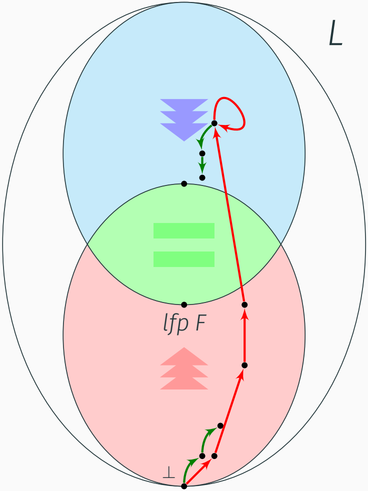
$$
\begin{aligned}
In_1^\sharp &= [x \mapsto [0, 0]] \\
In_2^\sharp &= [x \mapsto [1, 1000]] \\
In_3^\sharp &= [x \mapsto [1, 999]] \\
In_4^\sharp &= [x \mapsto [1000, 1000]]
\end{aligned}
$$

Now what?

- The transformed chain has stabilized at a given point.
- From there we can apply F again to get closer to the fixpoint zone.
- The descending chain might not stabilize, but it is safe to stop it at any place.



- If we want this descending chain to stabilize, or just accelerate its convergence, we need a **narrowing** operator.



Narrowing operators

- We say that an operator $\triangleright : L \times L \to L$ is a **narrowing operator** if it satisfies:
  - For every l_1, l_2 such that $l_2 \sqsubseteq l_1$ it holds that $l_2 \sqsubseteq (l_1 \triangleright l_2) \sqsubseteq l_1$.

$$\begin{tikzcd} l_1 \arrow[d, dash] \\ l_1 \triangleright l_2 \arrow[d, dash] \\ l_2 \end{tikzcd}$$

  - For any descending chain, if we use $\triangleright$ to transform it, the resulting chain always stabilizes.

- For any monotonically increasing function $F : L \rightarrow L$ and a narrowing operator $\triangleright$, let us define $G_{\triangleright} : L \rightarrow L$ as follows:

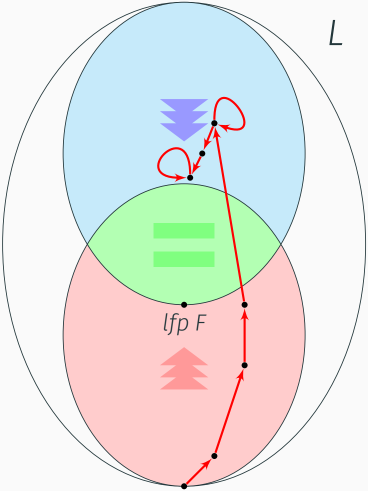
$$G_{\triangleright}(x) = x \triangleright F(x)$$

- Then, for any l such that F is reductive in l, we can build the following chain:

$$l \sqsupseteq G_{\triangleright}(l) \sqsupseteq G_{\triangleright}^{2}(l) \sqsupseteq G_{\triangleright}^{3}(l) \sqsupseteq \cdots$$

that always stabilizes above the least fixed point of F.

- The widening sequence led us to a point in *RED*.
- The narrowing sequence descends until its stabilization above *lfp F*.



- In $(\mathbf{Interval}, \sqsubseteq)$ there are two kinds of infinite descending chains:

$$[z_1, +\infty] \sqsupset [z_2, +\infty] \sqsupset [z_3, +\infty] \sqsupset \cdots \quad \text{if } z_1 < z_2 < z_3 < \cdots$$

$$[-\infty, z_1] \sqsupset [-\infty, z_2] \sqsupset [-\infty, z_3] \sqsupset \cdots \quad \text{if } z_1 > z_2 > z_3 > \cdots$$

- If we focus on the chains of the first kind, we can define a narrowing operator that forces the chain to stabilize when z_i reaches some threshold value N.

$$\begin{aligned} \bot \rhd int &= \bot \\ int \rhd \bot &= \bot \\ [a_1, b_1] \rhd [a_2, b_2] &= [z_1, z_2] \end{aligned} \qquad \text{where} \qquad \begin{aligned} z_1 &= \begin{cases} a_1 & \text{if } N < a_2 \text{ and } b_1 = +\infty \\ a_2 & \text{otherwise} \end{cases} \\ z_2 &= \begin{cases} b_1 & \text{if } b_2 < -N \text{ and } a_2 = -\infty \\ b_2 & \text{otherwise} \end{cases} \end{aligned}$$

- For example, if we have the following nonstable chain:

  $$[0, +\infty] \sqsupset [1, +\infty] \sqsupset [2, +\infty] \sqsupset [3, +\infty] \sqsupset [4, +\infty] \sqsupset \cdots$$

- We get the following sequence with $N = 3$:

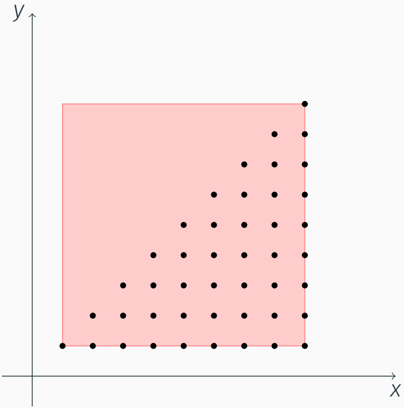
  $$[0, +\infty] \sqsupset [1, +\infty] \sqsupset [2, +\infty] \sqsupset [3, +\infty] \sqsupset [3, +\infty] \sqsupset \cdots$$

  which stabilizes.

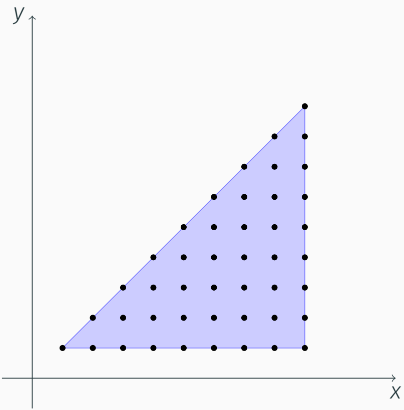
Polyhedral domains

- Our previous analysis computes an interval for each variable.
- Intervals contain standalone information corresponding to each variable.
- However, the values of the variables can be related at runtime:

```
x:=1;
while x < 10 do
   y:=1;
   while y <= x do
      [x↦[1,9],y↦[1,9]][x↦[1,9],y↦[1,x]]
      y:=y+1;
   x:=x+1;
```



- We represent the values (x, y) of program's state before the assignment $y := y + 1$.
- With intervals $x \in [1..9]$, $y \in [1..9]$ we overapproximate the set of possible values.



- We represent the values (x, y) of program's state before the assignment $y := y + 1$.
- With intervals $x \in [1..9]$, $y \in [1..9]$ we overapproximate the set of possible values.
- We can get more accurate approximations by using convex polyhedra.

How would you represent in a program the area shown before?

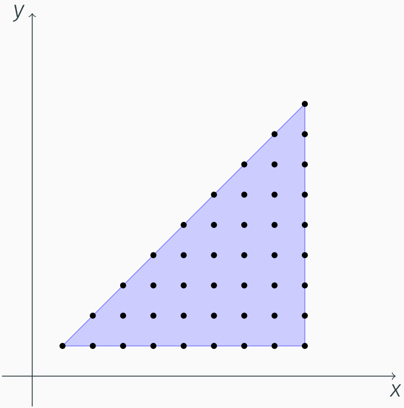
Convex polyhedra

- Assume that **Var** $= \{x_1, \ldots, x_n\}$ are the variables of the program we are analysing.
- Given some real numbers $a_1, \ldots, a_n$ and b, the set of $(x_1, \ldots, x_n) \in \mathbb{R}^n$ satisfying the following **constraint**:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$$

  make up a **closed half-space** of $\mathbb{R}^n$.
- A **convex polyhedron** is the intersection of a finite number of closed half-spaces.
- It is usually defined as a set of constraints:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \end{aligned}$$



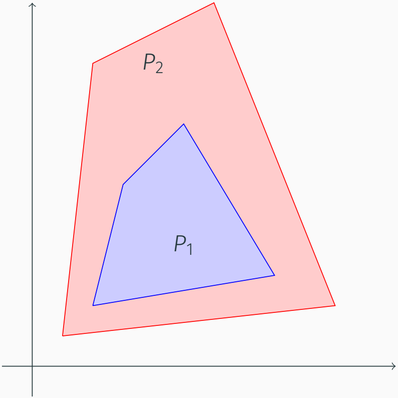
- The polyhedron on the left is represented by these constraints:

$$
\begin{aligned}
x &\leq 9 \\
-y &\leq -1 \\
y - x &\leq 0
\end{aligned}
$$

- A set P is **convex** iff for every pair $x_1, x_2 \in P$, the line segment connecting both lies entirely within P.

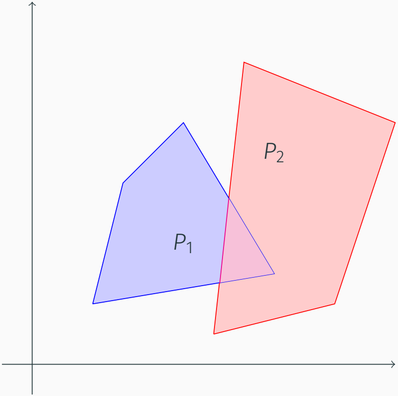
- We want to obtain, at each program point, a convex polyhedron that approximates the relations between variables.
- We use a data-flow approach with abstract interpreation.
- Let us denote by **Poly** the set of convex polyhedra in $\mathbb{R}^n$, where n is the number of variables in the program.
- **Concrate domain:** Sets of states ($\mathcal{P}(\mathbf{State})$).
- **Abstract domain:** Sets of polyhedra ($\mathbf{Poly}$).
- Let us define an order relation $\sqsubseteq$ in $\mathbf{Poly}$ and check whether $(\mathbf{Poly}, \sqsubseteq)$ is a lattice.

Order relation between polyhedra

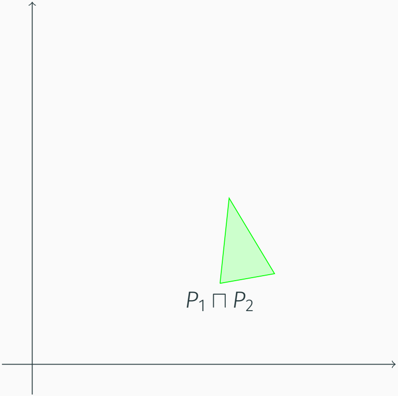


- Given two polyhedra P_1 and P_2, we say that $P_1 \sqsubseteq P_2$ if the set of points in P_1 is contained within P_2.

Greatest lower bounds ($\sqcap$)

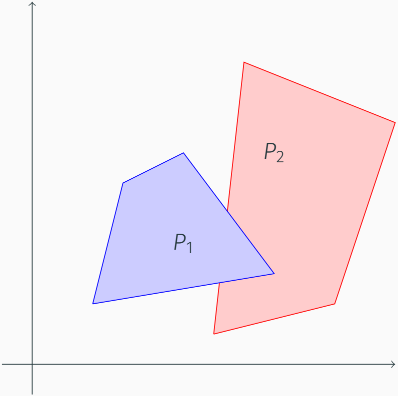


- Given two polyhedra P_1 and P_2, their greatest lower bound $P_1 \sqcap P_2$ is their intersection.



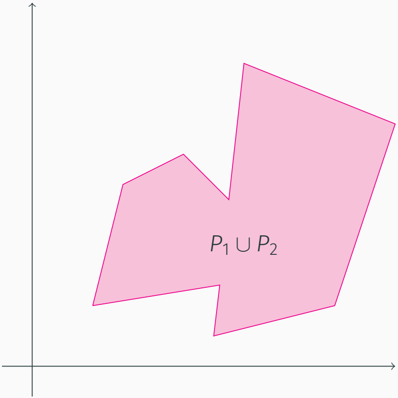
- Given two polyhedra P_1 and P_2, their greatest lower bound $P_1 \sqcap P_2$ is their intersection.

Least upper bounds ($\sqcup$)

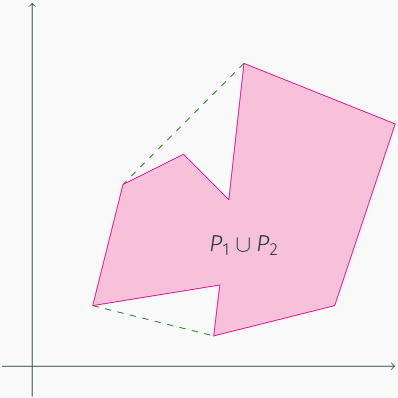


- Given P_1 and P_2, their union is not necessarily a convex polyhedron.

Least upper bounds ($\sqcup$)

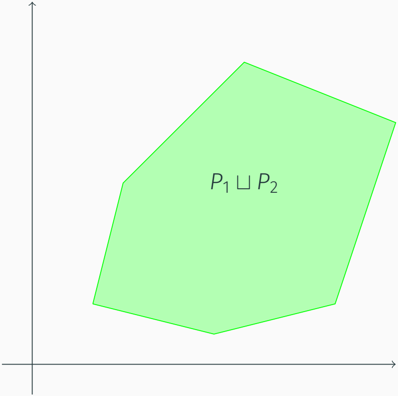


- Given P_1 and P_2, their union is not necessarily a convex polyhedron.
- The least upper bound $P_1 \sqcup P_2$ is the convex hull of $P_1 \cup P_2$.



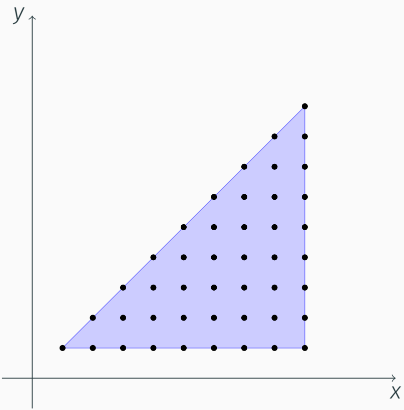
- Given P_1 and P_2, their union is not necessarily a convex polyhedron.
- The least upper bound $P_1 \sqcup P_2$ is the convex hull of $P_1 \cup P_2$.
- The convex hull of X (denoted by $\mathcal{C}(X)$) is the smallest convex polyhedron that contains X.

Least upper bounds ($\sqcup$)



- Given P_1 and P_2, their union is not necessarily a convex polyhedron.
- The least upper bound $P_1 \sqcup P_2$ is the convex hull of $P_1 \cup P_2$.
- The convex hull of X (denoted by $\mathcal{C}(X)$) is the smallest convex polyhedron that contains X.

- The ordered set $(\mathsf{Poly}, \sqsubseteq)$ is a complete lattice.
- $\bot$: empty set.
- $\top$: whole space $\mathbb{R}^n$.
- Let us define a Galois connection $(\mathcal{P}(\mathsf{State}), \alpha, \gamma, \mathsf{Poly})$ where:
    - $\alpha : \mathcal{P}(\mathsf{State}) \to \mathsf{Poly}$
    - $\gamma : \mathsf{Poly} \to \mathcal{P}(\mathsf{State})$



- Given a set Σ of states, each $\sigma \in \Sigma$ represents a point in $\mathbb{R}^n$.
- The abstraction of Σ is the smalles polyhedron containing all the points represented by Σ.
- This is the convex hull of the corresponding points.

- Assume that **Var** $= \{x_1, \ldots, x_n\}$ is the set of program variables.
- Given any set of states $\Sigma \in \mathcal{P}(\mathbf{State})$, its abstraction is defined as follows:

$$\alpha(\Sigma) = \mathcal{C}\left(\{(\sigma(x_1), \ldots, \sigma(x_n)) \mid \sigma \in \Sigma\}\right)$$

- Given a polyhedron $P \in \mathbf{Poly}$, its concretization is the set of states resulting from each point of integer coordinates inside P.

$$\gamma(P) = \{[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n] \mid (y_1, \ldots, y_n) \in P, y_1, \ldots, y_n \in \mathbb{Z}\}$$

- Our lattice of properties is $(\mathbf{Poly}, \sqsubseteq)$.
- The extreme value $\iota^\sharp$ is $\mathbb{R}^n$, that is, $\top$ de $\mathbf{Poly}$.
  - We could also use $\alpha(\sigma_0)$, where σ_0 is an initial state.
- Given m, let us define the transfer function $f^\sharp_m : \mathbf{Poly} \to \mathbf{Poly}$.
  - If the m-th block contains a `skip` or a boolean condition, then $f^\sharp_m = id$.
  - If the m-th block is an assignment, we have to distinguish cases according to the expression at the right-hand side of the assignment.

Case 1: Invertible assignment of linear expression

- Assume the following:

$$\downarrow In_n$$

$$\boxed{x_i := a_1 * x_1 + \cdots + a_i * x_i + \cdots + a_n * x_n + b \quad m}$$

$$\downarrow Out_n$$

- We handle the case in which we assign a linear expression to x_i, where $a_i \neq 0$.
  - That is, the variable being updated occurs in the right-hand side of the assignment.
- For example: $x := x + 1$, $y := 2 * x - 3 * y - 2$, etc.

Case 1: Invertible assignment of linear expression

- If x_i' is the value of x_i after the assignment:

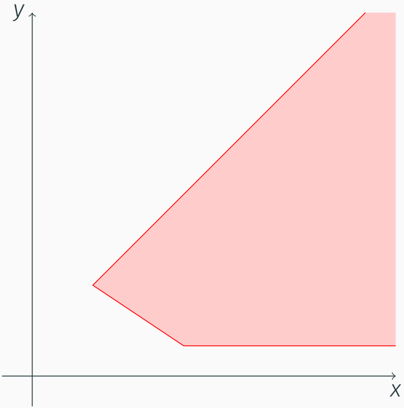
$$x_i' = a_1x_1 + a_2x_2 + \cdots + a_ix_i + \cdots + a_nx_n + b$$

- We solve x_i as a function of x_i':

$$x_i = -\frac{a_1}{a_i}x_1 - \frac{a_2}{a_i}x_2 - \cdots + \frac{1}{a_i}x_i' - \cdots - \frac{a_n}{a_i}x_n - \frac{b}{a_i}$$

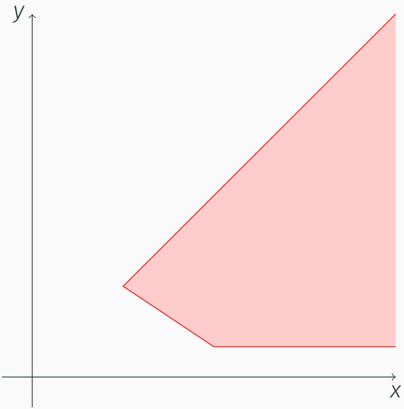
- Let P' be the polyhedron resulting from replacing x_i in the constraints of P by $-\frac{a_1}{a_i}x_1 - \frac{a_2}{a_i}x_2 - \cdots + \frac{1}{a_i}x_i - \cdots - \frac{a_n}{a_i}x_n - \frac{b}{a_i}$.
- We define $f_m^\sharp$ as follows:

$$f_m^\sharp(P) = P'$$



- Given the polyhedron defined by:

$$y \geq 1$$
$$x + y \geq 5$$
$$x - y \geq -1$$



- Given the polyhedron defined by:

$$\begin{aligned} y &\geq 1 \\ x + y &\geq 5 \\ x - y &\geq -1 \end{aligned}$$

- Assignment $x := x + 1$ transforms it into:

$$\begin{aligned} y &\geq 1 \\ x + y &\geq 6 \\ x - y &\geq 0 \end{aligned}$$

Case 2: Noninvertible assignment of linear expression

$$\big\downarrow In_n$$

$$\boxed{x_i := a_1 * x_1 + \cdots + a_i * x_i + \cdots + a_n * x_n + b \quad m}$$

$$\big\downarrow Out_n$$

- Again, we have a linear expression, but now $a_i = 0$.
    - That is, the variable being assigned to does not appear in the right-hand side of the assignment.
- For example: $x := 2 * y + 3$, $y := 5$.

Case 2: Noninvertible assignment of linear expression

- First we eliminate the information on the previous value of x_i by **projecting** the polyhedron on the remaining variables.
  - Given a polyhedron P, we define its projection onto $\{x_1, \ldots, x_{i-1}, x_i, \ldots x_n\}$ as follows:

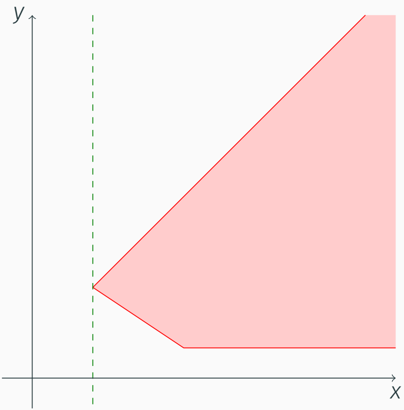
    $$Proj_{x_i}(P) = \{(x_1, \ldots, y_i, \ldots, x_n) \mid (x_1, \ldots x_i, \ldots, x_n) \in P, y_i \in \mathbb{R}\}$$

T. Huynh, C. Lassez, J-L. Lassez
Practical issues on the projection of polyhedral sets
Annals of Mathematics and Artificial Intelligence 6 (1992) 295-316

- Then we add the constraint $x_i = a_1x_1 + \cdots + a_nx_n + b$.
- Summarizing:

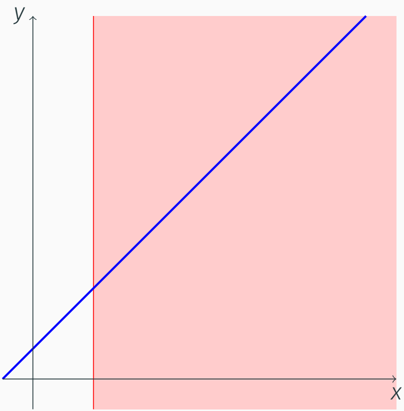
$$f_m^\sharp(P) = Proj_{x_i}(P) \sqcap \{x_i = a_1x_1 + \cdots + a_nx_n + b\}$$

Case 2: Example



- Given the polyhedron shown before, assume we execute $y := x + 1$. First we project onto x:

$$x \geq 2$$



- Given the polyhedron shown before, assume we execute $y := x + 1$. First we project onto x:

$$x \geq 2$$

- Now we add the constraint $y = x + 1$:

$$x \geq 2$$
$$y = x + 1$$

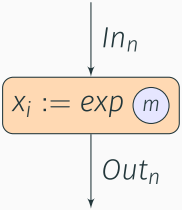
- Given the polyhedron shown before, assume we execute $y := x + 1$. First we project onto x:

$$x \geq 2$$

- Now we add the constraint $y = x + 1$:

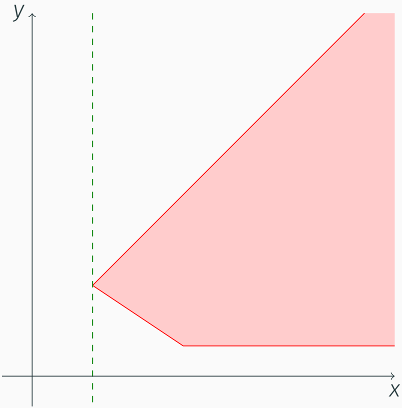
$$x \geq 2$$
$$y = x + 1$$

Case 3: Assignment of nonlinear expression



- For example: $x := y * x + 3$.
- The region is no longer a polyhedron, and it is difficult, in general, to find a polyhedron that encompasses this region.
- Therefore, we assume that x_i may take any value in $\mathbb{R}$. We just eliminate the information involving x_i from the input polyhedron.

$$f^\sharp(P) = Proj_{x_i}(P)$$

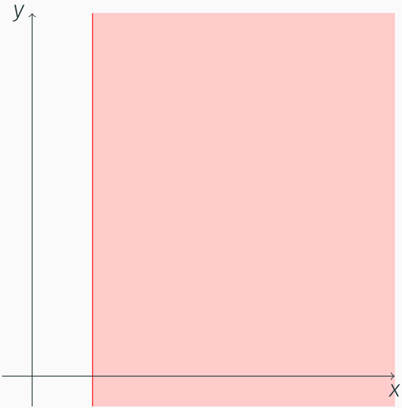


- Given P defined by:

$$y \geq 1$$
$$x + y \geq 5$$
$$x - y \geq -1$$

- Assignment $y := y * x$ transforms it into $Proy_y(P)$:

$$x \geq 2$$

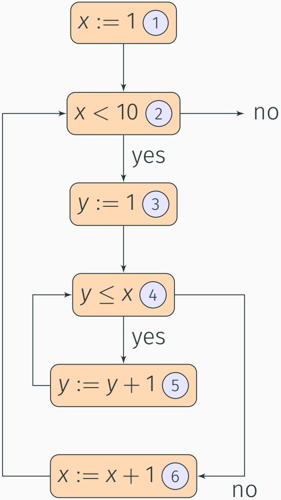


- Given P defined by:

$$\begin{aligned} y &\geq 1 \\ x + y &\geq 5 \\ x - y &\geq -1 \end{aligned}$$

- Assignment $y := y * x$ transforms it into $Proy_y(P)$:

$$x \geq 2$$



$$
\begin{aligned}
In_1^\sharp &= \top \\
In_2^\sharp &= Out_1^\sharp \sqcup Out_6^\sharp \\
In_3^\sharp &= Out_2^\sharp \sqcap \{x \leq 9\} \\
In_4^\sharp &= Out_3^\sharp \sqcup Out_5^\sharp \\
In_5^\sharp &= Out_4^\sharp \sqcap \{y \leq x\} \\
In_6^\sharp &= Out_4^\sharp \sqcap \{y \geq x + 1\}
\end{aligned}
$$

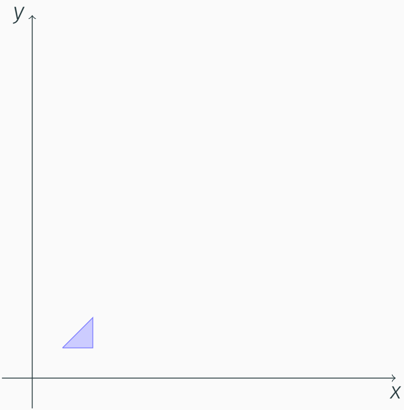
$$
\begin{aligned}
Out_1^\sharp &= \{x = 1\} \\
Out_2^\sharp &= In_2^\sharp \\
Out_3^\sharp &= Proy_y(In_3^\sharp) \sqcap \{y = 1\} \\
Out_4^\sharp &= In_4^\sharp \\
Out_5^\sharp &= f_5^\sharp(In_5^\sharp) \\
Out_6^\sharp &= f_6^\sharp(In_5^\sharp)
\end{aligned}
$$

- If we iterate we get the following sequence for $In_5^\sharp$:

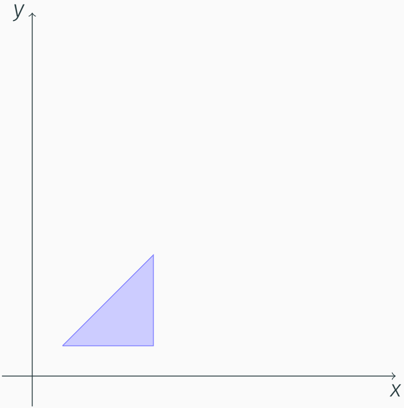
$$
\begin{aligned}
\bot \to \cdots \to \{&x = 1, 1 \le y \le x\} \\
\to \cdots \to \{&1 \le x \le 2, 1 \le y \le x\} \\
\to \cdots \to \{&1 \le x \le 3, 1 \le y \le x\} \\
\to \cdots \to \{&1 \le x \le 4, 1 \le y \le x\} \\
&\vdots \\
\to \{&1 \le x \le 9, 1 \le y \le x\}
\end{aligned}
$$

- The chain stabilizes after a great number of iterations.
- However, it does not necessarily have to, in general, since $(\mathsf{Poly}, \sqsubseteq)$ does not satisfy the ascending chain condition.

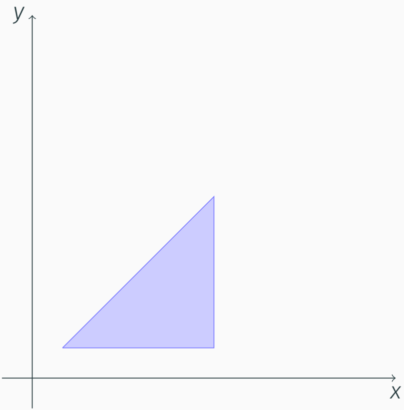
Computing the least fixed point



Computing the least fixed point



Computing the least fixed point



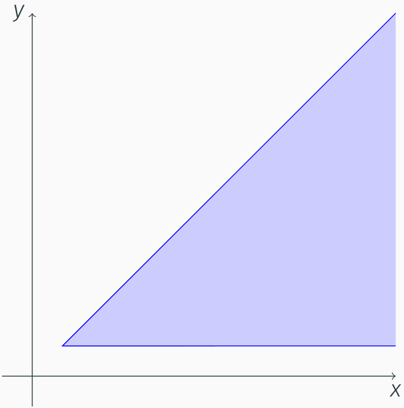
- We need a widening operator that ensures (or accelerates) the stabilization of the chain.

Widening operator

- A simple, but effective widening strategy consists in eliminating the “unstable” limits of the polyhedron.
  - Given P_i, assume that P_{i+1} is the polyhedron resulting from the next iteration.
  - Widening consists in eliminating from P_{i+1} those constraints not covered by any of the constraints in P_i.
- For example:

$$\underbrace{\begin{pmatrix} -x & \leq & -1 \\ x & \leq & 2 \\ y-x & \leq & 0 \\ -y & \leq & -1 \end{pmatrix}}_{P_i} \Longrightarrow \underbrace{\begin{pmatrix} -x & \leq & -1 \\ x & \leq & 3 \\ y-x & \leq & 0 \\ -y & \leq & -1 \end{pmatrix}}_{P_{i+1}}$$

- $x \leq 3$ is not implied by any of the constraints of P_i. We remove it from P_{i+1}.



- From the geometrical point-of-view, this is the same as "pushing towards $+\infty$" the unstable bounds.

- After applying widening, we would reach the following polyhedron in our example:

$$\begin{pmatrix} -x & \leq & -1 \\ y - x & \leq & 0 \\ -y & \leq & -1 \end{pmatrix}$$

- This falls in the reductive zone of **Poly**. We can apply another iteration so as to obtain:

$$\begin{pmatrix} -x & \leq & -1 \\ x & \leq & 9 \\ y - x & \leq & 0 \\ -y & \leq & -1 \end{pmatrix}$$

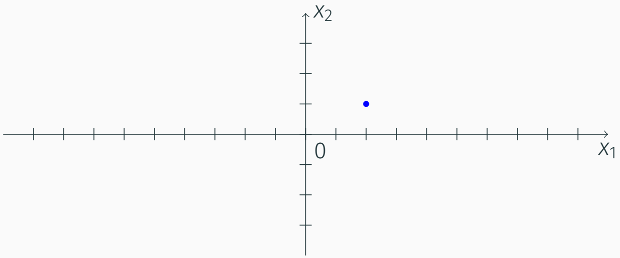
which is, in fact, a fixed point.

- There are more sophisticated narrowing techniques:

R. Bagnara, P. Hill, E. Ricci, E. Zaffanella
Precise widening operators for convex polyhedra
Static Analysis Symposium. Springer. (2003)

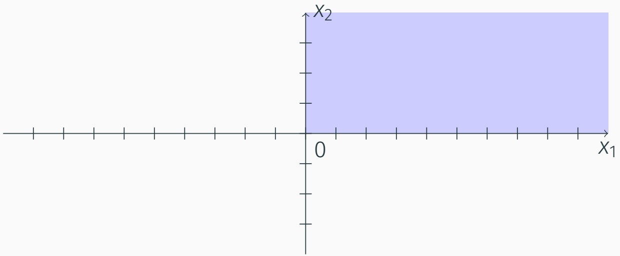
A. Simon, A. King
Widening polyhedra with landmarks
Asian Symposium on Programming Languages and Systems. Springer. (2006)

Summary of numerical domains



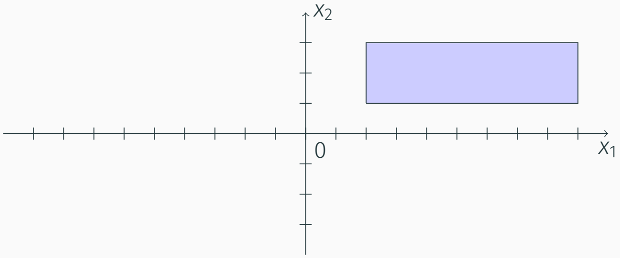
- It allows one to infer properties such as $x_i = k$.

G. Kildall
An unified approach to program optimization
Principles of Programming Languages. ACM. (1973)



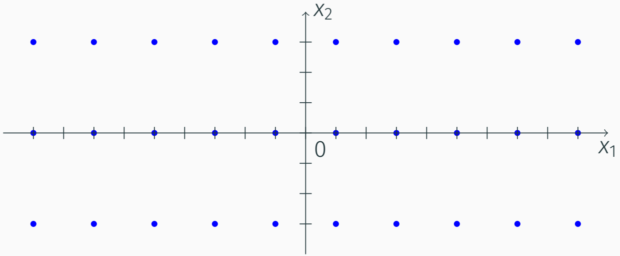
- Properties of the form $x_i > 0, x_i < 0$, or $x_i = 0$.

P. Cousot and R. Cousot.
Static determination of dynamic properties of programs
International Symposium on Programming pp. 106-130 (1976)



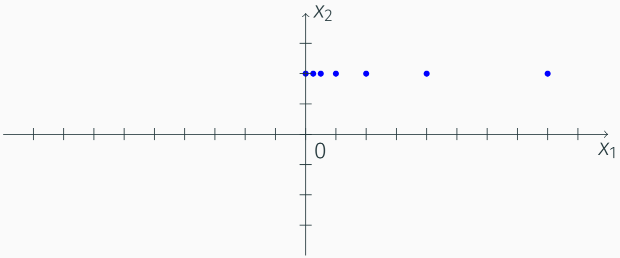
- Properties such as $x_i \in [a_i, b_i]$.

P. Cousot and R. Cousot.
Static determination of dynamic properties of programs
International Symposium on Programming pp. 106-130 (1976)



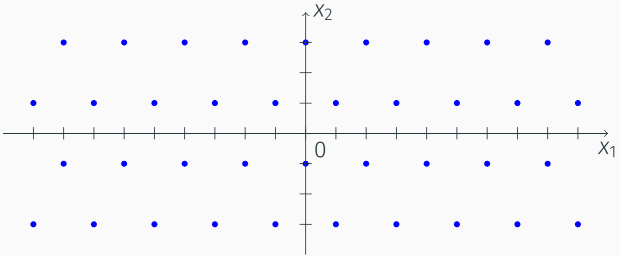
- Properties such as $x_i \equiv a \pmod{b}$.

P. Granger
Static analyses of congruence properties on rational numbers
Static Analysis Symposium, pp 278-292. Springer. 1997



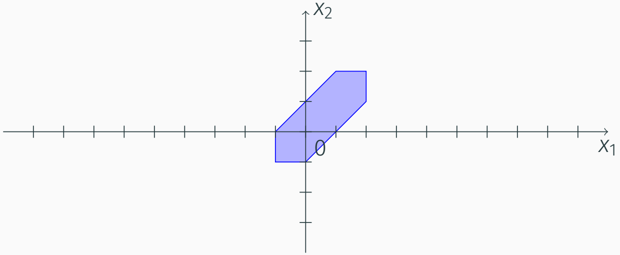
- Properties such as $x_i \in k^{a\mathbb{Z}+b}$.

I. Mastroeni
Numerical Power Analysis
PADO II, pp 117-137. Springer. 2001



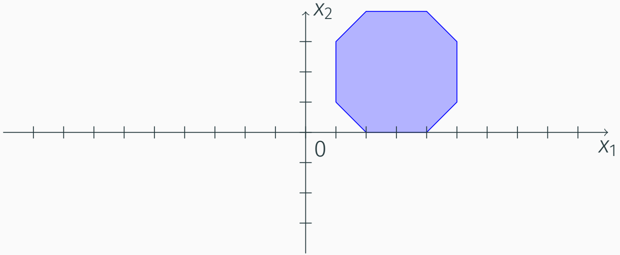
- Properties such as $a_1x_1 + \cdots + a_nx_n \equiv b \pmod{k}$.

P. Granger
Static analyses of congruence properties on rational numbers
Static Analysis Symposium, pp 278-292. Springer. 1997



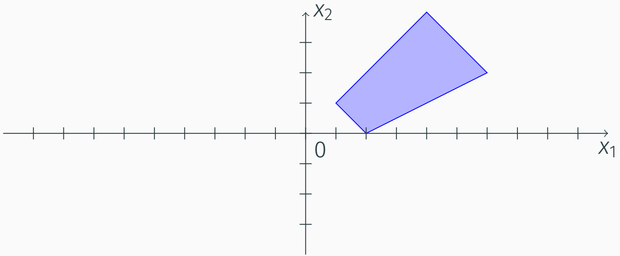
- Properties such as $x_i - x_j \leq k$.

A. Miné
A new numerical abstract domain based on difference-bound matrices
PADO II, pp 155-172. Springer. 2001.



- Properties such as $\pm x_i \pm x_j \leq k$.

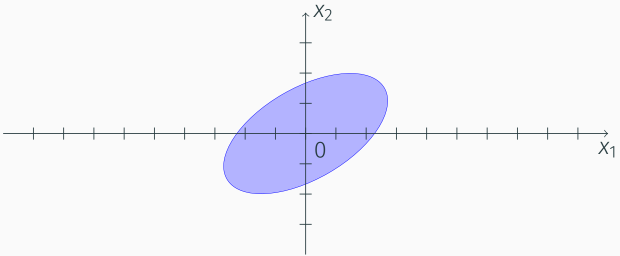
A. Miné
The octagon abstract domain
AST, pp 310-319. IEEE CS Press. 2001.



- Properties such as $a_1x_1 + \cdots + a_nx_n \leq k$.

P. Cousot, N. Halbwachs
Automatic discovery of linear restraints among variables of a program
Principles of Programming Languages, pp 84-97. ACM. 1978.

Relational domains: ellipsoids



- Properties such as $ax_i^2 + bx_j^2 + cx_ix_j \leq k$.

J. Feret
Static analysis of digital filters
European Symposium on Programming. Springer. 2004.

Bibliography

F. Nielson, H.R. Nielson, C. Hankin
Principles of program analysis
Springer, 1999
Capítulo 4.

H.R. Nielson, F. Nielson
Semantics with applications
Springer, 2007

N.D. Jones, F. Nielson
Abstract Interpretation: a semantics-based tool for program analysis
http://www.cs.sunysb.edu/~stoller/cse526/jones-nielson.ps.gz