Formal Analysis and Verification

FAV 2012/2013

Ondřej Lengál, Tomáš Vojnar

`{ilengal,vojnar}@fit.vutbr.cz`

Brno University of Technology
Faculty of Information Technology
Božetěchova 2, 612 66 Brno

Abstract Interpretation

Introduction

- Compared to model checking in which the stress is put on a systematic execution of a system being verified (or its model), the emphasis in static analysis is on minimization of the amount of execution of the code. It is either not executed at all (the case of looking for bug patterns) or just on some abstract level, typically with an in advance fixed abstraction (data flow analysis, abstract interpretation, . . .).
- However, the borderline between model checking and static analysis is not sharp (especially when considering abstract interpretation and model checking based on predicate abstraction).
- Many static analyses are such that they can be applied to parts of code without the need to describe their environment.
- Static analysis approaches: bug pattern analysis, type analysis, dataflow analysis, . . . , abstract interpretation, (and sometimes even model checking).

Abstract Interpretation

- Abstract interpretation was formally defined in 1977 by Patrick and Radhia Cousot.
- Currently used in many automated verification tools.
- It is a very general approach that evaluates a program for all possible inputs at once over abstract domains obtained using abstraction by executing abstract transformers that correspond to statements of the program. This is usually sound but incomplete (i.e., false alarms may appear).
- The abstraction used is generally fixed all the time, while some variations of model checking (e.g., *predicate abstraction* or *abstract regular model checking*) perform refinement using the CEGAR loop driven by the property being verified.

The Abstract Interpretation Loop

- Symbolically execute the verified program.
- For each line, accumulate possible concrete values into abstract contexts.
- When desirable, perform widening to accelerate getting a fixpoint.
- If widening gives a too rough result, try to refine it using narrowing.
- Thanks to the use of abstraction, the loop is guaranteed to terminate.

Galois Connections

- Abstract interpretation can be formally defined using Galois connections.
- Galois connection is a quadruple $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$ such that:
  - $\mathcal{P} = \langle P, \leq \rangle$ and $\mathcal{Q} = \langle Q, \sqsubseteq \rangle$ are *partially ordered sets* (posets),
  - $\alpha : P \to Q$ and $\gamma : Q \to P$ are functions such that $\forall p \in P$ and $\forall q \in Q$:

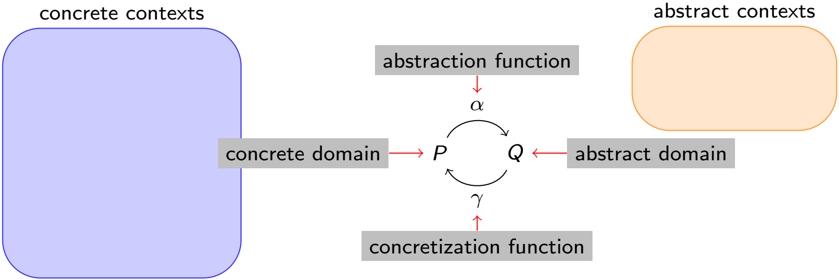
$$p \leq \gamma(q) \iff \alpha(p) \sqsubseteq q$$

Galois Connections

- Abstract interpretation can be formally defined using Galois connections.
- Galois connection is a quadruple $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$ such that:
  - $\mathcal{P} = \langle P, \leq \rangle$ and $\mathcal{Q} = \langle Q, \sqsubseteq \rangle$ are *partially ordered sets* (posets),
  - $\alpha : P \to Q$ and $\gamma : Q \to P$ are functions such that $\forall p \in P$ and $\forall q \in Q$:

$$p \leq \gamma(q) \iff \alpha(p) \sqsubseteq q$$

- In abstract interpretation the elements of π are called



concrete contexts

abstract contexts

abstraction function

concrete domain

abstract domain

concretization function

$$\begin{tikzcd} P \arrow[r, bend left, "\alpha"] & Q \arrow[l, bend left, "\gamma"] \end{tikzcd}$$

Galois Connections

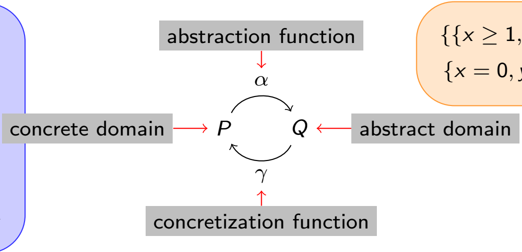
- Abstract interpretation can be formally defined using Galois connections.
- Galois connection is a quadruple $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$ such that:
  - $\mathcal{P} = \langle P, \leq \rangle$ and $\mathcal{Q} = \langle Q, \sqsubseteq \rangle$ are *partially ordered sets* (posets),
  - $\alpha : P \to Q$ and $\gamma : Q \to P$ are functions such that $\forall p \in P$ and $\forall q \in Q$:

$$p \leq \gamma(q) \iff \alpha(p) \sqsubseteq q$$

- In abstract interpretation the elements of π are called

concrete contexts

$\{\{x = 1, y = 0\},$
$\{x = 2, y = 0\},$
$\{x = 3, y = 0\}, \dots,$
$\{x = 0, y = 1\},$
$\{x = 0, y = 2\},$
$\{x = 0, y = 3\}, \dots\}$



$$\begin{tikzcd} P \arrow[r, bend left, "\alpha"] & Q \arrow[l, bend left, "\gamma"] \end{tikzcd}$$

abstraction function

concrete domain

abstract domain

concretization function

abstract contexts

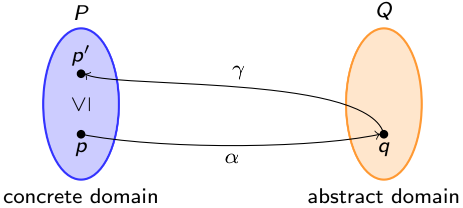
$\{\{x \geq 1, y = 0\},$
$\{x = 0, y \geq 1\}\}$

Galois Connections

- **Implication**: if abstraction and concretization functions of an abstract interpretation form a Galois connection, the abstract interpretation may only over-approximate (never under-approximate).

Proof.

$$\begin{array}{lcccl} & \alpha(p) \sqsubseteq q & \iff & p \leq \gamma(q) & \Rightarrow \\ \Rightarrow & \alpha(p) = q & \Rightarrow & p \leq \gamma(q) & \Rightarrow \\ \Rightarrow & \alpha(p) = q & \Rightarrow & p \leq \gamma(\alpha(p)) & \end{array}$$



□

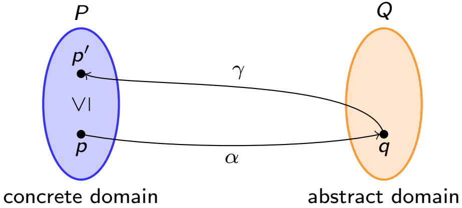
Galois Connections

- **Implication**: if abstraction and concretization functions of an abstract interpretation form a Galois connection, the abstract interpretation may only over-approximate (never under-approximate).

Proof.

$$\begin{array}{rlcll} & \alpha(p) \sqsubseteq q & \iff & p \leq \gamma(q) & \Rightarrow \\ \Rightarrow & \alpha(p) = q & \Rightarrow & p \leq \gamma(q) & \Rightarrow \\ \Rightarrow & \alpha(p) = q & \Rightarrow & p \leq \gamma(\alpha(p)) & \end{array}$$

□



- For every function $f_P : P^n \to P$, there exists a corresponding function $f_Q : Q^n \to Q$:

$$\alpha(f_P(p_1, \ldots, p_n)) \sqsubseteq f_Q(\alpha(p_1), \ldots, \alpha(p_n)).$$

Informal Intuitive Introduction

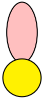
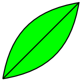
- We will informally present abstract interpretation using a graphical language.
- Concrete objects: $P =$ finite sets of coloured pixels.

Informal Intuitive Introduction

- Concrete operations:
    - nullary (constants), $\mathtt{op} : P^0 \to P \qquad (P^0 = \{\emptyset\})$

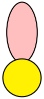
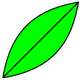
Informal Intuitive Introduction

- Concrete operations:
  - nullary (constants), $\texttt{op} : P^0 \to P \qquad (P^0 = \{\emptyset\})$

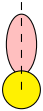
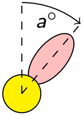
$\texttt{petal} : \emptyset \mapsto$  $\texttt{stem} : \emptyset \mapsto$ $\texttt{leaf} : \emptyset \mapsto$ 

Informal Intuitive Introduction

- Concrete operations:
  - nullary (constants), op : $P^0 \to P$ $\qquad (P^0 = \{\emptyset\})$

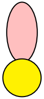
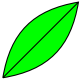
$\texttt{petal} : \emptyset \mapsto$  $\qquad \texttt{stem} : \emptyset \mapsto$ $\qquad \texttt{leaf} : \emptyset \mapsto$ 

  - unary, op : $P^1 \to P$

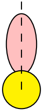
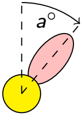
$\forall a \in \mathbb{R} : \qquad \texttt{rotation}[a] :$  $\mapsto$ 

Informal Intuitive Introduction

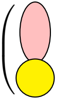
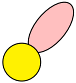
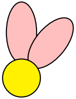
- Concrete operations:
  - nullary (constants), $\mathrm{op} : P^0 \to P \qquad (P^0 = \{\emptyset\})$

    $\texttt{petal} : \emptyset \mapsto$  $\texttt{stem} : \emptyset \mapsto$ $\texttt{leaf} : \emptyset \mapsto$ 

  - unary, $\mathrm{op} : P^1 \to P$

    $\forall a \in \mathbb{R} : \qquad \texttt{rotation}[a] :$  $\mapsto$ 

  - binary, $\mathrm{op} : P^2 \to P$

    $[\cdot \cup \cdot] : \Bigg($  ,  $\Bigg) \mapsto$ 

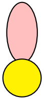
Informal Intuitive Introduction

- We can also define our own functions

Informal Intuitive Introduction

- We can also define our own functions

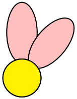
```
petal
```



Informal Intuitive Introduction

- We can also define our own functions

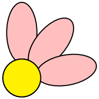
$$\texttt{petal} \cup \texttt{rotation}[40](\texttt{petal})$$



Informal Intuitive Introduction

- We can also define our own functions

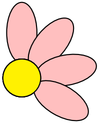
$$\texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal})$$



Informal Intuitive Introduction

- We can also define our own functions

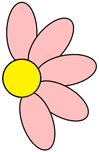
$$\mathtt{petal} \cup \mathtt{rotation}[40](\mathtt{petal}) \cup \mathtt{rotation}[80](\mathtt{petal}) \cup \mathtt{rotation}[120](\mathtt{petal})$$



Informal Intuitive Introduction

- We can also define our own functions

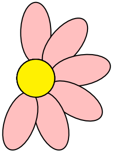
$$\texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup \texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal})$$



Informal Intuitive Introduction

- We can also define our own functions

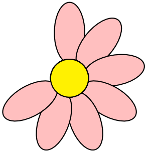
$$\texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup \texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup \texttt{rotation}[200](\texttt{petal})$$



Informal Intuitive Introduction

- We can also define our own functions

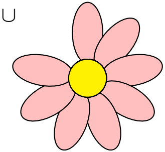
$$\texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup$$
$$\texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup$$
$$\texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal})$$



Informal Intuitive Introduction

- We can also define our own functions

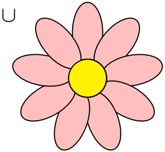
$\texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup$
$\texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup$
$\texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal}) \cup$
$\texttt{rotation}[280](\texttt{petal})$



Informal Intuitive Introduction

- We can also define our own functions

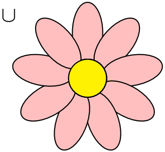
$$\begin{aligned}&\texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup\\ &\texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup\\ &\texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal}) \cup\\ &\texttt{rotation}[280](\texttt{petal}) \cup \texttt{rotation}[320](\texttt{petal})\end{aligned}$$



Informal Intuitive Introduction

- We can also define our own functions

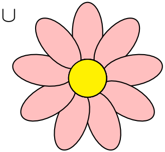
$$\begin{aligned}
& \texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup \\
& \texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup \\
& \texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal}) \cup \\
& \texttt{rotation}[280](\texttt{petal}) \cup \texttt{rotation}[320](\texttt{petal}) \\
= \quad & \mu Z \,.\, \texttt{petal} \cup \texttt{rotation}[40](Z)
\end{aligned}$$



Informal Intuitive Introduction

- We can also define our own functions

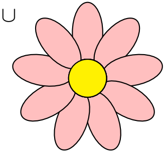
$$\begin{aligned}\texttt{corolla} \quad &:= \quad \texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup \\ &\quad\quad \texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup \\ &\quad\quad \texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal}) \cup \\ &\quad\quad \texttt{rotation}[280](\texttt{petal}) \cup \texttt{rotation}[320](\texttt{petal}) \\ &= \quad \mu Z \,.\, \texttt{petal} \cup \texttt{rotation}[40](Z)\end{aligned}$$



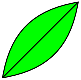
Informal Intuitive Introduction

- We can also define our own functions

$$\begin{aligned}\texttt{corolla} \quad &:= \quad \texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup \\ & \quad\ \texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup \\ & \quad\ \texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal}) \cup \\ & \quad\ \texttt{rotation}[280](\texttt{petal}) \cup \texttt{rotation}[320](\texttt{petal}) \\ &= \quad \mu Z \,.\, \texttt{petal} \cup \texttt{rotation}[40](Z)\end{aligned}$$



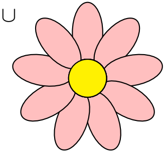
leaf



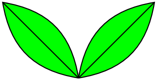
Informal Intuitive Introduction

- We can also define our own functions

$$\begin{aligned}\texttt{corolla} \quad &:= \quad \texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup \\ &\qquad \texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup \\ &\qquad \texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal}) \cup \\ &\qquad \texttt{rotation}[280](\texttt{petal}) \cup \texttt{rotation}[320](\texttt{petal}) \\ &= \quad \mu Z \,.\, \texttt{petal} \cup \texttt{rotation}[40](Z)\end{aligned}$$



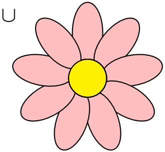
$$\texttt{leaf} \cup \texttt{rotation}[-90](\texttt{leaf})$$



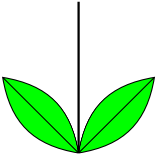
Informal Intuitive Introduction

- We can also define our own functions

$$\begin{aligned}\texttt{corolla} \quad := \quad & \texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup \\ & \texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup \\ & \texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal}) \cup \\ & \texttt{rotation}[280](\texttt{petal}) \cup \texttt{rotation}[320](\texttt{petal}) \\ = \quad & \mu Z \, . \, \texttt{petal} \cup \texttt{rotation}[40](Z)\end{aligned}$$



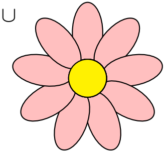
$$\texttt{leaf} \cup \texttt{rotation}[-90](\texttt{leaf}) \cup \texttt{stem}$$



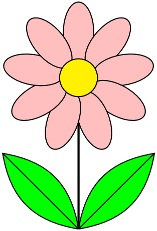
Informal Intuitive Introduction

- We can also define our own functions

$$\begin{aligned}\texttt{corolla} \quad &:= \quad \texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup \\ &\qquad \texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup \\ &\qquad \texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal}) \cup \\ &\qquad \texttt{rotation}[280](\texttt{petal}) \cup \texttt{rotation}[320](\texttt{petal}) \\ &= \quad \mu Z \,.\, \texttt{petal} \cup \texttt{rotation}[40](Z)\end{aligned}$$



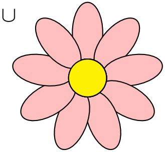
$$\texttt{leaf} \cup \texttt{rotation}[-90](\texttt{leaf}) \cup \texttt{stem} \cup \texttt{corolla}$$



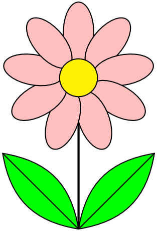
Informal Intuitive Introduction

- We can also define our own functions

$$\begin{aligned}
\texttt{corolla} \quad &:= \quad \texttt{petal} \cup \texttt{rotation}[40](\texttt{petal}) \cup \texttt{rotation}[80](\texttt{petal}) \cup \\
&\qquad \texttt{rotation}[120](\texttt{petal}) \cup \texttt{rotation}[160](\texttt{petal}) \cup \\
&\qquad \texttt{rotation}[200](\texttt{petal}) \cup \texttt{rotation}[240](\texttt{petal}) \cup \\
&\qquad \texttt{rotation}[280](\texttt{petal}) \cup \texttt{rotation}[320](\texttt{petal}) \\
&= \quad \mu Z \,.\, \texttt{petal} \cup \texttt{rotation}[40](Z)
\end{aligned}$$



$$\texttt{flower} \quad := \quad \texttt{leaf} \cup \texttt{rotation}[-90](\texttt{leaf}) \cup \texttt{stem} \cup \texttt{corolla}$$

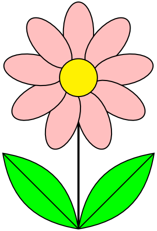


Informal Intuitive Introduction

- Over-approximation of an object is an object with more pixels (which may overlap) when considering a subtractive colour model.

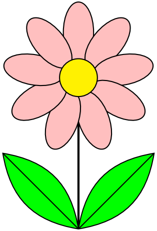
Informal Intuitive Introduction

- Over-approximation of an object is an object with more pixels (which may overlap) when considering a subtractive colour model.
- Example:

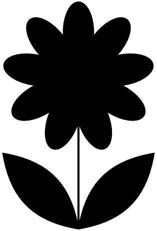


Informal Intuitive Introduction

- Over-approximation of an object is an object with more pixels (which may overlap) when considering a subtractive colour model.
- Example:

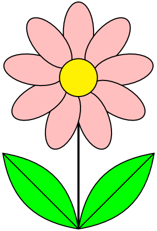


$\subseteq$

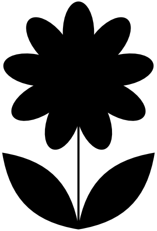


Informal Intuitive Introduction

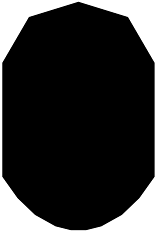
- Over-approximation of an object is an object with more pixels (which may overlap) when considering a subtractive colour model.
- Example:



$\subseteq$



$\subseteq$

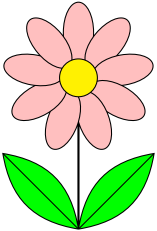


Informal Intuitive Introduction

- Abstract object: mathematical representation of an approximation of a concrete object.

Informal Intuitive Introduction

- Abstract object: mathematical representation of an approximation of a concrete object.
- Example:

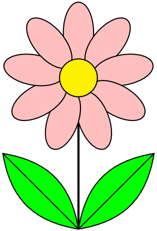


concrete object

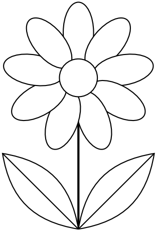
Informal Intuitive Introduction

- Abstract object: mathematical representation of an approximation of a concrete object.

- Example:



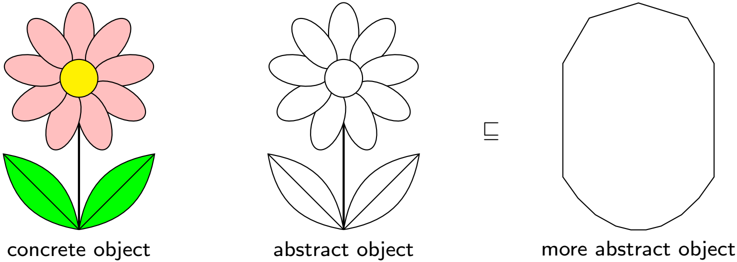
concrete object



abstract object

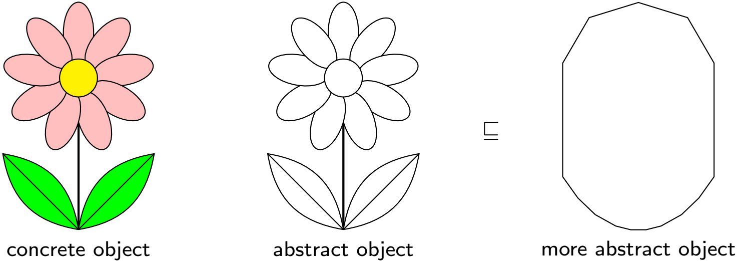
Informal Intuitive Introduction

- Abstract object: mathematical representation of an approximation of a concrete object.
- Example:



Informal Intuitive Introduction

- Abstract object: mathematical representation of an approximation of a concrete object.
- Example:



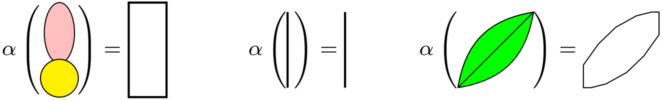
- We choose bounding convex polygon as an abstraction α for our example.
- Concretization γ then maps polygon to all inner pixels of the polygon.

Informal Intuitive Introduction

- Abstract operations:
    - nullary (constants), $\mathrm{op} : Q^0 \to Q \qquad (Q^0 = \{\emptyset\})$

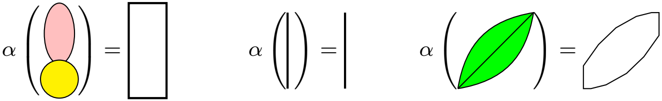
Informal Intuitive Introduction

- Abstract operations:
  - nullary (constants), op : $Q^0 \to Q$ $\qquad (Q^0 = \{\emptyset\})$



Informal Intuitive Introduction

- Abstract operations:
  - nullary (constants), op : $Q^0 \to Q$ $\qquad (Q^0 = \{\emptyset\})$

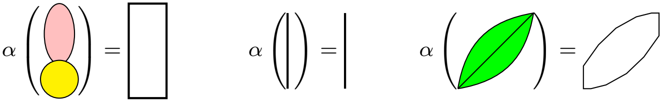


  - unary, op : $Q^1 \to Q$

$$\forall a \in \mathbb{R} : \qquad \texttt{rotation}_\alpha[a] = \lambda x \,.\, \alpha(\texttt{rotation}[a](\gamma(x)))$$

Informal Intuitive Introduction

- Abstract operations:
  - nullary (constants), op : $Q^0 \rightarrow Q \qquad (Q^0 = \{\emptyset\})$



  - unary, op : $Q^1 \rightarrow Q$

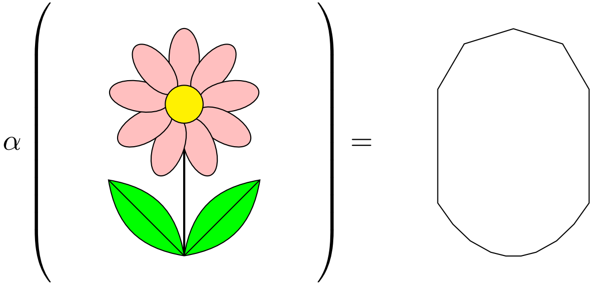
$$\forall a \in \mathbb{R} : \qquad \texttt{rotation}_\alpha[a] = \lambda x \,.\, \alpha(\texttt{rotation}[a](\gamma(x)))$$

  - binary, op : $P^2 \rightarrow P$

$$[\cdot \cup_\alpha \cdot] = \lambda x\ y \,.\, \alpha(\gamma(x) \cup \gamma(y))$$

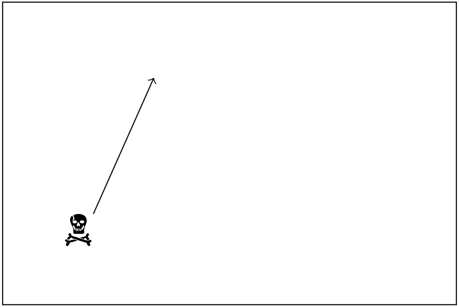
Informal Intuitive Introduction

- Example:



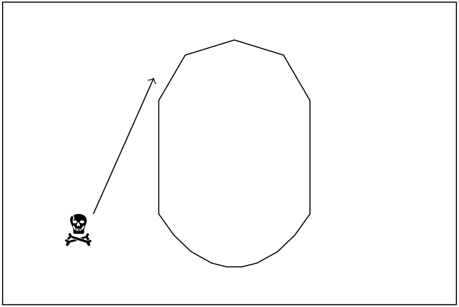
Informal Intuitive Introduction

- Verification question: is there a green pixel (bug) at a specific point in the picture?



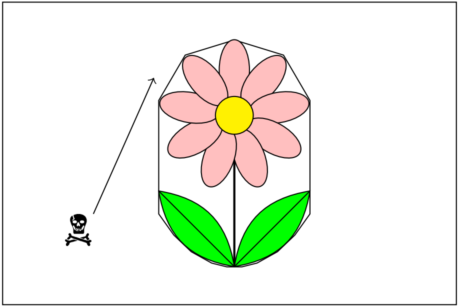
Informal Intuitive Introduction

- Verification question: is there a green pixel (bug) at a specific point in the picture?



Informal Intuitive Introduction

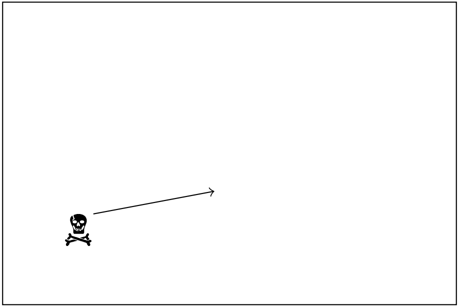
- Verification question: is there a green pixel (bug) at a specific point in the picture?



- Answer: No. (correct)

Informal Intuitive Introduction

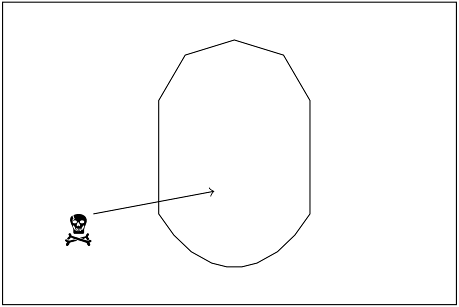
- Verification question: is there a green pixel (bug) at a specific point in the picture?



- Answer: No. (correct)

Informal Intuitive Introduction

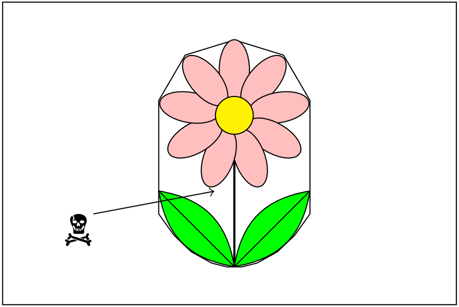
- Verification question: is there a green pixel (bug) at a specific point in the picture?



- Answer: No. (correct)
- Answer: Yes.

Informal Intuitive Introduction

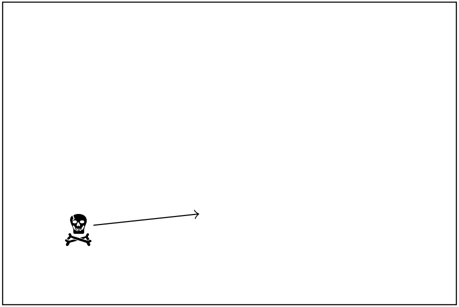
- Verification question: is there a green pixel (bug) at a specific point in the picture?



- Answer: No. (correct)
- Answer: Yes. (incorrect)

Informal Intuitive Introduction

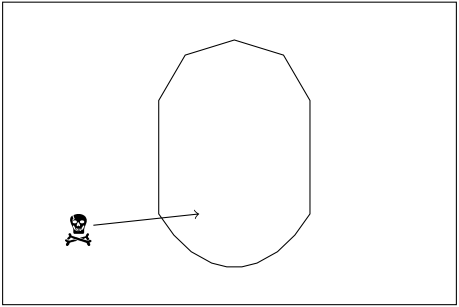
- Verification question: is there a green pixel (bug) at a specific point in the picture?



- Answer: No. (correct)
- Answer: Yes. (incorrect)
- Answer: Yes.

Informal Intuitive Introduction

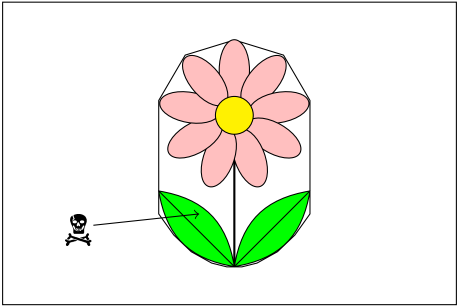
- Verification question: is there a green pixel (bug) at a specific point in the picture?



- Answer: No. (correct)
- Answer: Yes. (incorrect)
- Answer: Yes.

Informal Intuitive Introduction

- Verification question: is there a green pixel (bug) at a specific point in the picture?



- Answer: No. (correct)
- Answer: Yes. (incorrect)
- Answer: Yes. (correct)

Abstract Interpretation

- A semilattice is a poset with a join (least upper bound, supremum) for every finite subset of its base set.

Abstract Interpretation

- A semilattice is a poset with a join (least upper bound, supremum) for every finite subset of its base set.
- Abstract interpretation I of a program P with the instruction set `Instr` is a tuple

$$I = (Q, \circ, \sqsubseteq, \top, \bot, \tau)$$

  where

  - Q is the abstract domain (domain of abstract contexts),
  - $\top \in Q$ is the supremum of Q,
  - $\bot \in Q$ is the infimum of Q (we suppose it exists),
  - $\circ : Q \times Q \to Q$ is the operation for accumulation of abstract contexts (for example on branch and loop junctions), together with Q and $\top$ yielding the complete semilattice $(Q, \circ, \top)$,
  - $(\sqsubseteq) \subseteq Q \times Q$ is an ordering defined as $x \sqsubseteq y \iff x \circ y = y$ in $(Q, \circ, \top)$,
  - $\tau : \mathtt{Instr} \times Q \to Q$ defines the interpretation of instructions (i.e. for each program instruction there exists an abstract transformer), τ is monotonic.

Fixpoint Approximation

- In some cases (e.g., loops), computation of the *most precise* abstract fixpoint is not generally guaranteed to terminate (consider *id* as the abstraction function).
- To guarantee termination, the fixpoint can be approximated. This is done using the following two operations:
  - widening: performs over-approximation of a fixpoint,
  - narrowing: refines approximation of a fixpoint.
- Neither widening nor narrowing are necessary, but at least widening is often convenient. Narrowing may be sometimes missing (e.g., in polyhedral analysis).

Widening

- Let $I = (Q, \circ, \sqsubseteq, \top, \bot, \tau)$ be an abstract interpretation of a program.
- The binary widening operation $\triangledown$ is defined as:
    - $\triangledown : Q \times Q \to Q$,
    - $\forall C, D \in Q : (C \circ D) \sqsubseteq (C \triangledown D)$,
    - for all infinite sequences $(C_0, \ldots, C_n, \ldots) \in Q^\omega$, it holds that the infinite sequence $(s_0, \ldots, s_n, \ldots)$ defined recursively as

      $$\begin{array}{rcl} s_0 & = & C_0, \\ s_n & = & s_{n-1} \triangledown C_n \end{array}$$

      is not strictly increasing (and because the result of $\triangledown$ is an upper bound, the sequence eventually stabilizes).
- Widening can be applied later in the computation, the later it is applied the more precise is the result (but the computation takes longer time).

Narrowing

- Let $I = (Q, \circ, \sqsubseteq, \top, \bot, \tau)$ be an abstract interpretation of a program.
- The binary narrowing operation $\triangle$ is defined as:
    - $\triangle$: $Q \times Q \to Q$,
    - $\forall C, D \in Q : C \sqsupseteq D \Rightarrow (C \sqsupseteq (C \triangle D) \sqsupseteq D)$,
    - for all infinite sequences $(C_0, \dots, C_n, \dots) \in Q^\omega$, it holds that the infinite sequence $(s_0, \dots, s_n, \dots)$ defined recursively as

      $$\begin{aligned} s_0 &= C_0, \\ s_n &= s_{n-1} \triangle C_n \end{aligned}$$

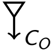
      is not strictly decreasing (and because the result of $C \triangle D$ is a lower bound of C, the sequence eventually stabilizes provided that the input sequence is not strictly increasing).
- Narrowing is performed only after widening.

Representation of a Program

- We choose (deterministic) finite flowcharts as a language independent representation of programs.
- Finite flowchart is a directed graph with 5 types of nodes:
    - entries,
    - assignments,
    - tests,
    - junctions,
    - exits.
- Abstract interpretation iteratively computes abstract contexts for each edge of the flowchart.
- An equation is associated with each edge of the flowchart according to the type of the tail node of the edge.

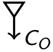
Representation of a Program

- Entry: denotes the entry point of a program. $C_O = \top$



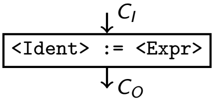
Representation of a Program

- Entry: denotes the entry point of a program. $C_O = \top$



C_O

- Assignment: denotes the assignment A of expression `<Expr>` to the variable `<Ident>`. $C_O = \tau(A, C_I)$



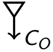
C_I

`<Ident> := <Expr>`

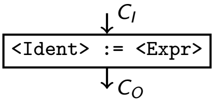
C_O

Representation of a Program

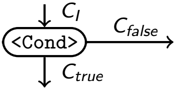
- Entry: denotes the entry point of a program. $C_O = \top$



- Assignment: denotes the assignment A of expression `<Expr>` to the variable `<Ident>`. $C_O = \tau(A, C_I)$

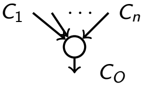


- Test: denotes splitting of the flow to branches B_{true} and B_{false} according to the Boolean condition `<Cond>`. Two context are computed: $C_{true} = \tau(B_{true}, C_I)$ and $C_{false} = \tau(B_{false}, C_I)$



Representation of a Program

- Junction: denotes join J of several branches of code execution (e.g., after `...then ...` and `...else ...` branches of an `if` statement or for a loop join).
  $C_O = \tau(J, C_1 \circ \cdots \circ C_n)$



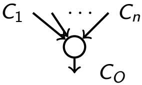
It often holds for junctions that

- $\tau(J) = \lambda x\ .\ x$ — for simple junctions (`if` branches),
- $\tau(J) = \lambda x\ .\ C_p \triangledown x$ — for loop junctions,
- $\tau(J) = \lambda x\ .\ C_p \vartriangle x$ — for loop junctions (only after widening),

where C_p is the abstract context computed for the node in the previous iteration.

Representation of a Program

- Junction: denotes join J of several branches of code execution (e.g., after `...then` `...` and `...else ...` branches of an `if` statement or for a loop join).
  $C_O = \tau(J, C_1 \circ \cdots \circ C_n)$

  It often holds for junctions that
  - $\tau(J) = \lambda x \,.\, x$ — for simple junctions (`if` branches),
  - $\tau(J) = \lambda x \,.\, C_p \nabla x$ — for loop junctions,
  - $\tau(J) = \lambda x \,.\, C_p \vartriangle x$ — for loop junctions (only after widening),

  where C_p is the abstract context computed for the node in the previous iteration.

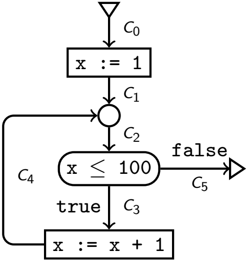
- Exit: denotes the exit point of a program.

Abstract Interpretation Algorithm

1. Derive the system of equations for the program.
2. Solve the equations until the results stabilize.

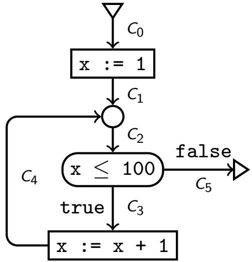
Program Example

- Consider the following flowchart program and interval analysis:



Program Example

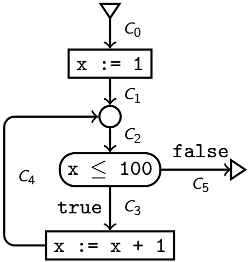
- Consider the following flowchart program and interval analysis:



- We will use notation $[a, b]$ for the predicate $a \le x \le b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ]$ is the null element of $\triangledown$,
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$.

Program Example

- Consider the following flowchart program and interval analysis:

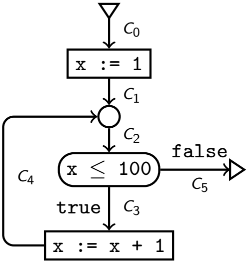


- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ]$ is the null element of $\triangledown$,
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangledown (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



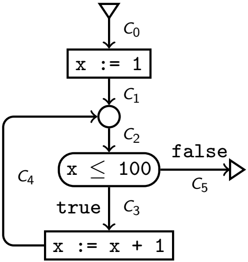
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ]$ is the null element of $\triangledown$,
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$
- $C_2^0 = [\ ,\ ]$
- $C_3^0 = [\ ,\ ]$
- $C_4^0 = [\ ,\ ]$
- $C_5^0 = [\ ,\ ]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangledown (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



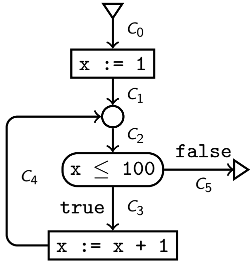
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ]$ is the null element of $\triangledown$,
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$
- $C_3^0 = [\ ,\ ]$
- $C_4^0 = [\ ,\ ]$
- $C_5^0 = [\ ,\ ]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangledown (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



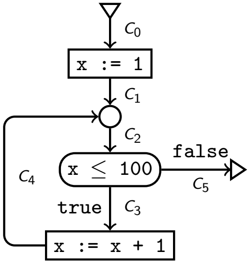
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ]$ is the null element of $\triangledown$,
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$
- $C_3^0 = [\ ,\ ]$
- $C_4^0 = [\ ,\ ]$
- $C_5^0 = [\ ,\ ]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangledown (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



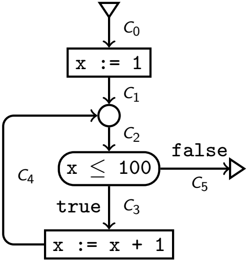
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
  - $[\,,\,]$ is the null element of $\triangledown$,
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\,,\,]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\,,\,]$ $C_2^1 = [1, 1]$
- $C_3^0 = [\,,\,]$ $C_3^1 = [1, 1]$
- $C_4^0 = [\,,\,]$
- $C_5^0 = [\,,\,]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangledown (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



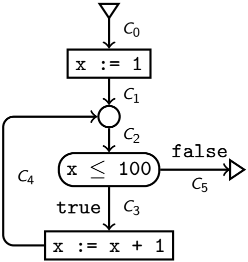
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
    - $[\ ,\ ]$ is the null element of $\triangledown$,
    - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]\ C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]\ C_2^1 = [1, 1]$
- $C_3^0 = [\ ,\ ]\ C_3^1 = [1, 1]$
- $C_4^0 = [\ ,\ ]\ C_4^1 = [2, 2]$
- $C_5^0 = [\ ,\ ]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangledown (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



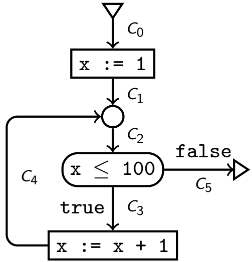
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening ∇ of intervals as:
  - $[\ ,\ ]$ is the null element of ∇,
  - $[i, j]\nabla[k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$ $C_2^2 = [1, +\infty]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$
- $C_4^0 = [\ ,\ ]$ $C_4^1 = [2, 2]$
- $C_5^0 = [\ ,\ ]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2\nabla(C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



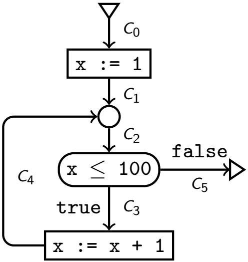
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
    - $[\ ,\ ]$ is the null element of $\triangledown$,
    - $[i, j]\triangledown[k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$ $C_2^2 = [1, +\infty]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$ $C_3^2 = [1, 100]$
- $C_4^0 = [\ ,\ ]$ $C_4^1 = [2, 2]$
- $C_5^0 = [\ ,\ ]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 \;=\; C_2 \triangledown (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



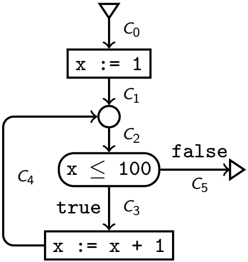
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ]$ is the null element of $\triangledown$,
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$ $C_2^2 = [1, +\infty]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$ $C_3^2 = [1, 100]$
- $C_4^0 = [\ ,\ ]$ $C_4^1 = [2, 2]$ $C_4^2 = [2, 101]$
- $C_5^0 = [\ ,\ ]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangledown (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



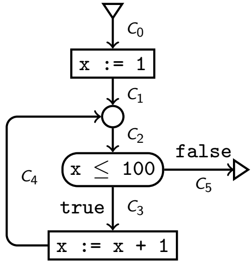
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$.
- Assignments are treated using an integer arithmetic (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Tests are treated using *interval arithmetic*.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ]$ is the null element of $\triangledown$,
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$ $C_2^2 = [1, +\infty]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$ $C_3^2 = [1, 100]$
- $C_4^0 = [\ ,\ ]$ $C_4^1 = [2, 2]$ $C_4^2 = [2, 101]$
- $C_5^0 = [\ ,\ ]$ $C_5^1 = [101, +\infty]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangledown (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



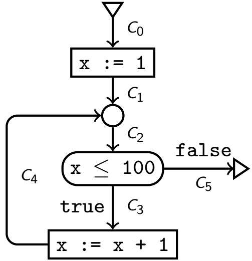
- Let us now define the narrowing operation $\triangle$ over intervals as
  - $[\,,\,]$ is the null element of $\triangle$,
  - $[i,j] \triangle [k,l] = [$
    if $i = -\infty$ then k else $\min(i,k)$,
    if $j = +\infty$ then l else $\max(j,l)]$.
- We now substitute the equation for C_2 with a new one that uses narrowing.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\,,\,]$ $C_1^1 = [1,1]$
- $C_2^0 = [\,,\,]$ $C_2^1 = [1,1]$ $C_2^2 = [1,+\infty]$
- $C_3^0 = [\,,\,]$ $C_3^1 = [1,1]$ $C_3^2 = [1,100]$
- $C_4^0 = [\,,\,]$ $C_4^1 = [2,2]$ $C_4^2 = [2,101]$
- $C_5^0 = [\,,\,]$ $C_5^1 = [101,+\infty]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1,1]$
- $C_2 \;=\; C_2 \triangle (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1,1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



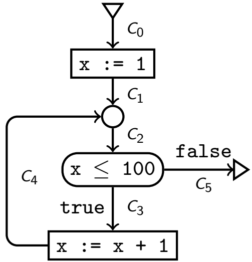
- Let us now define the narrowing operation $\triangle$ over intervals as
  - $[\ ,\ ]$ is the null element of $\triangle$,
  - $[i, j] \triangle [k, l] = [$
    if $i = -\infty$ then k else $\min(i, k)$,
    if $j = +\infty$ then l else $\max(j, l)]$.
- We now substitute the equation for C_2 with a new one that uses narrowing.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$ $C_2^2 = [1, +\infty]$ $C_2^3 = [1, 101]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$ $C_3^2 = [1, 100]$
- $C_4^0 = [\ ,\ ]$ $C_4^1 = [2, 2]$ $C_4^2 = [2, 101]$
- $C_5^0 = [\ ,\ ]$ $C_5^1 = [101, +\infty]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangle (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

Program Example

- Consider the following flowchart program and interval analysis:



- Let us now define the narrowing operation $\triangle$ over intervals as
  - $[\ ,\ ]$ is the null element of $\triangle$,
  - $[i, j] \triangle [k, l] = [$
    if $i = -\infty$ then k else $\min(i, k)$,
    if $j = +\infty$ then l else $\max(j, l)]$.
- We now substitute the equation for C_2 with a new one that uses narrowing.

- $C_0^0 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$ $C_2^2 = [1, +\infty]$ $C_2^3 = [1, 101]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$ $C_3^2 = [1, 100]$
- $C_4^0 = [\ ,\ ]$ $C_4^1 = [2, 2]$ $C_4^2 = [2, 101]$
- $C_5^0 = [\ ,\ ]$ $C_5^1 = [101, +\infty]$ $C_5^2 = [101, 101]$

- $C_0 = [-\infty, +\infty]$
- $C_1 = [1, 1]$
- $C_2 = C_2 \triangle (C_1 \cup C_4)$
- $C_3 = C_2 \cap [-\infty, 100]$
- $C_4 = C_3 + [1, 1]$
- $C_5 = C_2 \cap [101, +\infty]$

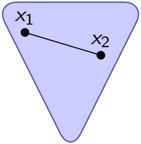
Examples of Abstract Interpretation

- Interval analysis: represents values of variables by intervals of possible values.
- Polyhedral analysis: represents values of variables by a convex polyhedron. This can be used to discover invariants of programs.
- Heap analysis: overapproximates graphs representing the memory heap. This can be used, e.g., for memory leak detection.
- Worst-case execution time (WCET) analysis: may involve the analysis of the cache behaviour, the pipelines, etc.
- and many more . . .

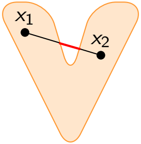
Polyhedral Analysis

Polyhedral Analysis

- An abstract interpretation-based approach of automatic discovering of relations among program variables expressible as linear inequations.
- Let $\vec{x} = x_1, \ldots, x_n \in \mathbb{R}^n$ be the variables of a program. We can use a convex polyhedron $P \subseteq \mathbb{R}^n$ to represent all possible valid assignments to $\vec{x}$.
- We use convex polyhedra because operations on them are reasonably efficient (a set $C \subseteq \mathbb{R}^n$ is convex iff $\forall x_1, x_2 \in C, \forall 0 \leq \lambda \leq 1 \ : \ \lambda x_1 + (1-\lambda)x_2 \in C$).



convex set



concave set

- **Use:** compile-time determination of bounds of variables, discovery of constants, ...

Representation of a Convex Polyhedron

- There are two dual ways to represent a convex polyhedron:
    - by a system of linear inequations, and
    - by the frame of the polyhedron.
- We can alter between these representations. However, after conversion (either way) the result may need to be simplified.
- Efficient execution of different operations require different representation.

System of Linear Inequations

- Let $\vec{x} = x_1, \ldots, x_n \in \mathbb{R}^n$ be the variables of a program. Given a finite set of m linear inequations over $\vec{x}$ of the form

  $$\left\{ \sum_{i=1}^{n} a_{ji} x_i \leq b_j \;\middle|\; 1 \leq j \leq m \right\}$$

  or equivalently using vectors and matrices as

  $$\vec{x} \cdot \mathbf{A} \leq \vec{b}$$

  we can geometrically interpret the solutions of the inequations as a convex polyhedron in $\mathbb{R}^n$ defined by the intersection of *halfspaces* corresponding to each inequality.

The Frame of a Convex Polyhedron

- Convex polyhedron P can also be characterized by its frame $F = (V, R, L)$:
  - vertices V: points $\vec{v}$ of a polyhedron P which are not *convex combinations* of other points $\{\vec{w_1}, \ldots, \vec{w_m}\}$ of P,

$$\left(\left(\vec{v} = \sum_{i=1}^{m} \lambda_i \vec{w_i}\right) \wedge (\forall 1 \leq i \leq m : (\vec{w_i} \in P \wedge \lambda_i \geq 0)) \wedge \left(\sum_{i=1}^{m} \lambda_i = 1\right)\right) \Rightarrow$$

$$\Rightarrow (\forall 1 \leq i \leq m : (\lambda_i = 0 \vee \vec{w_i} = \vec{v}))$$

  - Convex hull: the set of all convex combinations of V.

The Frame of a Convex Polyhedron

- extreme rays R: rays $\vec{r}$ of P (i.e. vectors such that there exists a half-line parallel to $\vec{r}$ and entirely included in P) which are not positive combinations of other rays $\vec{s_1}, \dots, \vec{s_p}$ of P:

$$\left(\vec{r} = \sum_{i=1}^{p} \mu_i \vec{s_i} \wedge \left(\forall 1 \leq i \leq p : \mu_i \in \mathbb{R}^+\right)\right) \Rightarrow \left(\forall 1 \leq i \leq p : \left(\mu_i = 0 \vee \vec{s_i} = \vec{r}\right)\right)$$

The Frame of a Convex Polyhedron

- extreme rays R: rays $\vec{r}$ of P (i.e. vectors such that there exists a half-line parallel to $\vec{r}$ and entirely included in P) which are not positive combinations of other rays $\vec{s_1}, \ldots, \vec{s_p}$ of P:

$$\left(\vec{r} = \sum_{i=1}^{p} \mu_i \vec{s_i} \wedge \left(\forall 1 \leq i \leq p : \mu_i \in \mathbb{R}^+\right)\right) \Rightarrow (\forall 1 \leq i \leq p : (\mu_i = 0 \vee \vec{s_i} = \vec{r}))$$

- lines L: vectors $\vec{l}$ such that both $\vec{l}$ and $-\vec{l}$ are rays of P:

$$\forall \vec{x} \in P, \forall \mu \in \mathbb{R} : \vec{x} + \mu \vec{l} \in P$$

The Frame of a Convex Polyhedron

- extreme rays R: rays $\vec{r}$ of P (i.e. vectors such that there exists a half-line parallel to $\vec{r}$ and entirely included in P) which are not positive combinations of other rays $\vec{s_1}, \ldots, \vec{s_p}$ of P:

$$\left(\vec{r} = \sum_{i=1}^{p} \mu_i \vec{s_i} \wedge \left(\forall 1 \leq i \leq p : \mu_i \in \mathbb{R}^+\right)\right) \Rightarrow (\forall 1 \leq i \leq p : (\mu_i = 0 \vee \vec{s_i} = \vec{r}))$$

- lines L: vectors $\vec{l}$ such that both $\vec{l}$ and $-\vec{l}$ are rays of P:

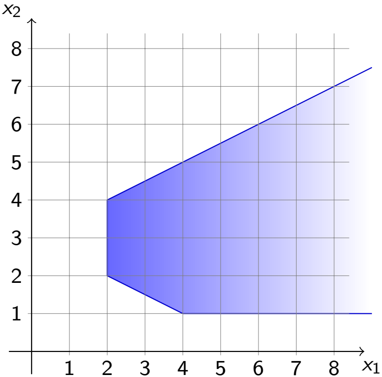
$$\forall \vec{x} \in P, \forall \mu \in \mathbb{R} : \vec{x} + \mu \vec{l} \in P$$

- Every point $\vec{x}$ of the polyhedron P defined by the frame $F = (V, R, L)$ can be obtained from V, R and L:

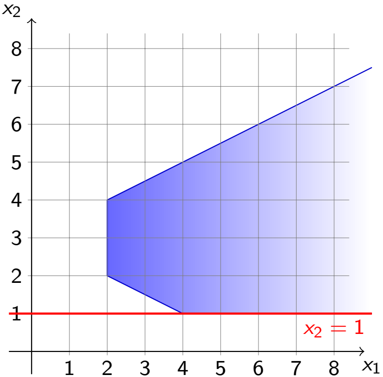
$$\vec{x} = \sum_{i=1}^{\sigma} \lambda_i \vec{v_i} + \sum_{j=1}^{\rho} \mu_j \vec{r_j} + \sum_{k=1}^{\delta} \nu_k \vec{l_k}$$

$$\text{where} \quad 0 \leq \lambda_1, \ldots, \lambda_\sigma \leq 1, \sum_{i=1}^{\sigma} \lambda_i = 1, \mu_1, \ldots, \mu_\rho \in \mathbb{R}^+, \nu_1, \ldots, \nu_\delta \in \mathbb{R}$$

Example of a Convex Polyhedron



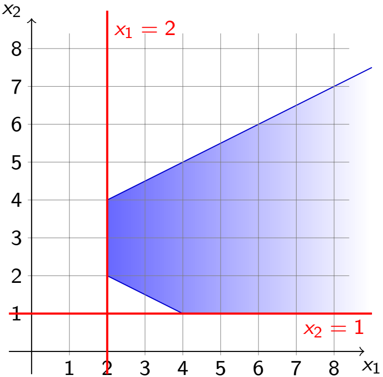
Example of a Convex Polyhedron



System of linear inequations

$$x_2 \geq 1$$

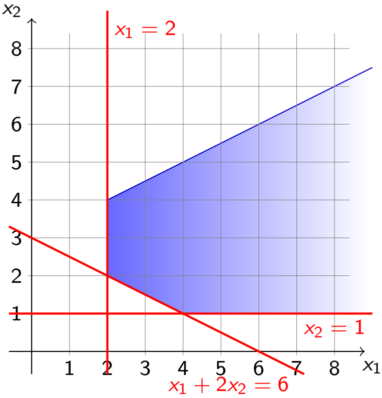
Example of a Convex Polyhedron



System of linear inequations

$$\begin{aligned} x_2 &\geq 1 \\ x_1 &\geq 2 \end{aligned}$$

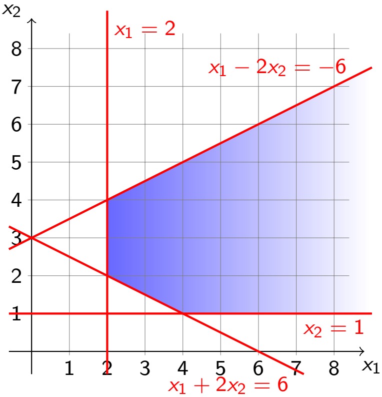
Example of a Convex Polyhedron



System of linear inequations

$$\begin{aligned} x_2 &\geq 1 \\ x_1 &\geq 2 \\ x_1 + 2x_2 &\geq 6 \end{aligned}$$

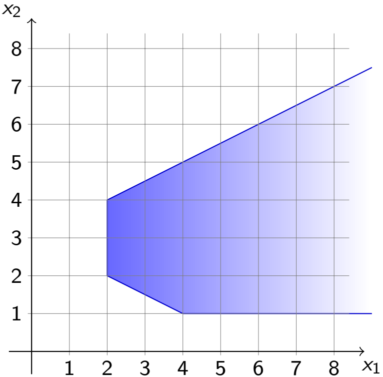
Example of a Convex Polyhedron



System of linear inequations

$$\begin{aligned}
x_2 &\geq 1 \\
x_1 &\geq 2 \\
x_1 + 2x_2 &\geq 6 \\
x_1 - 2x_2 &\geq -6
\end{aligned}$$

Example of a Convex Polyhedron



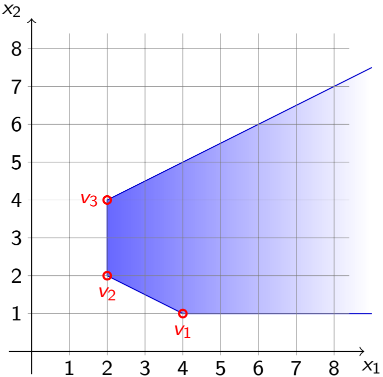
System of linear inequations

$$
\begin{aligned}
x_2 &\geq 1 \\
x_1 &\geq 2 \\
x_1 + 2x_2 &\geq 6 \\
x_1 - 2x_2 &\geq -6
\end{aligned}
$$

Frame of a polyhedron

$$F = (V, R, L)$$

Example of a Convex Polyhedron



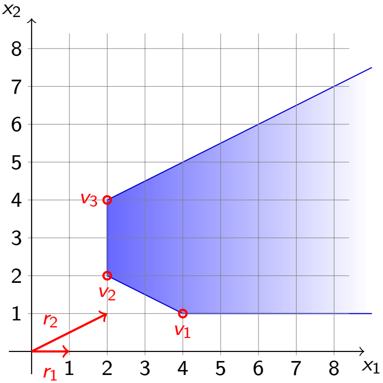
System of linear inequations

$$\begin{aligned} x_2 &\geq 1 \\ x_1 &\geq 2 \\ x_1 + 2x_2 &\geq 6 \\ x_1 - 2x_2 &\geq -6 \end{aligned}$$

Frame of a polyhedron

$$\begin{aligned} F &= (V, R, L) \\ V &= \{\vec{v_1} = [4,1], \vec{v_2} = [2,2], \vec{v_3} = [2,4]\} \end{aligned}$$

Example of a Convex Polyhedron



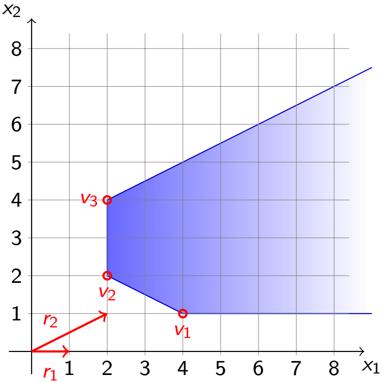
System of linear inequations

$$\begin{aligned} x_2 &\geq 1 \\ x_1 &\geq 2 \\ x_1 + 2x_2 &\geq 6 \\ x_1 - 2x_2 &\geq -6 \end{aligned}$$

Frame of a polyhedron

$$\begin{aligned} F &= (V, R, L) \\ V &= \{\vec{v_1} = [4, 1], \vec{v_2} = [2, 2], \vec{v_3} = [2, 4]\} \\ R &= \{\vec{r_1} = (1, 0), \vec{r_2} = (2, 1)\} \end{aligned}$$

Example of a Convex Polyhedron



System of linear inequations

$$\begin{aligned} x_2 &\geq 1 \\ x_1 &\geq 2 \\ x_1 + 2x_2 &\geq 6 \\ x_1 - 2x_2 &\geq -6 \end{aligned}$$

Frame of a polyhedron

$$\begin{aligned} F &= (V, R, L) \\ V &= \{\vec{v_1} = [4, 1], \vec{v_2} = [2, 2], \vec{v_3} = [2, 4]\} \\ R &= \{\vec{r_1} = (1, 0), \vec{r_2} = (2, 1)\} \\ L &= \emptyset \end{aligned}$$

Transformations of Convex Polyhedra

- Different types of nodes of the flowchart representation of a program perform distinct transformation on the polyhedron. The number of input and output polyhedra differs according to the type of the node.
- Entries: create a polyhedron according to restraints on the input values of variables (in case there are none for variable x_i, the polyhedron is unbounded in i-th dimension).

Assignments

- Performed operations vary according to assigned expression:
    - non-linear expression $x_i :=$ `<non-linear expression>`: because these cannot be represented using convex polyhedra, any restraint on x_i is dropped (we add line $\vec{d}$ to frame such that $d_i = 1$ and $\forall 1 \leq j \leq n, i \neq j : d_j = 0$).
    - linear expression $x_i := \sum_{j=1}^{n} a_j x_j + b$: the frame $F' = (V', R', L')$ of the output polyhedron can be computed from the frame $F = (V, R, L)$ of the input as

Assignments

- Performed operations vary according to assigned expression:
  - non-linear expression $x_i :=$ `<non-linear expression>`: because these cannot be represented using convex polyhedra, any restraint on x_i is dropped (we add line $\vec{d}$ to frame such that $d_i = 1$ and $\forall 1 \leq j \leq n, i \neq j : d_j = 0$).
  - linear expression $x_i := \sum_{j=1}^{n} a_j x_j + b$: the frame $F' = (V', R', L')$ of the output polyhedron can be computed from the frame $F = (V, R, L)$ of the input as
    - $V' = \{\vec{v'_1}, \ldots, \vec{v'_\sigma}\}$ where $\vec{v'_j}$ is defined by $v'_{ji} = \vec{a}\vec{v_j} + b$ and $v'_{mi} = v_{mi}$ where $\forall 1 \leq m \leq \sigma, m \neq j$.

Assignments

- Performed operations vary according to assigned expression:
  - non-linear expression x_i := `<non-linear expression>`: because these cannot be represented using convex polyhedra, any restraint on x_i is dropped (we add line $\vec{d}$ to frame such that $d_i = 1$ and $\forall 1 \leq j \leq n, i \neq j : d_j = 0$).
  - linear expression $x_i := \sum_{j=1}^{n} a_j x_j + b$: the frame $F' = (V', R', L')$ of the output polyhedron can be computed from the frame $F = (V, R, L)$ of the input as
    - $V' = \{\vec{v_1'}, \ldots, \vec{v_\sigma'}\}$ where $\vec{v_j'}$ is defined by $v_{ji}' = \vec{a}\vec{v_j} + b$ and $v_{mi}' = v_{mi}$ where $\forall 1 \leq m \leq \sigma, m \neq j$.
    - $R' = \{\vec{r_1'}, \ldots, \vec{r_\delta'}\}$ where $\vec{r_j'}$ is defined by $r_{ji}' = \vec{a}\vec{r_j}$ and $r_{jm}' = r_{jm}$ for $\forall 1 \leq m \leq \rho, m \neq j$.

Assignments

- Performed operations vary according to assigned expression:
    - non-linear expression x_i := `<non-linear expression>`: because these cannot be represented using convex polyhedra, any restraint on x_i is dropped (we add line $\vec{d}$ to frame such that $d_i = 1$ and $\forall 1 \leq j \leq n, i \neq j : d_j = 0$).
    - linear expression $x_i := \sum_{j=1}^{n} a_j x_j + b$: the frame $F' = (V', R', L')$ of the output polyhedron can be computed from the frame $F = (V, R, L)$ of the input as
        - $V' = \{\vec{v_1'}, \ldots, \vec{v_\sigma'}\}$ where $\vec{v_j'}$ is defined by $v_{ji}' = \vec{a}\vec{v_j} + b$ and $v_{mi}' = v_{mi}$ where $\forall 1 \leq m \leq \sigma, m \neq j$.
        - $R' = \{\vec{r_1'}, \ldots, \vec{r_\delta'}\}$ where $\vec{r_j'}$ is defined by $r_{ji}' = \vec{a}\vec{r_j}$ and $r_{jm}' = r_{jm}$ for $\forall 1 \leq m \leq \rho, m \neq j$.
        - $L' = \{\vec{l_1'}, \ldots, \vec{l_\delta'}\}$ where $\vec{l_j'}$ is defined by $l_{ji}' = \vec{a}\vec{l_j}$ and $l_{jm}' = l_{jm}$ for $\forall 1 \leq m \leq \delta, m \neq j$.

Tests

- The input polyhedron P is transformed into two output polyhedra: P_t for the `true` branch and P_f for the `false` branch.
- For Boolean condition C it should hold $P_t = P \cap C, P_f = P \setminus C$ (but it is not guaranteed that these are convex polyhedra).
- The operation that is performed varies according to the Boolean condition of the test:

Tests

- The input polyhedron P is transformed into two output polyhedra: P_t for the `true` branch and P_f for the `false` branch.
- For Boolean condition C it should hold $P_t = P \cap C, P_f = P \setminus C$ (but it is not guaranteed that these are convex polyhedra).
- The operation that is performed varies according to the Boolean condition of the test:
    - Non-linear tests: $P_t = P_f = P$ (can be refined for some cases)

Tests

- The input polyhedron P is transformed into two output polyhedra: P_t for the `true` branch and P_f for the `false` branch.

- For Boolean condition C it should hold $P_t = P \cap C, P_f = P \setminus C$ (but it is not guaranteed that these are convex polyhedra).

- The operation that is performed varies according to the Boolean condition of the test:
  - Non-linear tests: $P_t = P_f = P$ (can be refined for some cases)
  - Linear equality tests: Boolean condition $C : \vec{a}\vec{x} = b$ defines a *hyperplane* H. If P is included in H then $P_t = P, P_f = \emptyset$. If P is not included in H then $P_t = P \cap H$ and $P_f = P$.

Tests

- The input polyhedron P is transformed into two output polyhedra: P_t for the `true` branch and P_f for the `false` branch.
- For Boolean condition C it should hold $P_t = P \cap C, P_f = P \setminus C$ (but it is not guaranteed that these are convex polyhedra).
- The operation that is performed varies according to the Boolean condition of the test:
  - Non-linear tests: $P_t = P_f = P$ (can be refined for some cases)
  - Linear equality tests: Boolean condition $C : \vec{a}\vec{x} = b$ defines a *hyperplane* H. If P is included in H then $P_t = P, P_f = \emptyset$. If P is not included in H then $P_t = P \cap H$ and $P_f = P$.
  - Linear inequality tests: for Boolean condition $\vec{a}\vec{x} \leq b$ the outputs are $P_t = P \cap \vec{a}\vec{x} \leq b$ and $P_f = P \cap \vec{a}\vec{x} \geq b$.

Junctions

- Junctions correspond to merge of several program paths so the output polyhedron P is union of all input polyhedra P_i. It is computed according to the kind of the junction:

Junctions

- Junctions correspond to merge of several program paths so the output polyhedron P is union of all input polyhedra P_i. It is computed according to the kind of the junction:
    - Simple junctions: for input polyhedra $P_1, \ldots, P_m$ we compute the convex hull of $P_1 \cup \cdots \cup P_m$

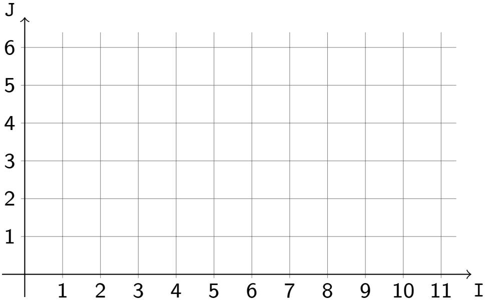
Junctions

- Junctions correspond to merge of several program paths so the output polyhedron P is union of all input polyhedra P_i. It is computed according to the kind of the junction:
  - Simple junctions: for input polyhedra $P_1, \dots, P_m$ we compute the convex hull of $P_1 \cup \dots \cup P_m$
  - Loop junctions: for input polyhedra $P_1, \dots, P_m$ let Q be the convex hull of $P_1 \cup \dots \cup P_m$. Then $P' = P \triangledown Q$ is the convex polyhedron consisting of linear restraints of P satisfied by every element of Q.

Example

```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```

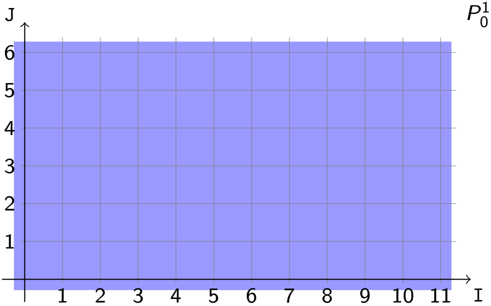
(P_0)
(P_1)
(P_2)
(P_3)
(P_4)
(P_5)
(P_6)
(P_7)



Example

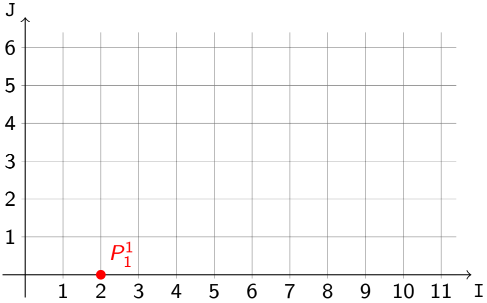
```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```

(P_0)
(P_1)
(P_2)
(P_3)
(P_4)
(P_5)
(P_6)
(P_7)



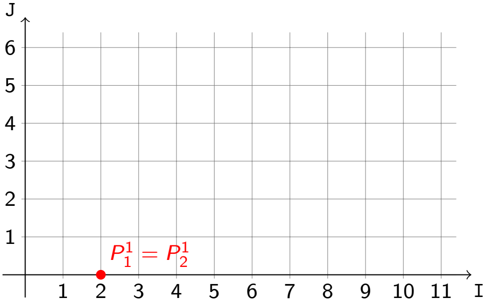
Example

```
                                     (P0)
I := 2, J := 0;                      (P1)
L:                                   (P2)
if ... then                          (P3)
  I := I + 4                         (P4)
else                                 (P5)
  J := J + 1, I := I + 2;            (P6)
fi;                                  (P7)
go to L;
```



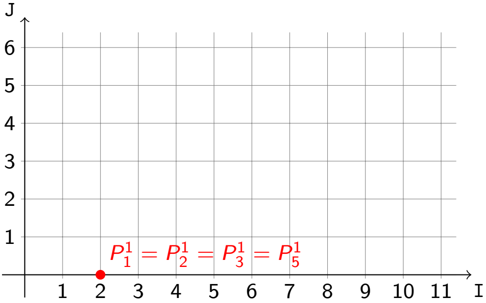
Example

```
I := 2, J := 0;        (P_0)
L:                     (P_1)
if ... then            (P_2)
  I := I + 4           (P_3)
else                   (P_4)
  J := J + 1, I := I + 2;   (P_5)
fi;                    (P_6)
go to L;               (P_7)
```



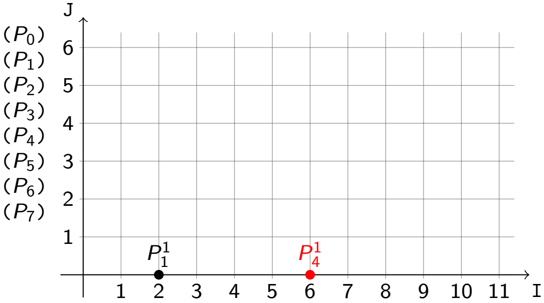
Example

```
I := 2, J := 0;         (P0)
L:                      (P1)
if ... then             (P2)
  I := I + 4            (P3)
else                    (P4)
  J := J + 1, I := I + 2;   (P5)
fi;                     (P6)
go to L;                (P7)
```



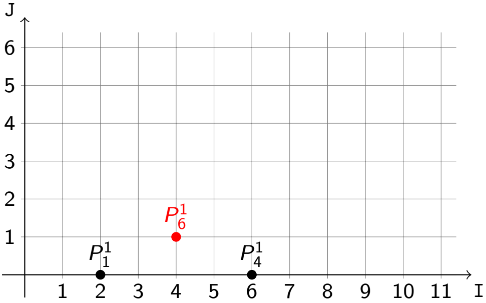
Example

```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```



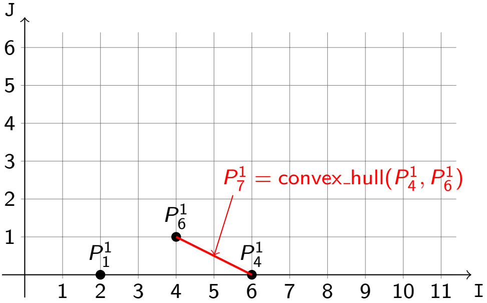
Example

```
                                      (P0)
I := 2, J := 0;                       (P1)
L:                                    (P2)
if ... then                           (P3)
  I := I + 4                          (P4)
else                                  (P5)
  J := J + 1, I := I + 2;             (P6)
fi;                                   (P7)
go to L;
```



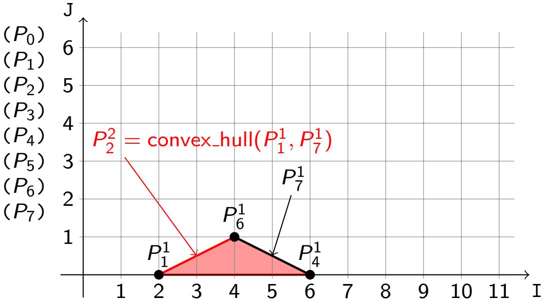
Example

```
                                      (P0)
I := 2, J := 0;                       (P1)
L:                                    (P2)
if ... then                           (P3)
  I := I + 4                          (P4)
else                                  (P5)
  J := J + 1, I := I + 2;             (P6)
fi;                                   (P7)
go to L;
```



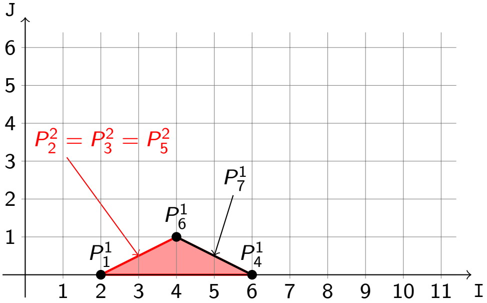
Example

```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```



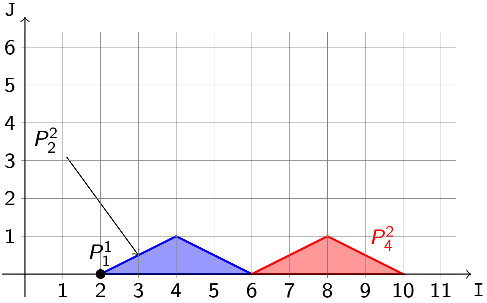
Example

```
                                          (P0)
I := 2, J := 0;                           (P1)
L:                                        (P2)
if ... then                               (P3)
  I := I + 4                              (P4)
else                                      (P5)
  J := J + 1, I := I + 2;                 (P6)
fi;                                       (P7)
go to L;
```



Example

```
                                      (P0)
I := 2, J := 0;                       (P1)
L:                                    (P2)
if ... then                           (P3)
  I := I + 4                          (P4)
else                                  (P5)
  J := J + 1, I := I + 2;             (P6)
fi;                                   (P7)
go to L;
```



Example

```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```

(P_0)

(P_1)

(P_2)

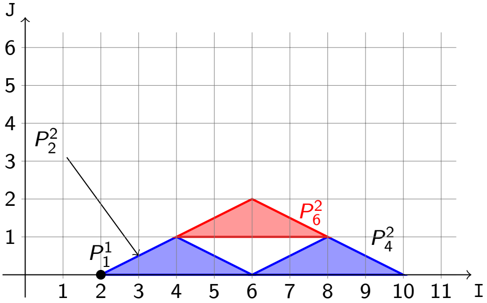
(P_3)

(P_4)

(P_5)

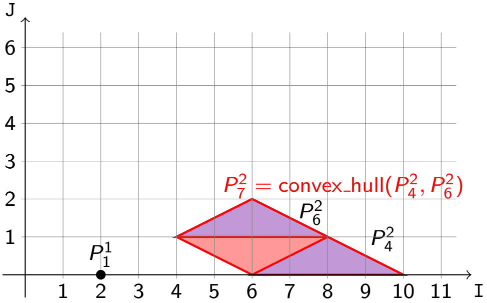
(P_6)

(P_7)



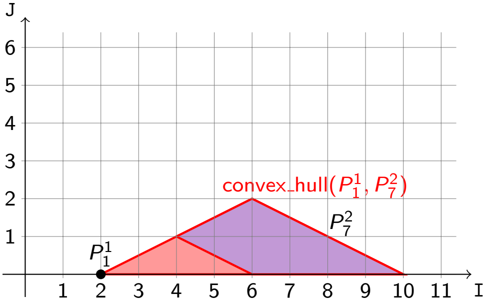
Example

```
                                  (P0)
I := 2, J := 0;                   (P1)
L:                                (P2)
if ... then                       (P3)
  I := I + 4                      (P4)
else                              (P5)
  J := J + 1, I := I + 2;         (P6)
fi;                               (P7)
go to L;
```



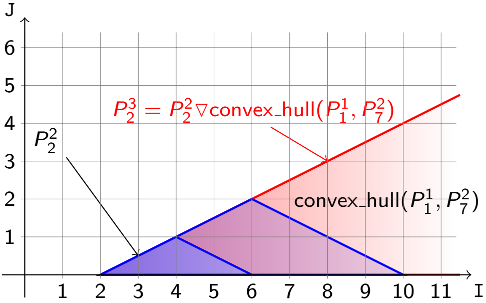
Example

```
                                      (P0)
I := 2, J := 0;                       (P1)
L:                                    (P2)
if ... then                           (P3)
  I := I + 4                          (P4)
else                                  (P5)
  J := J + 1, I := I + 2;             (P6)
fi;                                   (P7)
go to L;
```



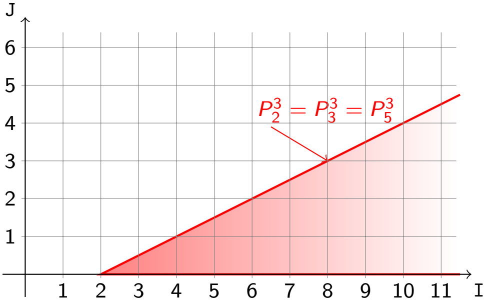
Example

```
                                      (P0)
I := 2, J := 0;                       (P1)
L:                                    (P2)
if ... then                           (P3)
  I := I + 4                          (P4)
else                                  (P5)
  J := J + 1, I := I + 2;             (P6)
fi;                                   (P7)
go to L;
```



Example

```
                                   (P0)
I := 2, J := 0;                    (P1)
L:                                 (P2)
if ... then                        (P3)
  I := I + 4                       (P4)
else                               (P5)
  J := J + 1, I := I + 2;          (P6)
fi;                                (P7)
go to L;
```



Example

```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```

(P_0)

(P_1)

(P_2)

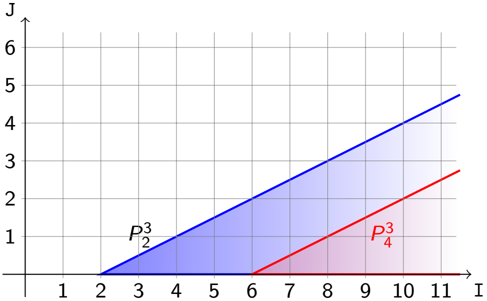
(P_3)

(P_4)

(P_5)

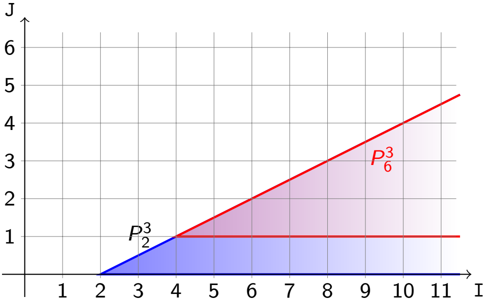
(P_6)

(P_7)



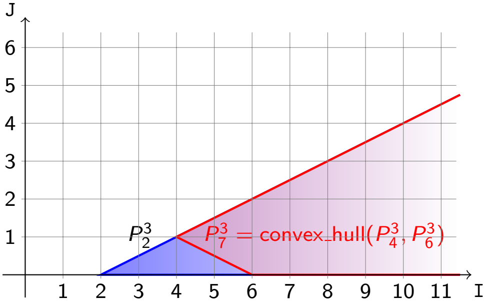
Example

```
I := 2, J := 0;          (P0)
L:                       (P1)
if ... then              (P2)
  I := I + 4             (P3)
else                     (P4)
  J := J + 1, I := I + 2; (P5)
fi;                      (P6)
go to L;                 (P7)
```



Example

```
                                   (P0)
I := 2, J := 0;                    (P1)
L:                                 (P2)
if ... then                        (P3)
  I := I + 4                       (P4)
else                               (P5)
  J := J + 1, I := I + 2;          (P6)
fi;                                (P7)
go to L;
```



Tools

- APRON: numerical abstract domain library
  - `http://apron.cri.ensmp.fr/library/`