Static Analysis and Verification

SAV 2025/2026

Ondřej Lengál, Tomáš Vojnar

`{lengal,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 minimizing 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 searching, type analysis, data flow analysis, . . . , abstract interpretation, (and sometimes even model checking).

Static Analyses

- Efficiency (effectiveness) of an analysis often crucially depends on the abstraction used.
- An analysis successful for one class of programs/properties to be checked may (and is very likely to) fail for a different one: too much imprecision, inefficiency, divergence.
- Analyses are tailored for specific classes of programs and their properties of interest.
    - This implies a need to prove soundness (completeness) of each analysis.

Abstract Interpretation

- Introduced by Patrick and Radhia Cousot at POPL'77.
- A general framework for static analyses.
- Particular analyses are created by providing specific components (abstract domain, abstract transformers, ...) to the framework.
- Abstract interpretation assigns a program an abstract semantics over an abstract domain and analyses this abstract semantics (cf. predicate abstraction).
- When certain properties of the components are met (wrt. the concrete semantics), the analysis is guaranteed to be sound.

Ingredients of Abstract Interpretation

- Abstract domain
  - a set of abstract contexts,
  - an abstract context represents a set of program states (typically used to represent a set of program states reachable at some program location).
- Abstract transformers
  - for each program operation there is a corresponding transformer that represents the effect of the operation performed on an abstract context.
- Join operator
  - combines abstract contexts from several branches into a single one.
- Widening
  - performed on a sequence of abstract contexts appearing at a given location to accelerate obtaining a fixpoint.
- Narrowing
  - may be used to refine the result of widening.

- (Product operators, e.g., reduced product: combining abstract domains.)

Abstract Interpretation — formally

- Abstract interpretation I of a program P with the instruction set `Instr` is a tuple

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

where

  - Q is the abstract domain (domain of abstract contexts),
  - $\sqcup : Q \times Q \to Q$ is the join operator for accumulation of abstract contexts,
  - $\sqsubseteq \subseteq Q \times Q$ is an ordering defined as $x \sqsubseteq y \iff x \sqcup y = y$ where

    $(Q, \sqsubseteq)$ is a complete lattice,
  - $\top \in Q$ is the supremum of $(Q, \sqsubseteq)$,
  - $\bot \in Q$ is the (single) infimum of $(Q, \sqsubseteq)$,
  - $\tau : \texttt{Instr} \times Q \to Q$ defines the abstract transformers for particular instructions, required to be monotone on Q for each instruction from `Instr`.

- The soundness of abstract interpretation may be guaranteed using Galois connections.

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 $\forall q \in Q$:

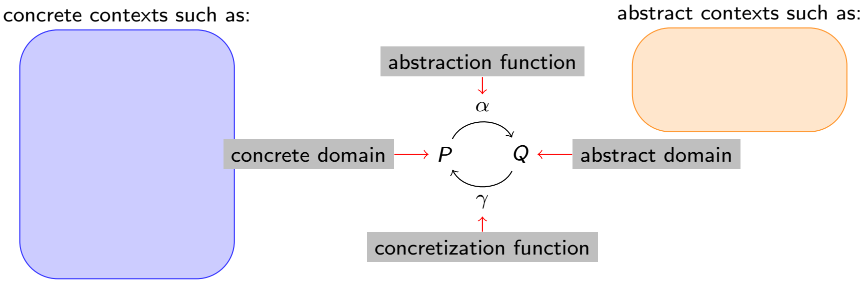
$$p \leq \gamma(q) \iff \alpha(p) \sqsubseteq q$$

Galois Connections

- 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, Q is the *abstract domain* and P a (more) *concrete domain* – elements of both domains, called abstract/concrete *contexts*, represent sets of states:



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),
  - $\alpha : P \rightarrow Q$ and $\gamma : Q \rightarrow P$ are functions such that $\forall p \in P$ and $\forall q \in Q$:

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

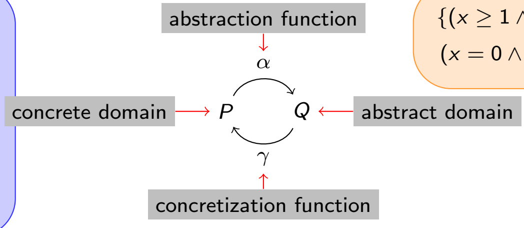
- In abstract interpretation, Q is the *abstract domain* and P a (more) *concrete domain* – elements of both domains, called abstract/concrete *contexts*, represent sets of states:

concrete contexts such as:

$$\begin{aligned}\{\{x \mapsto 1, y \mapsto 0\},\\ \{x \mapsto 2, y \mapsto 0\},\\ \{x \mapsto 3, y \mapsto 0\},\\ \{x \mapsto 0, y \mapsto 1\},\\ \{x \mapsto 0, y \mapsto 2\},\\ \{x \mapsto 0, y \mapsto 3\}\}\end{aligned}$$

abstract contexts such as:

$$\begin{aligned}\{(x \geq 1 \wedge y = 0),\\ (x = 0 \wedge y \geq 1)\}\end{aligned}$$

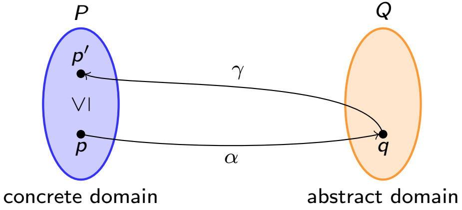


Galois Connections

- **Implication**: if the abstraction and concretization functions of an abstract interpretation form a Galois connection, $\forall p \in P.\ p \leq \gamma(\alpha(p))$.

Proof.

Take any $p \in P$, and let $q = \alpha(p)$.
As $q = \alpha(p) \Rightarrow \alpha(p) \sqsubseteq q$, the definition of Galois connections implies $p \leq \gamma(q)$.
By plugging in the fact that $q = \alpha(p)$, we get $p \leq \gamma(\alpha(p))$. □



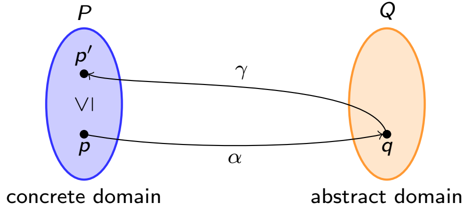
Galois Connections

- **Implication**: if the abstraction and concretization functions of an abstract interpretation form a Galois connection, $\forall p \in P.\ p \leq \gamma(\alpha(p))$.

Proof.

Take any $p \in P$, and let $q = \alpha(p)$.
As $q = \alpha(p) \Rightarrow \alpha(p) \sqsubseteq q$, the definition of Galois connections implies $p \leq \gamma(q)$.
By plugging in the fact that $q = \alpha(p)$, we get $p \leq \gamma(\alpha(p))$.

□



- If each instruction i from `Instr` and the corresponding abstract transformer τ_i respect the Galois connection, i.e., for each $p \in P$,

$$\alpha(i(p)) \sqsubseteq \tau_i(\alpha(p)),$$

the abstract interpretation may only over-approximate the concrete semantics.
Hence, it is sound.

Fixpoint Approximation

- The analysis of a program through an abstract interpretation may be viewed as finding the least/greatest fixpoint of the equation $\overline{A} = \overline{\tau}(\overline{A})$ where $\overline{A}$ is a vector of abstract contexts (one per program location) and $\overline{\tau}$ is an extension of τ to the entire program.
- By Knaster-Tarski, these fixpoints exist.
- In some cases (e.g., given a program with loops and using an infinite domain), 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 an over-approximation of a fixpoint,
  - narrowing: refines an 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 $I = (Q, \sqcup, \sqsubseteq, \top, \bot, \tau)$ be an abstract interpretation of a program.
- The binary widening operation $\triangledown$ is defined as:
  - $\triangledown : Q \times Q \to Q$,
  - $\forall C, D \in Q : (C \sqcup D) \sqsubseteq (C \triangledown D)$,
  - for all increasing infinite sequences $C_0 \sqsubseteq C_1 \sqsubseteq \ldots \sqsubseteq C_n \sqsubseteq \ldots$, it holds that the infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ defined recursively as

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

    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 $I = (Q, \sqcup, \sqsubseteq, \top, \bot, \tau)$ be an abstract interpretation of a program.
- The binary narrowing operation $\vartriangle$ is defined as:
  - $\vartriangle: Q \times Q \to Q$,
  - $\forall C, D \in Q : C \sqsupseteq D \Rightarrow (C \sqsupseteq (C \vartriangle D) \sqsupseteq D)$,
  - for all decreasing infinite sequences $C_0 \sqsupseteq C_1 \sqsupseteq \ldots \sqsupseteq C_n \sqsupseteq \ldots$, it holds that the infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ defined recursively as

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

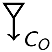
    is not strictly decreasing (and because the result of $C \vartriangle D$ is a lower bound of C, the sequence eventually stabilizes).
- Narrowing is performed only once a fixpoint is reached via widening.

Representation of a Program

- We choose (deterministic) finite flowcharts as a language independent representation of programs.
- A 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.

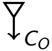
Representation of a Program

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

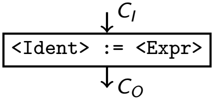


Representation of a Program

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

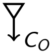


- Assignment: denotes the assignment A of expression `<Expr>` to the variable `<Ident>`.
  $C_O = \tau(A, C_I)$.

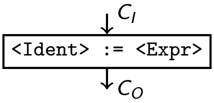


Representation of a Program

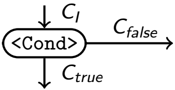
- Entry: denotes the entry point of a program. $C_O = \top$.



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

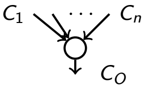


- Test: denotes splitting of the flow to branches B_{true} and B_{false} according to the Boolean condition <Cond>. Two contexts are computed: $C_{true} = \tau(B_{true}, C_I)$ and $C_{false} = \tau(B_{false}, C_I)$.



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).
  $C_O = \tau(J, C_1 \sqcup \cdots \sqcup C_n)$.



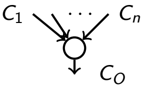
It often holds for junctions that:

- $\tau(J) = \lambda x \,.\, x$ — 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).
  $C_O = \tau(J, C_1 \sqcup \cdots \sqcup C_n)$.

  It often holds for junctions that:

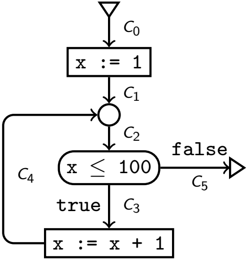
  - $\tau(J) = \lambda x \,.\, x$ — for simple junctions (`if` branches),
  - $\tau(J) = \lambda x \,.\, C_p \nabla x$ — for loop junctions,
  - $\tau(J) = \lambda x \,.\, C_p \vartriangle x$ — for loop junctions (only after widening)

  where C_p is the abstract context computed for the node in the previous iteration.
- Exit: denotes the exit point of a program.

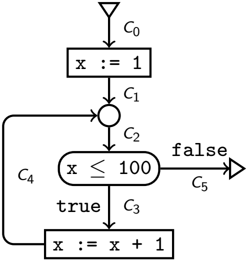
Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



Program Example

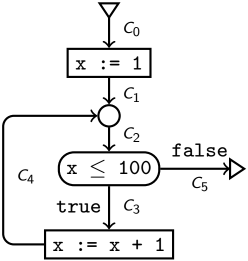
- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \triangledown [k, l] = [$if $k < i$ then $-\infty$ else i, if $l > j$ then $+\infty$ else $j]$.

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



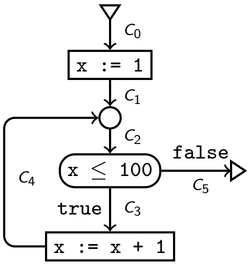
- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \triangledown [k, l] = [$if $k < i$ then $-\infty$ else i, if $l > j$ then $+\infty$ else $j]$.

Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

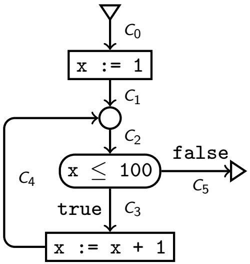
Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$.

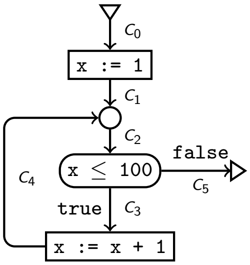
- $C_0^0 = [\ ,\ ]$ $C_0^1 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$
- $C_3^0 = [\ ,\ ]$
- $C_4^0 = [\ ,\ ]$
- $C_5^0 = [\ ,\ ]$

Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$.

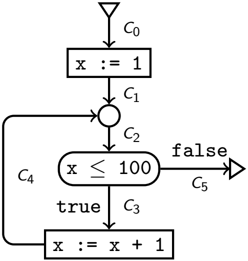
Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
    - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
    - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$.

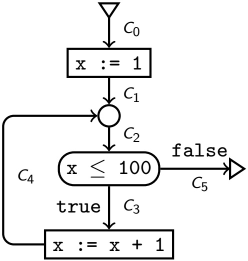
- $C_0^0 = [\ ,\ ]$ $C_0^1 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$
- $C_4^0 = [\ ,\ ]$
- $C_5^0 = [\ ,\ ]$

Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening ∇ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \nabla [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

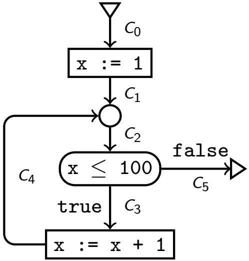
Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

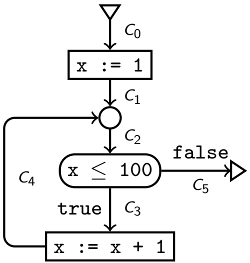
- $C_0^0 = [\ ,\ ]$ $C_0^1 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$ $C_2^2 = [1, +\infty]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$
- $C_4^0 = [\ ,\ ]$ $C_4^1 = [2, 2]$
- $C_5^0 = [\ ,\ ]$

Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

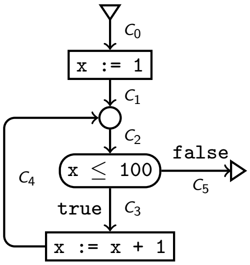
Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

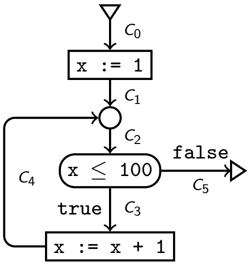
Equations that describe the flow of the analysis; their fix-point solution represents results of the analysis:

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

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- We will use notation $[a, b]$ for the predicate $a \leq x \leq b$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$.
- Assignments: *interval arithmetic* (e.g., $[i, j] + [k, l] = [i + k, j + l]$).
- Join: $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Tests: *intersections* of intervals.
- We define the widening $\triangledown$ of intervals as:
  - $[\ ,\ ] = glb(\mathbb{I})$, needs special handling (skipped)
  - $[i, j] \triangledown [k, l] = [\text{if } k < i \text{ then } -\infty \text{ else } i,$ $\text{if } l > j \text{ then } +\infty \text{ else } j]$.

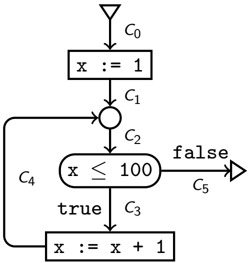
Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- Let us now define the narrowing operation $\triangle$ over intervals as
  - $[i, j] \triangle [k, l] = [$
    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.

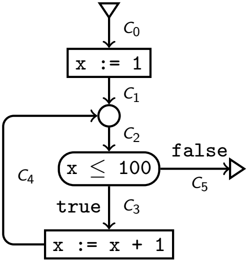
- $C_0^0 = [\ ,\ ]$ $C_0^1 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$ $C_2^2 = [1, +\infty]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$ $C_3^2 = [1, 100]$
- $C_4^0 = [\ ,\ ]$ $C_4^1 = [2, 2]$ $C_4^2 = [2, 101]$
- $C_5^0 = [\ ,\ ]$ $C_5^1 = [101, +\infty]$

Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- Let us now define the narrowing operation $\triangle$ over intervals as
  - $[i, j] \triangle [k, l] = [$
    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.

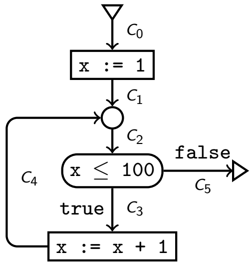
- $C_0^0 = [\ ,\ ]$ $C_0^1 = [-\infty, +\infty]$
- $C_1^0 = [\ ,\ ]$ $C_1^1 = [1, 1]$
- $C_2^0 = [\ ,\ ]$ $C_2^1 = [1, 1]$ $C_2^2 = [1, +\infty]$ $C_2^3 = [1, 101]$
- $C_3^0 = [\ ,\ ]$ $C_3^1 = [1, 1]$ $C_3^2 = [1, 100]$
- $C_4^0 = [\ ,\ ]$ $C_4^1 = [2, 2]$ $C_4^2 = [2, 101]$
- $C_5^0 = [\ ,\ ]$ $C_5^1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

Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

Program Example

- Consider the below flowchart program and analysis with the interval abstract domain $\mathbb{I}$:



- Let us now define the narrowing operation $\triangle$ over intervals as
  - $[i, j] \triangle [k, l] = [$
    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.

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

Equations that describe the flow of the analysis; their fixpoint solution represents results of the analysis:

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

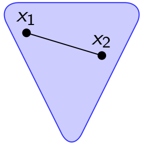
From a Practical Point of View

- Concepts like Galois connections and even the abstraction and concretisation functions typically stay in the theory only (not really implemented): may be used to prove correctness of the analysis.
- One typically needs to implement the following:
    - the abstract domain,
    - the ordering relation,
    - join,
    - widening,
    - abstract transformers,
    - fixpoint loop, input, output, translation to some intermediate format, ...
      – unless provided by some framework.

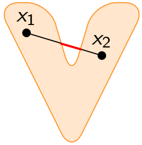
Polyhedral Analysis

Polyhedral Analysis

- An abstract interpretation-based approach of automatic discovery of relations among numerical program variables expressible as linear inequations,
    - this can be seen as a generalization of the interval analysis.
- Let $\vec{x} = x_1, \ldots, x_n \in \mathbb{R}^n$ be the variables of a program. We can use a convex polyhedron (convex polytope) $P \subseteq \mathbb{R}^n$ to represent a set of 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



non-convex set

- Still usually quite expensive, so often replaced by cheaper domains: signs, intervals, DBMs, octagons (mentioned later on).

Representation of a Convex Polyhedron

- We use 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 (with some overhead).
- 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

- A convex polyhedron P can also be characterized by its frame $F = (V, R, L)$:
  - Vertices V: points $\vec{v}$ of a polyhedron P that 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) that are not positive combinations of other rays $\vec{s_1}, \ldots, \vec{s_p}$ of P:

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

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) that 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) that 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:

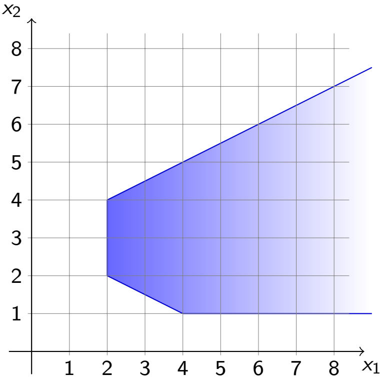
$$\forall \vec{x} \in P, \forall \mu \in \mathbb{R} : \vec{x} + \mu \vec{l} \in P.$$

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

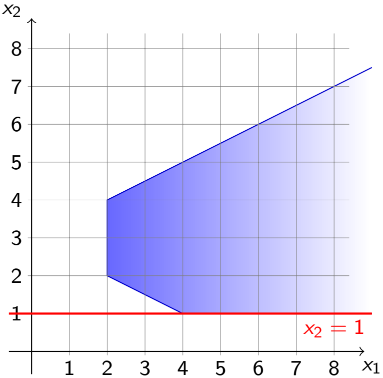
$$\vec{x} = \sum_{i=1}^{\sigma} \lambda_i \vec{v_i} + \sum_{j=1}^{\rho} \mu_j \vec{r_j} + \sum_{k=1}^{\delta} \nu_k \vec{l_k}$$

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

Example of a Convex Polyhedron (Polygon)



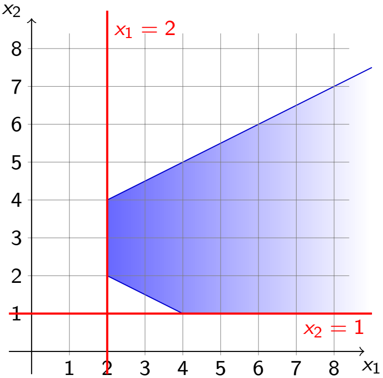
Example of a Convex Polyhedron (Polygon)



System of linear inequations

$$x_2 \geq 1$$

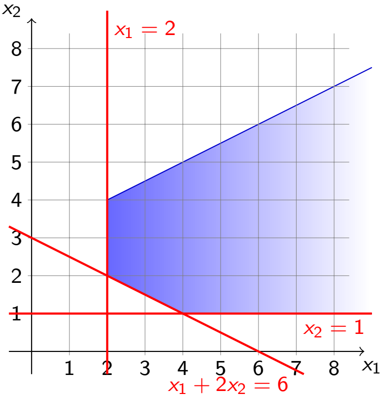
Example of a Convex Polyhedron (Polygon)



System of linear inequations

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

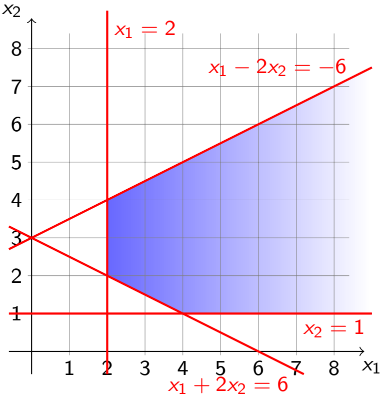
Example of a Convex Polyhedron (Polygon)



System of linear inequations

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

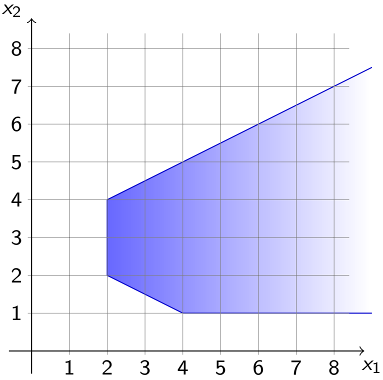
Example of a Convex Polyhedron (Polygon)



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 (Polygon)



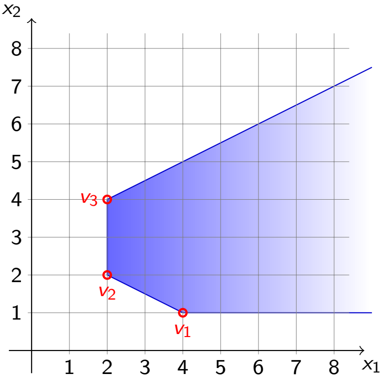
System of linear inequations

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

Frame of a polyhedron

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

Example of a Convex Polyhedron (Polygon)



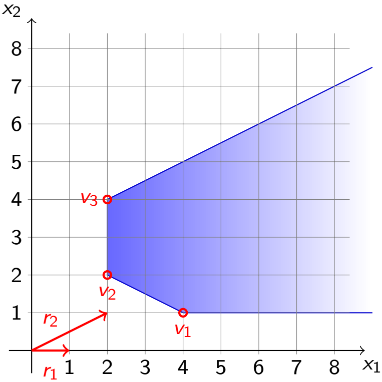
System of linear inequations

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

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 (Polygon)



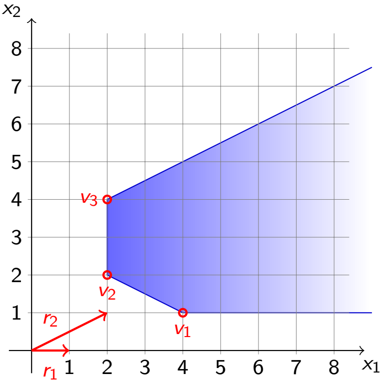
System of linear inequations

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

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 (Polygon)



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 constraints 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 constraint 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 constraint 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 $v_{jm}' = v_{jm}$ where $\forall 1 \leq m \leq \sigma, m \neq i$.

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 constraint 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 $v_{jm}' = v_{jm}$ where $\forall 1 \leq m \leq \sigma, m \neq i$.
    - $R' = \{\vec{r_1'}, \ldots, \vec{r_\rho'}\}$ where $\vec{r_j'}$ is defined by $r_{ji}' = \vec{a}\vec{r_j}$ and $r_{jm}' = r_{jm}$ for $\forall 1 \leq m \leq \rho, m \neq i$.

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 constraint 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 $v'_{jm} = v_{jm}$ where $\forall 1 \leq m \leq \sigma, m \neq i$.
    - $R' = \{\vec{r_1'}, \ldots, \vec{r_\rho'}\}$ where $\vec{r_j'}$ is defined by $r'_{ji} = \vec{a}\vec{r_j}$ and $r'_{jm} = r_{jm}$ for $\forall 1 \leq m \leq \rho, m \neq i$.
    - $L' = \{\vec{l_1'}, \ldots, \vec{l_\delta'}\}$ where $\vec{l_j'}$ is defined by $l'_{ji} = \vec{a}\vec{l_j}$ and $l'_{jm} = l_{jm}$ for $\forall 1 \leq m \leq \delta, m \neq i$.

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 a Boolean condition C, it needs to hold that $P_t \supseteq P \cap T_C, P_f \supseteq P \setminus T_C$ where T_C is the subset of $\mathbb{R}^n$ such that each point of T_C satisfies C (the right sides of the inclusions are not necessarily 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 a Boolean condition C, it needs to hold that $P_t \supseteq P \cap T_C, P_f \supseteq P \setminus T_C$ where T_C is the subset of $\mathbb{R}^n$ such that each point of T_C satisfies C (the right sides of the inclusions are not necessarily 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 a Boolean condition C, it needs to hold that $P_t \supseteq P \cap T_C, P_f \supseteq P \setminus T_C$ where T_C is the subset of $\mathbb{R}^n$ such that each point of T_C satisfies C (the right sides of the inclusions are not necessarily 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 a Boolean condition C, it needs to hold that $P_t \supseteq P \cap T_C, P_f \supseteq P \setminus T_C$ where T_C is the subset of $\mathbb{R}^n$ such that each point of T_C satisfies C (the right sides of the inclusions are not necessarily 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

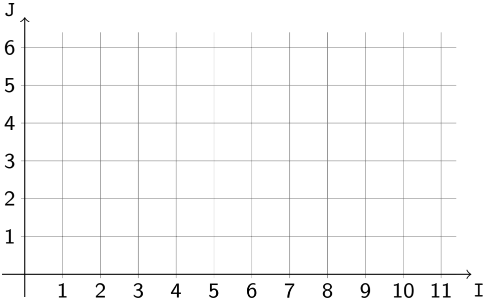
- Junctions correspond to merge of several program paths so the output polyhedron P is union of all input polyhedra P_i. It is computed according to the kind of the junction:
    - Simple junctions: for input polyhedra $P_1, \ldots, P_m$, we compute the convex hull of $P_1 \cup \cdots \cup P_m$.

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 $P_1, \ldots, P_m$, we compute the convex hull of $P_1 \cup \cdots \cup P_m$.
  - Loop junctions: for input polyhedra $P_1, \ldots, P_m$, let Q be the convex hull of $P_1 \cup \cdots \cup P_m$. Then $P' = P \triangledown Q$ is the convex polyhedron consisting of linear constraints of P satisfied by every element of Q.

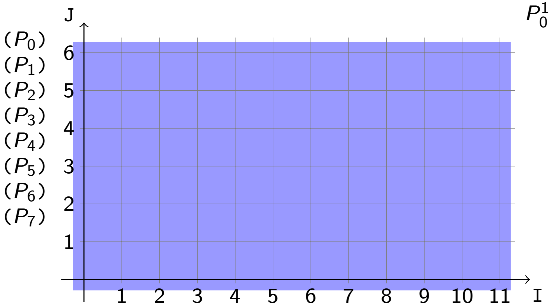
Example

```
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)
```



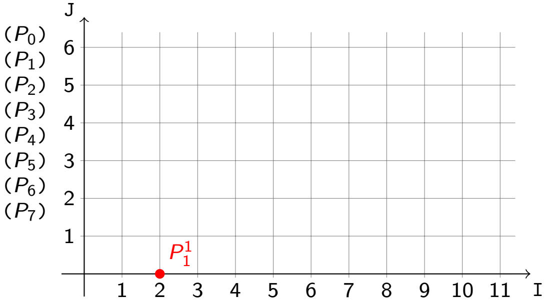
Example

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



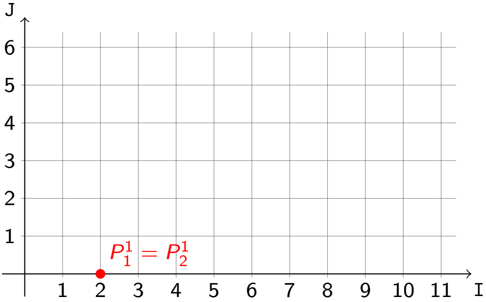
Example

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



Example

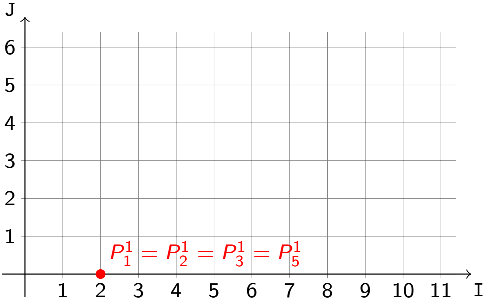
```
I := 2, J := 0;           (P0)
L:                        (P1)
if ... then               (P2)
  I := I + 4              (P3)
else                      (P4)
  J := J + 1, I := I + 2; (P5)
fi;                       (P6)
go to L;                  (P7)
```



Example

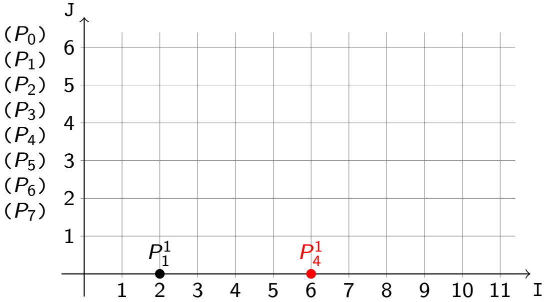
```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```

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



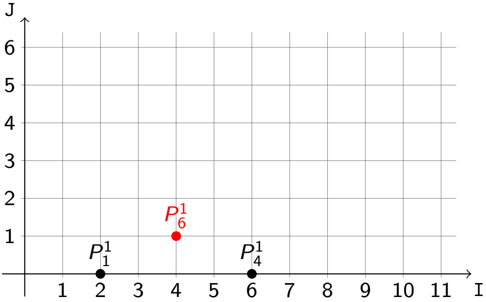
Example

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



Example

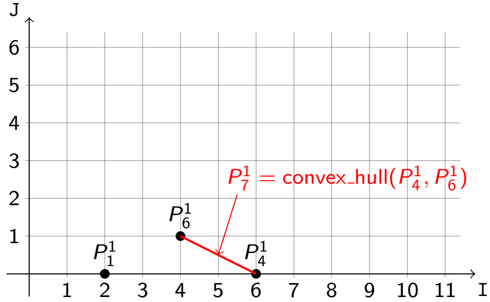
```
I := 2, J := 0;       (P0)
L:                    (P1)
if ... then           (P2)
  I := I + 4          (P3)
else                  (P4)
  J := J + 1, I := I + 2;   (P5)
fi;                   (P6)
go to L;              (P7)
```



Example

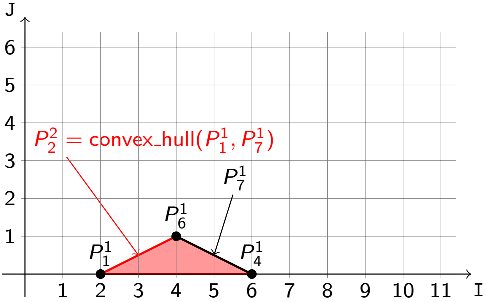
```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```

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



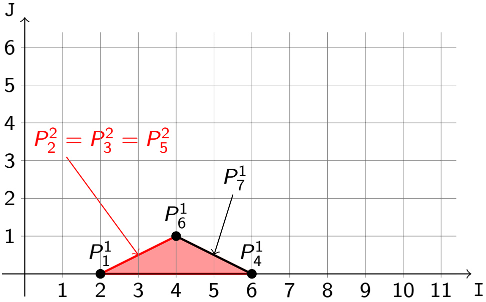
Example

```
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)
```



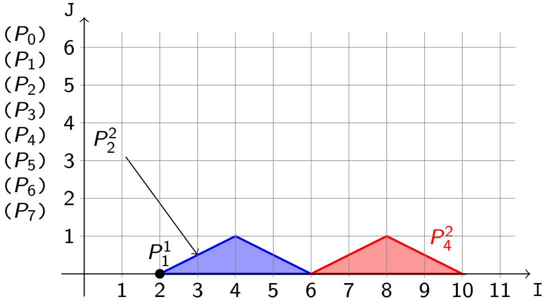
Example

```
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)
```



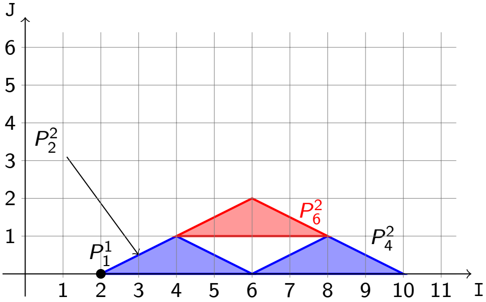
Example

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



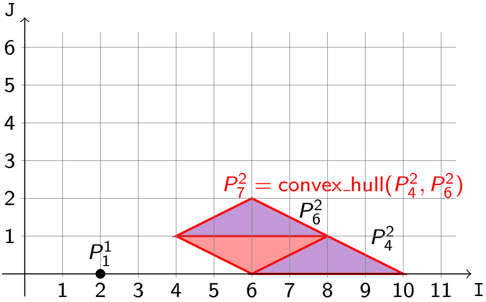
Example

```
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

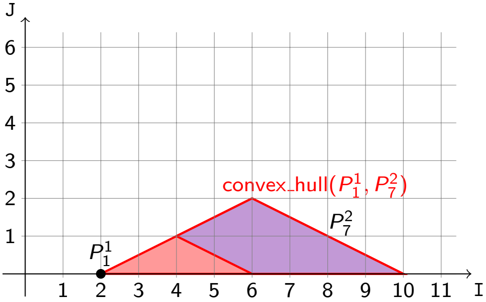
```
I := 2, J := 0;      (P0)
L:                   (P1)
if ... then          (P2)
  I := I + 4         (P3)
else                 (P4)
  J := J + 1, I := I + 2;   (P5)
fi;                  (P6)
go to L;             (P7)
```



Example

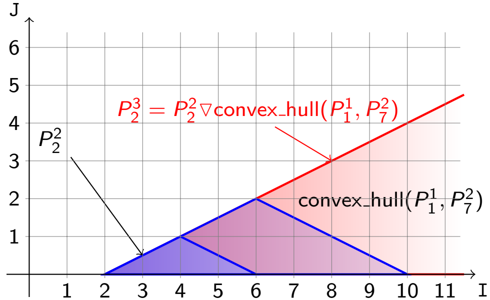
```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```

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



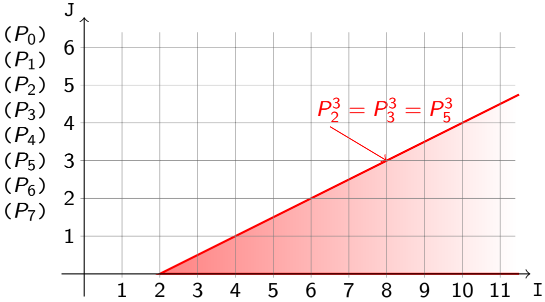
Example

```
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)
```



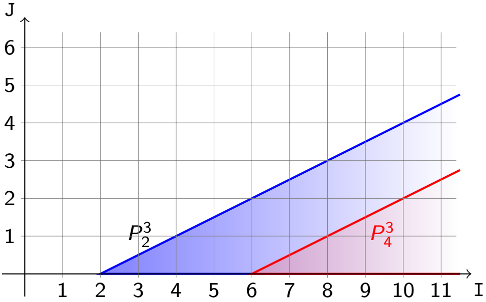
Example

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



Example

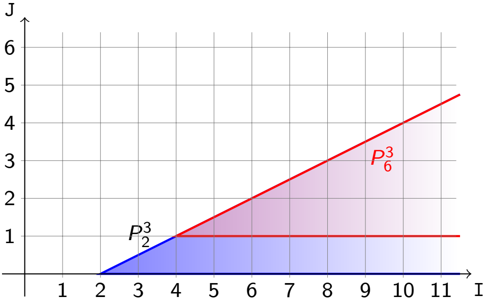
```
I := 2, J := 0;            (P0)
L:                         (P1)
if ... then                (P2)
  I := I + 4               (P3)
else                       (P4)
  J := J + 1, I := I + 2;  (P5)
fi;                        (P6)
go to L;                   (P7)
```



Example

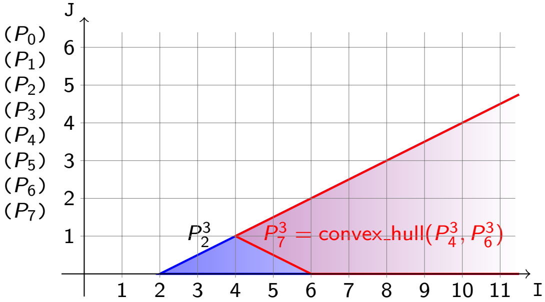
```
I := 2, J := 0;
L:
if ... then
  I := I + 4
else
  J := J + 1, I := I + 2;
fi;
go to L;
```

(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;
```



Various Known Domains

Numerical Domains

Typically, conjunctions of constraints of some form for $a, b, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$:

- Signs: $x\ [\leq \mid < \mid = \mid > \mid \geq]\ 0$, $x = [\perp|\top]$.
- Intervals: $a \leq x \leq b$.
- Difference Bound Matrices: $x - y \leq c$, $[+|-]x \leq c$.
- Octagons: $[+|-]x\ [+|-]y \leq c$, $[+|-]x \leq c$.
- Polyhedra: $\vec{a}\vec{x} \leq b$.
- Simple Congruences: $x = a\ mod\ b$.
- Linear Congruences: $\vec{a}\vec{x} = b\ mod\ c$.
- ...

Facebook/Meta Infer bounds analysis – for buffer sizes and values of indices, intervals with the following kinds of bounds are used:

$[+|-]\infty$, linear expressions, $a[+|-][min|max](b, x)$.

Improving the Precision

The precision of numerical domains can be improved by various kinds of partitioning:

- Characterizing abstract contexts reachable at each program line separately (used already in the example with interval analysis).
- Tracking values up to some preconfigured bound precisely.
- Cutting concrete domains of some variables into some preconfigured sets of regions (e.g., $(-\infty, a_1)$, $[a_1, a_2)$, $[a_2, a_3)$, ..., $[a_n, +\infty)$ for some $n > 1$ and $a_i < a_{i+1}$ for $1 \leq i \leq n-1$) and then widening within the regions only.

Shape Analysis

– Shape analysis: analysis of shapes of dynamic data structures (not just a may/must point-to analysis).

– Quite complex: characterising infinite sets of graphs!

– Many different domains: one of them is separation logic.

– A fragment of separation logic with a list predicate:

- Pure part – Boolean combinations of $x = [y|NULL]$.
- Spatial part – shape predicates:
  - $x \mapsto [y|NULL]$ – points-to predicate.
  - $ls(x, [y|NULL])$ – non-empty, singly-linked list predicate.
- Shape predicates joined by the separating conjunction "$*$". Conjoined heaps must allocate different memory cells, e.g.:
  - $x \mapsto y * x \mapsto z$ is false,
  - while $x \mapsto y * ls(z, y) * u \mapsto v$ is satisfiable.

Shape Analysis – Continued

– The pure and spatial parts are conjoined.

– The variables are either program variables or existentially quantified logical variables (denoting anonymous memory cells).

– Separation allows for modular analysis.

– Computing sets of formulae per location (a top-level disjunction).

– Abstract transformers for pointer manipulating statements based on (partial) concretisation:

- $ls(x, y) \equiv x \mapsto y \vee \exists z : x \mapsto z * ls(z, y)$.
- Then manipulate $x \mapsto y$ or $x \mapsto z$.

– Join: union of sets of formulae, perhaps with pruning those covered by other (weaker) formulae (e.g., $x \mapsto y \sqsubseteq ls(x, y)$).

– Widening: based on abstracting sequences of points-to and list segments through logical variables to a single list segment:

- $\alpha(\exists y, z : x \mapsto y * ls(y, z) * z \mapsto NULL) \equiv ls(x, NULL)$.

Abstract Regular Model Checking

- Regular model checking (RMC): system configurations $\sim$ words, sets of configurations $\sim$ regular languages (finite automata).
- Transitions: finite transducers or special operations on automata.
- Abstract RMC – can be viewed as abstract interpretation with widening that abstracts automata by collapsing some states:
  - Overapproximation wrt. language inclusion.
  - States to collapse: same languages of words up to some bound or intersecting the same predicate languages.
  - Abstraction is automatically refineable using a CEGAR loop.
- Generalisations: trees and nested forests with leaf-to-root references – (tuples of (nested)) tree automata.
- Applications: parametric protocols (mutual exclusion, ...), recursion, communication queues, strings, heap structures (lists, trees, ...), microprocessor pipelining, ...

Interprocedural Analysis

Summaries

- Summaries: (sets of) pre-/post-condition pairs for functions.
- Record under which precondition a function can be executed leading to a given postcondition.
- Pre-/post-conditions encoded using abstract contexts recording relevant parts of the entire contexts: values of parameters, relevant global variables, accessible parts of heap, ..., returned/changed parts of contexts.
- Can be computed top-down:
  - If a function has never been called with the precondition, analyse it and record a new pair into its summary.
  - If a function has already been called under the same precondition, use directly the recorded result.
  - If the input context is covered ($\sqsubseteq$) by some recorded precondition: may use the postcondition for a loss of precision.

Summaries: Bottom-Up

– Summaries can also be computed bottom-up along the call-tree.

– Each function analysed without any known context: needs to derive under which precondition(s) a function can produce post-condition(s) meaningful for the given analysis.

– For example:

  - If analysing pointer safety and seeing a dereference without a check, derive that the pointer should be non-null.
  - If analysing locking and seeing a lock without a preceding unlock, derive that the pointer should be unlocked.

– Can be very scalable: each function analysed precisely once!

– Easier to lose information.

– Harder to design the analysis.