Program Analysis **November 15, 2015**

Lecture 4: Numeric Abstract Domains

Lecturer: Mooly Sagiv *Scribe: Omer Anson, Shelly Grossman*

4.1 Background

The research of numeric abstract domains is of interest and value in the field of static analysis due to its multiple uses in the analysis of programs (beyond simple numeric conclusions on simple programs), as well as the significant successes in developing precise abstractions.

Numeric abstract domains allow us to conclude on numeric invariants and numerical bounds in programs. In the basic setting there will be a set of n real variables, and with the assistance of numerical abstractions it is possible to conclude these n variables are bounded in some set $A \in R^n$.

Some practical examples that are feasible with the assistance of numerical abstract domains:

- Detection of invalid array accesses (Array Out-Of-Bounds exceptions)
- Detecting a program's termination
- Arriving at conditions for a program's termination
- Cost of a program: runtime, memory consumption
- C Pointers - can be understood as offsets, thus Integer analysis can be used. In languages such as Java, on the other hand, we will prefer other techniques such as *shape analysis*
- String analysis in C programs - understanding a buffer's length, detecting overflows
- Variables' relations and correlations

4.2 Numeric Semantics

The analysis is constructed of expressions and commands.

Definition 1. *Expressions are constructed by structural induction. An expression can be a variable, range constant, unary operation, or binary operation on other expressions. The grammar is presented in equations (4.1)-(4.4).*

$$\begin{aligned} \langle exp \rangle &= V \mid && (4.1)\\ &\quad -\langle exp \rangle \mid && (4.2)\\ &\quad \langle exp \rangle \circ \langle exp \rangle \mid && (4.3)\\ &\quad [c, c'] && (4.4) \end{aligned}$$

In equation (4.3), an operation is defined as one of $\circ \in \{+, -, \cdot, /\}$. In equation (4.4), the interval in this case deals with unknown information, such as user input.

Algorithm 1 Example programme

```
1. X <- [1,10];
2. Y <- 100;
while 3. X > 0:
        4. X <- X - 1
        5. Y <- Y + 10
6. End
```

Definition 2. *A command is an operation that sets a variable to an expression, or narrows the possible states under a condition. The assume command can be used in control flows. The grammar is defined in equations (4.5)-(4.7).*

$$\begin{aligned} \langle command \rangle &= V := \langle exp \rangle \mid && (4.5) \\ & \quad assume\ \langle exp \rangle \bowtie 0 \mid && (4.6) \\ & \quad assert\ \langle exp \rangle \bowtie 0 && (4.7) \end{aligned}$$

In equations (4.6) and (4.7), a relation is defined as one of $\bowtie \in \{=, <, >, \leq, \geq, \neq\}$. *The* assume *command can be seen as a projection. Assertion is used at the end of an application, to prove certain properties. It is not part of the analysis proper. Therefore, we will concentrate only on* assume.

This syntactic language is complete. Any programme in any language can be compiled into this language.

Definition 3. *We describe the programme as a Control Flow Graph,* $G(N, E, s)$, *where* N *is the set of states, used as graph vertices.* E *is the set of edges,* $E \subseteq N \times N$. *Each edge is anotated with a command,* $c(E)$. s *is the start node, i.e. the starting point in programme.*

4.2.1 A Program (Without Procedures)

Using definition 3, we support loops, but we do not support functions and function calls. Each programme has a unique control flow graph, with unique edges and unique commands over these edges.

Example 1. *Figure 1 shows an example programme. Note that this programme has 6 states, so* $N = [6]$. *The control flow graph can be seen in figure 4.1. The edges and the commands on the edges can also be seen there. Note also that state 3 starts after the* while *keyword, since it corresponds to the termination table. It tells the compiler or interpreter where to return to at the end of the block.*

4.3 Concrete Operational Semantics

We would like the syntax above to have meaning beyond the syntax itself. To this end, we define a concrete semantics. There are many different options for a concrete semantics, and they vary in precision. Let Var be the set of variables, and let $\mathbb{R}$ be the set of reals. We define a state $\sigma \in \Sigma$ as a function $\sigma : Var \to \mathbb{R}$. We assume that in each state, a variable can hold only one value. We then define the semantics $E[\![]\!] : \langle exp \rangle \to (\Sigma \to P(\mathbb{R}))$, where P is the powerset operation, as follows:

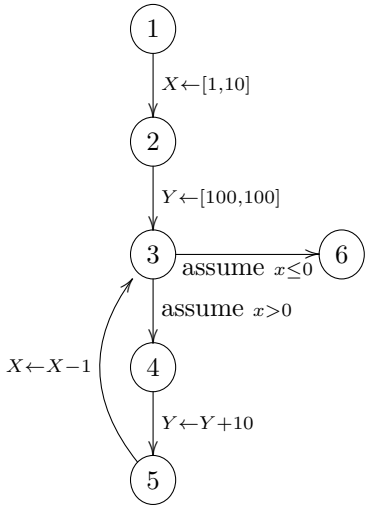


Figure 4.1: Control Flow Graph for the programme in algorithm 1

$$
\begin{aligned}
E[\![V]\!](\sigma) &= \{\sigma(V)\} && (4.8)\\
E[\![[c,c']]\!](\sigma) &= \{x | x \in [c,c']\} && (4.9)\\
E[\![-\langle exp\rangle]\!](\sigma) &= \{-x | x \in E[\![\langle exp\rangle]\!](\sigma)\} && (4.10)\\
E[\![\langle exp\rangle \circ \langle exp\rangle']\!](\sigma) &= \left\{x \circ x' | x \in E[\![\langle exp\rangle]\!](\sigma) \wedge x' \in E[\![\langle exp\rangle']\!](\sigma)\right\} \quad \circ \in \{+,-,\cdot\} && (4.11)\\
E[\![\langle exp\rangle / \langle exp\rangle']\!](\sigma) &= \left\{x/x' | x \in E[\![\langle exp\rangle]\!](\sigma) \wedge x' \in E[\![\langle exp\rangle']\!](\sigma) \wedge x' \neq 0\right\} && (4.12)
\end{aligned}
$$

There are several points worth mentioning of the above semantics. Firstly, it is defined inductively over the structure. It can be seen in equations (4.10–4.12) that the evaluated expression is constructed form its syntactical parts. Secondly, in equation (4.9), the result does not depend on the state σ, since the expression contains only constants, and not variables. Thirdly, division in equation (4.12) is partial, since it is not allowed to divide by 0. There is no need to treat undefined behaviour.

This semantics is not suitable for languages such as C, since we have not added support for side-effects, which change the state. Support for side-effects can be added, but would require higher complexity.

We now define the semantics of commands. Intuitively, a command maps between sets of states. Here, again, a state $\sigma \in \Sigma$ is a function $\sigma : Var \to \mathbb{R}$. $Z \subseteq \Sigma$ is an arbitrary set of states. We treat $\sigma[V \mapsto v]$ as the state identical to σ on all variables except V, where $\sigma[V \mapsto v](V) = v$. A leading requirement in this definition is support for non-determinism.

The semantics of commands, $C[\![]\!] : \langle command\rangle \to (P(\Sigma) \to P(\Sigma))$ is as follows

$$
\begin{aligned}
C[\![V := \langle exp\rangle]\!](Z) &= \{\sigma[V \mapsto x] | \sigma \in Z \wedge x \in E[\![\langle exp\rangle]\!](\sigma)\} && (4.13)\\
C[\![\text{assume } \langle exp\rangle \bowtie 0]\!](Z) &= \{\sigma | \sigma \in Z \wedge \forall x.x \in E[\![\langle exp\rangle]\!](\sigma) \implies x \bowtie 0\} && (4.14)\\
C[\![\text{assert } \langle exp\rangle \bowtie 0]\!](Z) &= C[\![\text{assume } \langle exp\rangle \bowtie 0]\!](Z) && (4.15)
\end{aligned}
$$

As we have said, assert happens when the analysis ends. We treat it like *assume*, but generate a message to the user if there is a state on which the condition does not hold. In other words, all states must hold under the asserted condition

Note that under different domains, there may be different behaviours. Treating this case is not trivial.

We insist that the semantics is not only monotone, but also distributive. In other words,

$$C\left[\!\left[\langle command\rangle\right]\!\right](Z) = \bigcup_{\sigma\in Z} C\left[\!\left[\langle command\rangle\right]\!\right](\{\sigma\}) \tag{4.16}$$

This means that if $Z \subseteq \Sigma$ is a set of states, running the command on all states at once as per the semantics, or running the command on each state separately and unifying the results, should give the same outcome. This allows us to run the semantics on a per-element basis.

4.3.1 Concrete Semantics of a Programme

Given the control flow graph of a program, $G(N, E, s)$, we treat $[\![G(N, E, s)]\!] : P(\Sigma) \times N \times P(\Sigma)$. Given the initial state of a programme, $s \in \Sigma$, $[\![G(N, E, s)]\!]$ returns all reachable states.

Given the set of initial state, s, we are looking for the minimal solution to the equations

$$CS_s = \iota \tag{4.17}$$

$$CS_n = \bigcup_{\langle m,n\rangle\in E} C\left[\!\left[c\left(\langle m, n\rangle\right)\right]\!\right](CS_m) \qquad n \neq s \tag{4.18}$$

In equation (4.17), we set the initial states. Afterwards, for every edge from every visited state, we add the states the are created by applying the command on the edge.

We define a lattice $D = \langle P(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma\rangle$. The iteration equation (4.18) is monotone in D. ι represents the possible states at the beginning of the programme.

Such a minimal solution exists, according to Tarski's theorem. However, such a solution is not computable, in general. The number of states, edges, and commands can become very large.

Example 2. *We want to review the behaviour of this analysis on the programme in example 1. In this case, the initial node is $s = 1$, let ι be an arbitrary state when the programme starts, and G can be trivially*

understood from the graph in Figure 4.1. Therefore,

$$CS_1 = \iota \tag{4.19}$$

$$CS_2 = \left\{ \begin{array}{l} [X \mapsto 1], [X \mapsto 2], [X \mapsto 3], [X \mapsto 4], [X \mapsto 5], \\ [X \mapsto 6], [X \mapsto 7], [X \mapsto 8], [X \mapsto 9], [X \mapsto 10] \end{array} \right\} \tag{4.20}$$

$$CS_3 = \left\{ \begin{array}{l} [X \mapsto 1, Y \mapsto 100], [X \mapsto 2, Y \mapsto 100], [X \mapsto 3, Y \mapsto 100], \\ [X \mapsto 4, Y \mapsto 100], [X \mapsto 5, Y \mapsto 100], [X \mapsto 6, Y \mapsto 100], \\ [X \mapsto 7, Y \mapsto 100], [X \mapsto 8, Y \mapsto 100], [X \mapsto 9, Y \mapsto 100], \\ [X \mapsto 10, Y \mapsto 100] \end{array} \right\} \tag{4.21}$$

$$CS_4 = CS_3 \tag{4.22}$$

$$CS_5 = \left\{ \begin{array}{l} [X \mapsto 1, Y \mapsto 110], [X \mapsto 2, Y \mapsto 110], [X \mapsto 3, Y \mapsto 110], \\ [X \mapsto 4, Y \mapsto 110], [X \mapsto 5, Y \mapsto 110], [X \mapsto 6, Y \mapsto 110], \\ [X \mapsto 7, Y \mapsto 110], [X \mapsto 8, Y \mapsto 110], [X \mapsto 9, Y \mapsto 110], \\ [X \mapsto 10, Y \mapsto 110] \end{array} \right\} \tag{4.23}$$

$$CS_3 = \left\{ \begin{array}{l} [X \mapsto 1, Y \mapsto 100], [X \mapsto 2, Y \mapsto 100], [X \mapsto 3, Y \mapsto 100], \\ [X \mapsto 4, Y \mapsto 100], [X \mapsto 5, Y \mapsto 100], [X \mapsto 6, Y \mapsto 100], \\ [X \mapsto 7, Y \mapsto 100], [X \mapsto 8, Y \mapsto 100], [X \mapsto 9, Y \mapsto 100], \\ [X \mapsto 10, Y \mapsto 100], \\ [X \mapsto 0, Y \mapsto 110], [X \mapsto 1, Y \mapsto 110], [X \mapsto 2, Y \mapsto 110], \\ [X \mapsto 3, Y \mapsto 110], [X \mapsto 4, Y \mapsto 110], [X \mapsto 5, Y \mapsto 110], \\ [X \mapsto 6, Y \mapsto 110], [X \mapsto 7, Y \mapsto 110], [X \mapsto 8, Y \mapsto 110], \\ [X \mapsto 9, Y \mapsto 110] \end{array} \right\} \tag{4.24}$$

$$CS_6 = \{[X \mapsto 0, Y \mapsto 110]\} \tag{4.25}$$

$$CS_4 = \left\{ \begin{array}{l} [X \mapsto 1, Y \mapsto 100], [X \mapsto 2, Y \mapsto 100], [X \mapsto 3, Y \mapsto 100], \\ [X \mapsto 4, Y \mapsto 100], [X \mapsto 5, Y \mapsto 100], [X \mapsto 6, Y \mapsto 100], \\ [X \mapsto 7, Y \mapsto 100], [X \mapsto 8, Y \mapsto 100], [X \mapsto 9, Y \mapsto 100], \\ [X \mapsto 10, Y \mapsto 100], \\ [X \mapsto 1, Y \mapsto 110], [X \mapsto 2, Y \mapsto 110], [X \mapsto 3, Y \mapsto 110], \\ [X \mapsto 4, Y \mapsto 110], [X \mapsto 5, Y \mapsto 110], [X \mapsto 6, Y \mapsto 110], \\ [X \mapsto 7, Y \mapsto 110], [X \mapsto 8, Y \mapsto 110], [X \mapsto 9, Y \mapsto 110] \end{array} \right\} \tag{4.26}$$

$$\vdots \quad \vdots \quad \vdots \tag{4.27}$$

$$CS_6 = \left\{ \begin{array}{l} [X \mapsto 0, Y \mapsto 110], [X \mapsto 0, Y \mapsto 120], [X \mapsto 0, Y \mapsto 130], \\ [X \mapsto 0, Y \mapsto 140], [X \mapsto 0, Y \mapsto 150], [X \mapsto 0, Y \mapsto 160], \\ [X \mapsto 0, Y \mapsto 170], [X \mapsto 0, Y \mapsto 180], [X \mapsto 0, Y \mapsto 190], \\ [X \mapsto 0, Y \mapsto 200] \end{array} \right\} \tag{4.28}$$

It can be seen that the number of states and iterations explode for even a simple programme.

4.4 Abstract Domains

To overcome the computability issue, we turn to abstractions of the programme. The abstraction causes over-approximation. This means that some states are analysed, even though they can not appear in the programme. This may cause false alarms, also known as false positives. Note however, that the abstraction keeps the analysis sound, and if there is an error, the analysis will find it. We run the analysis on the abstract domain.

4.4.1 Basic Definitions

We define the abstract lattice $\langle D^\#, \sqsubseteq^\#, \sqcup^\#, \sqcap^\#, \bot^\#, \top^\# \rangle$, and a concretisation of the abstract domain, $\gamma : D^\# \to D$, where $D = P(\Sigma) = P(Var \to \mathbb{R})$. We require that γ is monotone, i.e. $d^\# \sqsubseteq d^{\#\prime} \implies \gamma(d^\#) \subseteq \gamma(d^{\#\prime})$, and strict, i.e. $\gamma(\bot^\#) = \emptyset$ and $\gamma(\top^\#) = P(Var \to \mathbb{R})$. We do not require γ to be one-to-one.

Let α be the abstraction function, such that α forms a Galois Connection with γ. Note that α is not required to be defined in advance, and it can be inferred from γ, from the properties of Galois Connections. However, it is not required for α to be computable.

In this manner, it is possible to analyse and prove elements for more than a single programme at a time.

We constrain all abstract domains with the following requirements:

- Computability
  Commands in the abstract domain must be computable.
- The algorithm is known for $\sqcup^\#$
  The $\sqcup^\#$ operation is used for joins.
- The algorithm is known for $\sqcap^\#$
  The $\sqcap^\#$ operation is used for meet. Meet is used for *assume.*
- The algorithm is known for $\sqsubseteq^\#$
  The $\sqsubseteq^\#$ operation is used to check termination.

As in the concrete case, we define a lattice, $D^\# = \langle D^\#, \sqsubseteq^\#, \sqcup^\#, \sqcap^\#, \bot^\#, \top^\# \rangle$. We define $[\![G(N, E, s)]\!]^\# : D^\# \to N \to D^\#$ as the function to find reachable states in the programme. This is done with the following equations:

$$AS_S = \iota^\# \tag{4.29}$$

$$AS_n = \bigsqcup_{\langle m,n\rangle \in E} C^\# [\![c(\langle m, n\rangle)]\!](AS_m) \tag{4.30}$$

Tarski's theorem shows that there is a unique solution here as well.

Example 3. *Here are several numeric abstract domains.*

- *Signs:*
  The states in this abstract domain only contain sign information on variables, e.g. $x \geq 0$, $y < 0$.
- *Intervals*
  The states in this abstract domain contain, for each variable, the interval in which it can appear, e.g. $x \in [a, b]$.
- *Octagons*
  The states in this abstract domain contain constraints of the form $\pm x \pm y \leq c$.
- *Polehydra*
  The states in this abstract domain contain constraints of the form $\sum a_i x_i \leq c$.

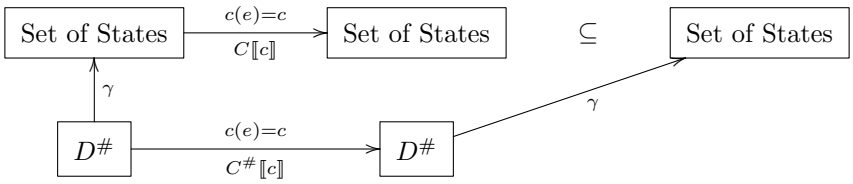


Figure 4.2: Abstraction and soundness

4.4.2 Soundness

The above definitions allow us to say that given a vertex n in the control flow graph, we know that $\forall n \in N.CS_n \subseteq \gamma(AS_n)$. Therefore, any solution in the abstract domain contains at least the solution in the concrete domain. In other words, the reachable concrete states are contained in the concretisation of the reachable states in the abstract domain. Therefore, there are no states in the concrete domain that are not verified with the assertions in the abstract domain. Therefore, no erroneous states will be missed. However, we have over-approximation in the abstract domain. This means that we check additional states, which can cause false alarms.

To ensure soundness, we require the following two conditions:

$$\iota \subseteq \gamma\left(\iota^{\#}\right) \tag{4.31}$$

$$\forall c \in \langle command \rangle, d \in D^{\#}.C[\![c]\!](\gamma(d)) \subseteq \gamma\left(C^{\#}[\![c]\!](d)\right) \tag{4.32}$$

Figure 4.2 shows a schema of how a command affects a set of states in the concrete domain, and in an arbitrary abstract domain. It can be seen that the set of concrete states that match the state in the abstract domain is a super-set of the concrete states that are the result of applying the command on the initial states.

4.4.3 Optimality

In order to ensure optimality, we require an abstraction operator $\alpha : P(\Sigma) \to D^{\#}$ s.t. α and γ form a Galois connection. In this case, we define $C^{\#}[\![c]\!] = \lambda d.\alpha(C[\![c]\!](d))$. Note that α may not always exist, and that $C^{\#}[\![c]\!]$ may be hard to compute.

4.4.4 Widening

We want to be able to treat infinite lattices. To this end, we define a widening operator $\nabla : D^{\#} \times D^{\#} \to D^{\#}$, with the following properties:

$$d \sqcup^{\#} d' \sqsubseteq d \nabla d' \tag{4.33}$$

$$\forall d_0, \ldots, d_n.d_0 \sqsubseteq^{\#} d_1 \sqsubseteq^{\#} \cdots \sqsubseteq^{\#} d_n \sqsubseteq^{\#} \ldots \implies \text{The sequence } s_0 = d_0, s_{i+1} = s_i \nabla d_i \text{ is finite} \tag{4.34}$$

Equation (4.33) ensures us that the widening operator is an over approximation. This way, no erroneous cases are missed. Equation (4.34) promises us that the sequence is finite, and therefore infinite concrete, or even infinite abstract chains can be analysed by over-approximation.

4.5 Non-Relational Domains

Non-relational domains do not keep correlations between variables. Any dependency of a variable on another variable, whether explicitly (e.g. by an assignment) or implicitly (as in a loop running X times, each time increasing a different variable Y with an immediate value of 2), is lost. Formally, the set of abstract states in a non-relational domain can be seen as a mapping $Var \to AbsDomain$, while in a relational domain, it will be $Var \times \ldots \times Var \to AbsDomain$.

4.5.1 Cartezian Abstraction

Cartezian abstraction is a simple, intuitive, non-relational abstraction. Given the concrete domain $D = P(\Sigma) = P(Var \to V)$, we define the abstraction over $D^{\#} = Var \to P(V)$. An abstract state returns a set of values for a given variable. We will begin with a leading example.

Example 4. *Suppose that at vertex n there are two possible states, $\sigma_1 = \{x \to 2, y \to 6\}$ and $\sigma_2 = \{x \to 3, y \to 7\}$. It can be seen that there is a relation between x and y. Specifically, when $x = 2$, then $y = 6$, and otherwise, when $x = 3$, $y = 7$.*

The abstraction of this node will be $\sigma^{\#} = \alpha(\sigma_1, \sigma_2) = \{x \to \{2, 3\}, y \to \{6, 7\}\}$. The relation between x and y is lost. The concretisation of $\sigma^{\#}$ is $\gamma\left(\sigma^{\#}\right) = \{\{x \to 2, y \to 6\}, \{x \to 2, y \to 7\}, \{x \to 3, y \to 6\}, \{x \to 3, y \to 7\}\}$.

Note that even though we have used α to show that an abstraction took place, it is not required for such an operator to exist, nor for it to be computable.

Using this example, we can define the concretisation function $\gamma\left(\sigma^{\#}\right) = \left\{\sigma \in Var \to \mathbb{R} | \forall v \in Var. \sigma(v) \in \sigma^{\#}(v)\right\}$. The abstraction function $\alpha(Z) = \lambda V. \bigcup_{\sigma \in Z} \sigma(V)$ can be inferred trivially from γ. Note that the smaller the set $\gamma\left(\sigma^{\#}\right)$, we have managed to prove more information. As we have shown in the example above, abstracting the states makes us lose the relation between variables.

Example 5. *We re-visit the example programme in example 1. $S = 1$, ι is some single state. Applying the equations (4.29), and (4.30), results in the following sets of states:*

$$
\begin{array}{rcl}
AS_1 & = & \iota \\
AS_2 & = & [X \mapsto \{1 \ldots 10\}] \\
AS_3 & = & [X \mapsto \{1 \ldots 10\}, Y \mapsto 100] \\
AS_4 & = & AS_3 \\
AS_5 & = & [X \mapsto \{1 \ldots 10\}, Y \mapsto 110] \\
AS_3 & = & [X \mapsto \{0 \ldots 9\}, Y \mapsto \{100, 110\}] \\
\vdots & \vdots & \vdots \\
AS_3 & = & [X \mapsto \{0 \ldots 8\}, Y \mapsto \{100, 110, 120\}] \\
\vdots & \vdots & \vdots \\
AS_3 & = & [X \mapsto \{0 \ldots 7\}, Y \mapsto \{100, 110, 120, 130\}] \\
\vdots & \vdots & \vdots \\
AS_6 & = & [X \mapsto 0, Y \mapsto \{100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200\}] \\
\gamma(AS_6) & = & \left\{ \begin{array}{l}
[X \mapsto 0, Y \mapsto 100], [X \mapsto 0, Y \mapsto 110], [X \mapsto 0, Y \mapsto 120], \\
[X \mapsto 0, Y \mapsto 130], [X \mapsto 0, Y \mapsto 140], [X \mapsto 0, Y \mapsto 150], \\
[X \mapsto 0, Y \mapsto 160], [X \mapsto 0, Y \mapsto 170], [X \mapsto 0, Y \mapsto 180], \\
[X \mapsto 0, Y \mapsto 190], [X \mapsto 0, Y \mapsto 200]
\end{array} \right\}
\end{array}
$$

In this example, it can be seen that the concretisation of the last state is indeed a super-set of the application of the analysis on the concrete domain. However, note that the result in the abstract case also includes the concrete state $[X \mapsto 0, Y \mapsto 100]$, which cannot happen. If there was an assertion, say $X - 100 > 0$, there would have been a false alarm.

4.5.2 The Intervals Domain

In the interval abstract domain, a variable is given an interval, i.e. $\sigma^{\#}(V) = \{[a,b] \,|a \leq b \in \mathbb{R} \vee a = -\infty \vee b = \infty\} \cup \{\bot^{\#}\}$. We define $\top^{\#} = [-\infty, \infty]$, as it contains every possible number.

The join operation is defined to be the smallest interval containing both joined intervals. Formally, it is defined as follows:

$$\sigma^{\#}(V) \sqcup^{\#} \sigma^{\#\prime}(V) = \begin{cases} \sigma^{\#\prime}(V) & \sigma^{\#}(V) = \bot^{\#} \\ \sigma^{\#}(V) & \sigma^{\#\prime}(V) = \bot^{\#} \\ [\min(l,l'), \max(u,u')] & \text{Otherwise, where } \sigma^{\#}(V) = [l,u], \sigma^{\#\prime}(V) = [l',u'] \end{cases}$$

The meet operation is defined to be the largest interval that appears in both joined intervales. Formally, it is defined as foolows:

$$\sigma^{\#}(V) \sqcap^{\#} \sigma'(V) = \begin{cases} \bot^{\#} & \sigma^{\#} = \bot^{\#} \vee \sigma^{\#\prime} = \bot^{\#} \\ \bot^{\#} & \max(l,l') > \min(u,u') \text{ where } \sigma^{\#} = [l,u], \sigma^{\#\prime} = [l',u'] \\ [\max(l,l'), \min(u,u')] & \text{Otherwise} \end{cases}$$

The widening operator is defined such that if a join would increase the interval on any one side (lower or upper limit of the interval), that limit is stretched to (minus) infinity.

$$\begin{aligned} \text{Let } \sigma^{\#}(V) &= [l,u] \\ \sigma^{\#\prime}(V) &= [l',u'] \\ a &= \begin{cases} l & l \leq l' \\ -\infty & \text{Otherwise} \end{cases} \\ b &= \begin{cases} u & u \leq u' \\ \infty & \text{Otherwise} \end{cases} \\ \sigma^{\#}(V) \nabla \sigma^{\#\prime}(V) &= \begin{cases} \sigma^{\#\prime} & \sigma^{\#} = \bot^{\#} \\ \sigma^{\#} & \sigma^{\#\prime} = \bot^{\#} \\ [a,b] & \text{Otherwise} \end{cases} \end{aligned}$$

Note that the widening operator heavily depends on the domain. If, for example, the domain states that $\top^{\#} = [0, \infty]$, then the widening operator can set the lower limit to 0. If the domain states other limits, then they can also be applied to the operator.

4.5.2.1 Galois Connection

We now define the galois connection for the interval abstract domain. Let $\Sigma = P(Var \to \mathbb{R})$, and $\Sigma^{\#} = P\left(Var \to \left(\mathbb{R} \times \mathbb{R} \cup \left\{\bot^{\#}\right\}\right)\right)$. Note that in some cases we will ignore the case when the value is $\bot^{\#}$, since any operation with $\bot^{\#}$ remains $\bot^{\#}$.

The concretisation function $\gamma : \Sigma^{\#} \to P(\Sigma)$, is defined as

$$\gamma\left(\sigma^{\#}\right) = \begin{cases} \emptyset & \sigma^{\#} = \bot^{\#} \\ \left\{\sigma | \sigma \in P\left(Var \to \mathbb{R}\right) \wedge \forall v \in Var . \sigma\left(v\right) \in \sigma^{\#}\left(v\right)\right\} & \text{Otherwise} \end{cases}$$

Note that $\sigma^{\#} \in \Sigma^{\#}$ is a function $\sigma^{\#} : Var \to \left(\mathbb{R} \times \mathbb{R} \cup \left\{\bot^{\#}\right\}\right)$ which returns an interval of possible values for the queried variable.

We would like the interval to be as small as possible. This way it represents fewer states, and our analysis is stronger.

We now apply the abstract semantics on the interval abstract domain. We want to always return the smallest possible interval. The syntactic definition of the expression evaluation semantics is:

$$\begin{array}{rcl} E\left[\!\left[\langle exp \rangle\right]\!\right] & : & \left(Var \to \mathbb{R}\right) \to P\left(\mathbb{R}\right) \\ E^{\#}\left[\!\left[\langle exp \rangle\right]\!\right] & : & \left(Var \to \left(\mathbb{R} \times \mathbb{R} \cup \left\{\bot^{\#}\right\}\right)\right) \to \left(\mathbb{R} \times \mathbb{R} \cup \left\{\bot^{\#}\right\}\right) \end{array}$$

Therefore, we define the expression semantics as follows, where $E^{\#}\left[\!\left[\langle exp \rangle\right]\!\right]\left(\sigma^{\#}\right) = [l, u]$ if $\langle exp \rangle \neq \bot^{\#}$, and $E^{\#}\left[\!\left[\langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) = [l', u']$ if $\langle exp' \rangle \neq \bot^{\#}$.

$$E^{\#}\left[\!\left[V\right]\!\right]\left(\sigma^{\#}\right) = \sigma^{\#}\left(V\right) \tag{4.35}$$

$$E^{\#}\left[\!\left[[c, c']\right]\!\right]\left(\sigma^{\#}\right) = [c, c'] \tag{4.36}$$

$$E^{\#}\left[\!\left[-\langle exp \rangle\right]\!\right]\left(\sigma^{\#}\right) = \begin{cases} \bot^{\#} & E^{\#}\left[\!\left[\langle exp \rangle\right]\!\right]\left(\sigma^{\#}\right) = \bot^{\#} \\ [-u, -l] & \text{Otherwise} \end{cases} \tag{4.37}$$

$$E^{\#}\left[\!\left[\langle exp \rangle + \langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) = \begin{cases} \bot^{\#} & E^{\#}\left[\!\left[\langle exp \rangle\right]\!\right]\left(\sigma^{\#}\right) = \bot^{\#} \vee E^{\#}\left[\!\left[\langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) = \bot^{\#} \\ [l + l', u + u'] & \text{Otherwise} \end{cases} \tag{4.38}$$

$$E^{\#}\left[\!\left[\langle exp \rangle - \langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) = \begin{cases} \bot^{\#} & E^{\#}\left[\!\left[\langle exp \rangle\right]\!\right]\left(\sigma^{\#}\right) = \bot^{\#} \vee E^{\#}\left[\!\left[\langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) = \bot^{\#} \\ [l - u', u - l'] & \text{Otherwise} \end{cases} \tag{4.39}$$

$$E^{\#}\left[\!\left[\langle exp \rangle \cdot \langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) = \begin{cases} \bot^{\#} & E^{\#}\left[\!\left[\langle exp \rangle\right]\!\right]\left(\sigma^{\#}\right) = \bot^{\#} \vee E^{\#}\left[\!\left[\langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) = \bot^{\#} \\ \left[\min\left(l \cdot l', l \cdot u', u \cdot l', u \cdot u'\right), \right. & \text{Otherwise} \\ \quad \left. \max\left(l \cdot l', l \cdot u', u \cdot l', u \cdot u'\right)\right] & \end{cases} \tag{4.40}$$

$$E^{\#}\left[\!\left[\langle exp \rangle \,/\, \langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) = \begin{cases} \bot^{\#} & E^{\#}\left[\!\left[\langle exp \rangle\right]\!\right]\left(\sigma^{\#}\right) = \bot^{\#} \vee E^{\#}\left[\!\left[\langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) = \bot^{\#} \\ \bot^{\#} & \{0\} = E^{\#}\left[\!\left[\langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) \\ \top^{\#} & 0 \in E^{\#}\left[\!\left[\langle exp' \rangle\right]\!\right]\left(\sigma^{\#}\right) \\ \left[\min\left(l/l', l/u', u/l', u/u'\right), \right. & \text{Otherwise} \\ \quad \left. \max\left(l/l', l/u', u/l', u/u'\right)\right] & \end{cases} \tag{4.41}$$

In equation (4.41), we take special care when dividing by zero. Dividing by an arbitrary number close to zero can return an arbitrary large number, and that is why we return $\top^{\#}$. Additionally, dividing by zero proper is undefined, so we cannot eliminate any state. This is also necessary to maintain monotonicity.

In case we know that the divisor is exclusively and exactly 0, then the division result is undefined. Therefore, we can set the result to $\bot^{\#}$.

The command semantics are

$$C^{\#}\left[\!\left[\langle command\rangle\right]\!\right] \quad : \quad P\left(Var \to \left(\mathbb{R}\times\mathbb{R}\cup\bot^{\#}\right)\right) \to P\left(Var \to \left(\mathbb{R}\times\mathbb{R}\cup\bot^{\#}\right)\right) \tag{4.42}$$

$$C^{\#}\left[\!\left[V := \langle exp\rangle\right]\!\right]\left(\sigma^{\#}\right) \quad = \quad \sigma^{\#}\left[V \mapsto E^{\#}\left[\!\left[\langle exp\rangle\right]\!\right]\left(\sigma^{\#}\right)\right] \tag{4.43}$$

$$C^{\#}\left[\!\left[\text{assume } \langle exp\rangle \bowtie 0\right]\!\right]\left(\sigma^{\#}\right)(V) \quad = \quad \sigma^{\#} \sqcap^{\#} \begin{cases} [a,b] & V \text{ in } \langle exp\rangle, \langle exp\rangle \text{ holds when } V \in [a,b] \\ \top^{\#} & V \text{ not in } \langle exp\rangle \end{cases} \tag{4.44}$$

In equation (4.44), we reduce the interval of any variable only to the interval in which the relation holds. We will show this in an example.

Example 6. *Suppose we are given the command* $C^{\#}\left[\!\left[\text{assume } x \geq 0\right]\!\right]$ *on state* $\sigma^{\#}$. *Therefore,* $\sigma^{\#\prime} = C^{\#}\left[\!\left[\text{assume } x \geq 0\right]\!\right]\left(\sigma^{\#}\right)$, *where* $\sigma^{\#\prime}(V) = \begin{cases} \sigma^{\#}(V) & V \neq x \\ \sigma^{\#}(V) \sqcap^{\#} [0,\infty] & V = x \end{cases}$. *In this way, variables that are not in the expression are not affected. Variables in the expression are narrowed only to the values where the expression holds.*

This semantics is sound, since $C\left[\!\left[\langle command\rangle\right]\!\right]\left(\gamma\left(\sigma^{\#}\right)\right) \subseteq \gamma\left(C^{\#}\left[\!\left[\langle command\rangle\right]\!\right]\left(\sigma^{\#}\right)\right)$. Therefore, the result of applying the command is an over-approximation, and no states are lost. Specifically in *assume*, we can define $C^{\#}\left[\!\left[\text{assume } \langle exp\rangle \bowtie 0\right]\!\right]\left(\sigma^{\#}\right) = \sigma^{\#}$.

However, some information is lost, which leads to false alarms, for instance:

Example 7. *Let's review the command* $X := Y - Y$. *Syntacticly, the result should be* $\sigma^{\#}[X \mapsto [0,0]]$. *However, according to the semantics, where* $Y = [l,u]$ *(If* $Y = \bot^{\#}$, *then* $X = \bot^{\#}$*), then*

$$\begin{aligned} X &:= Y - Y \\ &= \sigma^{\#}[X \mapsto Y - Y] \\ E^{\#}\left[\!\left[Y - Y\right]\!\right]\left(\sigma^{\#}\right) &= [l-u, u-l] \\ &\neq [0,0] \quad \textit{Unless } Y \textit{ is known exactly.} \end{aligned}$$

Here X *includes 0, i.e.* $0 \in \sigma^{\#}[l-u, u-l]$, *but it also includes other values. This may give rise to assertion errors on states that can not exist.*

There are several options as to how to deal with the situation in the example above.

1. Use a stronger abstract domain, which retains more information.

2. Temporarily operate on the concrete domain

3. Use a theorem prover to prove certain queries. This solution is still relatively cheap, since the theorem prover is not called often. However, changing from one method to the other is costly.

Example 8. *We review the programme in example (1). When applying equations (4.29) and (4.30), we get*

the following states.

$$\begin{aligned}
AS_1 &= \iota \\
AS_2 &= [X \to [1,10]] \\
AS_3 &= [X \to [1,10], Y \to [100,100]] \\
AS_4 &= AS_3 \\
AS_5 &= [X \to [1,10], Y \to [110,110]] \\
AS_3 &= [X \to [0,10], Y \to [100,110]] \\
AS_4 &= [X \to [1,10], Y \to [100,110]] \\
AS_5 &= [X \to [1,10], Y \to [100,110]] \\
AS_3 &= [X \to [0,10], Y \to [100,120]] \\
&\vdots \quad \vdots \quad \vdots \\
AS_6 &= [X \to [0,0], Y \to [100,\infty]]
\end{aligned}$$

In this case we see that the abstraction covers many more states than in the concrete case, or even in the cartezian case. There are many assertions, e.g. $Y - 200 \leq 0$*, that will fail in this abstraction, but not in the concrete case.*

4.5.3 The Signs Domain

This domain will retain the information regarding the sign of the (integer) variable. The abstract domain $D^{\#}$ will contain 3 elements: $\{+, -, 0\}$, as well as $\top, \bot$. It is imprecise but very efficient. Following α and γ will form the Galois Connection;

$$\alpha(X) = \begin{cases} \bot & X = \phi \\ 0 & X = \{0\} \\ + & X = \{z | z \geq 0\} \\ - & X = \{z | z \leq 0\} \end{cases}$$

$$\gamma(a) = \begin{cases} \mathbb{Z}^+ & a = + \\ \mathbb{Z}^- & a = - \\ \{0\} & a = 0 \\ \phi & a = \bot \\ \mathbb{Z} & a = \top \end{cases}$$

The expressions can also be easily defined in abstract terms:

$$E^{\#}[-x] = \begin{cases} \bot & x = \bot \\ - & x = + \\ + & x = - \\ 0 & x = 0 \\ \top & x = \top \end{cases}$$

$$E^{\#}[x+y] = \begin{cases} \bot & x = \bot \vee y = \bot \\ + & x = + \wedge y = + \\ - & x = - \wedge y = - \\ 0 & x = 0 \wedge y = 0 \\ x & y = 0 \\ y & x = 0 \\ \top & otherwise \end{cases}$$

$$E^{\#}[x*y] = \begin{cases} \bot & x = \bot \vee y = \bot \\ + & (x = + \wedge y = +) \vee (x = - \wedge y = -) \\ - & (x = + \wedge y = -) \vee (x = - \wedge y = +) \\ 0 & x = 0 \vee y = 0 \\ \top & otherwise \end{cases}$$

$$E^{\#}[x/y] = \begin{cases} \bot & x = \bot \vee y = \bot \vee y = 0 \\ + & (x = + \wedge y = +) \vee (x = - \wedge y = -) \\ - & (x = + \wedge y = -) \vee (x = - \wedge y = +) \\ 0 & x = 0 \\ \top & otherwise \end{cases}$$

Commands:

$$C^{\#}[x := exp](d) = d[x \mapsto E^{\#}[exp]d]$$

$$C^{\#}[assume\ x\, relop\, 0](d) = d \sqcap \lambda_v \begin{cases} \{0\} & v = x, relop = '=' \\ \{+,-\} & v = x, relop = '\neq' \\ \{-,0\} & v = x, relop = '\leq' \\ \{-\} & v = x, relop = '<' \\ \{0,+\} & v = x, relop = '\geq' \\ \{+\} & v = x, relop = '>' \\ D^{\#} & otherwise \end{cases}$$

Assuming on expressions may be more complex (for example, assuming x+y¿0 and having y=-, surely we can leave only states where x=+). By not filtering out any states we stay sound, albeit less exact:

$$C^{\#}[assume\ exp\, relop\, 0](d) = d$$

For example, consider this program:

```
x := 100
y := 10
while (x - y > 0) {
        x := x - y
```

```
        y := y * 10
}
assert x > 0
```

The while condition is translated to an "assume x - y ¿ 0". Both x and y are known to be equal '+'. The value of the abstract boolean condition can't be evaluated to a definite true or false, therefore none of the states are filtered out. It will conclude that $x = \top, y = +$ even though given the initial values, x will never be negative.

4.5.4 The Congruence Domain

The congruence domain maps each variable v to 2 constants (a, b) such that $x \in a\mathbb{Z}+b$. Upon each command, the abstraction will map the variable to the equivalence class $a\mathbb{Z} + b$ that retains most information on the possible values of the variable. Therefore:

$$\alpha(v) = 0 \cdot \mathbb{Z} + v$$

$$\gamma(d = a\mathbb{Z} + b) = \begin{cases} \{ak + b | k \in \mathbb{Z}\} & a \neq 0 \\ \{b\} & a = 0 \end{cases}$$

To retain most information, the join operator will be defined using the gcd of the current moduli and the remainders' difference:

$$(a\mathbb{Z} + b) \sqcup (a'\mathbb{Z} + b') = gcd(a, a', b - b')\mathbb{Z} + min(b, b')$$

It is also required to define a partial order on the congruence lattice. Intuitively, each equivalence class $a\mathbb{Z}+b$ is an inifnite set of numbers, but all those numbers should belong to other equivalence classes as well. For example, all numbers belong to $1 \cdot \mathbb{Z} + 0$, which is also equal to $\top$. Also, all classes of the form $2^n\mathbb{Z} + 1$ belong to $2 \cdot \mathbb{Z} + 1$. On the other hand, classes of $p\mathbb{Z} + k$, where p is a prime number and $k \in \{0, ..., p - 1\}$ are the "smallest" possible classes. Formally:

$$(a\mathbb{Z} + b) \sqsubseteq (a'\mathbb{Z} + b') \iff (a' \mid a) \wedge (b \equiv b' \mod a')$$

This abstract domain is in a way unique from previous domains encountered in class, because it contains inifintely decreasing chains (starting from $1 \cdot \mathbb{Z}$ and continuing with infinite combinations of prime factors $p_1 p_2 \dots p_n\mathbb{Z}$), but not infinitely increasing chains (because each number has a prime number factorization). In such a domain, there's no need for a widening operator, just a narrowing operator.

Expressions can be evaluated using known rules of Number theory, in a way that retains maximum information. Imprecision will be encountered only upon conditions and loops, when the join operator will be activated. This is still a severe limitation of the domain.

Consider the following program:

```
x := 0
y := 2
while (x < 40) {
        x := x + 2
```

```
        if x < 5
                y := y + 18
        if x > 8
                y := y - 30
}
```

The abstract interpretation of the program will discover that x has a factor of 2, that is $x \in 2\mathbb{Z}$, and that $y \in 6\mathbb{Z}+2$ - as the gcd of the constants modifying y (18,30) is 6. Note, however, that the analysis will not be able to detect termination of the loop, as the actual cardinality of a variable disappears - only its congruence class.

4.6 Relational Domains

Relational domains keep information on the correlations between variables. Such information is naturaly giving more precise analysis compared with non-relational domains. It is very powerful in programs where the dependency graph is not trivial, and without it certain properties of the program can't be proven.

For example, the following program should emit in every iteration a signal (in the Y variable) which is bounded (in [-128, 128]), but running chaotic iteration in the non-relational interval domain will not be able to prove it:

```
Y := 0
while true {
        X := Random([-128, 128])
        D := Random([0, 16])
        S := Y
        Y := X
        R := X - S
        if R <= -D
                Y := S - D
        if R >= D
                Y := S + D
        emit Y
}
```

The interval domain cannot determine which of the branchings took place, thus increasing the interval by 16 in each iteration. With widening will give $Y = [-\infty, \infty] = \top$.

It is also useful to conclude inductive statements which are strong enough to be meaningful. For example:

```
        X := 0
        I := 1
        while (I < 5000) {
                if Random([0,1]) == 1
                        X := X + 1
                else
                        X := X - 1
                I := I + 1
        }
```

With the interval domain, widening and narrowing operators will discover that in the end of the loop, the value of I is 5000, and X is $[-\infty, \infty]$. If the relation between X and I is kept ($-I \leq X \leq I$), then it is possible to discover that $X \in [-4999, 4999]$ which is more precise.

Additionaly, relational information paves the way for modular and procedural analysis, by providing the relations between the input and output arguments of a procedure (both out parameters and side effects to the original referenced variables). At each call to the procedure, use the relational data calculated before instead of re-analyzing the procedure. For example, the following procedure takes 2 arguments X, Y and outputs $max(x, y, 0)$ into a variable Z. So by denoting the input variables as X, Y, Z, and the output variables as X', Y', Z', the relation to be obtained is $X = X',\ Y = Y',\ Z' \geq X \wedge Z' \geq Y \wedge Z' \geq 0$.

4.6.1 The Zone Domain

The zone domain abstraction saves information on variables' differences and bounds. Specifcially, it keeps constraints of the form:

$V_i - V_j \leq c$

$\pm V_i \leq c$

The constraint information can be kept in several representations, each useful for different operations on the lattice.

The different representations:

- Inequality system
- Matrix form - called DBM (Difference Bound Matrix. A $n \times n$ matrix where $A_{i,j}$ corresponds to the constraint to $c_{i,j}$ such that $V_i - V_j \leq c_{i,j}$. If the constraint on the difference $V_i - V_j$ is not known, the value of $A_{i,j}$ is ∞. Unary constraints ($\pm V_i \leq c$) are represented using a dummy variable V_0 whose value is 0.
- Potential graph - Each variable corresponds to a node, and each edge label corresponds the constraint on the 2 nodes it connects. The DBM form presented above is actually the adjacency matrix of the potential graph.
- Geometry - the set of all points corresponding to the variables form a convex shape in R^n.

As an example, take the following constraint system on 2 variables V_1, V_2:

$$-4 \leq V_1 \leq -1$$

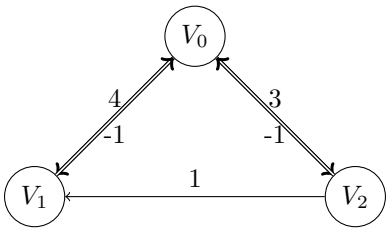
$$-3 \leq V_2 \leq -1$$

$$V_2 - V_1 \leq 1$$

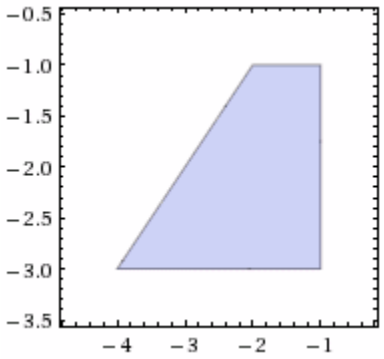
It can be represented with this DBM matrix:

$$\begin{matrix} \infty & 4 & 3 \\ -1 & \infty & \infty \\ -1 & 1 & \infty \end{matrix}$$

With this graph:



And the following geometry:



The concretization function γ is constructed from the DBM form as follows:

$$m : N \times N \to R \cup \infty$$

$$\gamma(m) = \{\sigma \in Var \to R, \forall v_1, v_2 \in Var.\ m(v_1, v_2) \geq \sigma(v_1) - \sigma(v_2)\}$$

In words, the concretization function maps to all concrete states where all variables fulfill the constraints induced by the DBM matrix m.

Proceeding with the definition of the DBM lattice:

- $\top = \begin{matrix} \infty & \infty & \cdots \\ \infty & \infty & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$
- $\bot$ = graph with a negative circle
- $\sqcup$ = matrix with each cell containing the maximum over all corresponding cells of the matrices participating in the join.
- $\sqcap$ = matrix with each cell containing the minimum over all corresponding cells of the matrices participating in the join.
- $\sqsubseteq$= all cells in the first argument matrix should be smaller or equal to those in the second argument matrix. This yields a partial order on the matrices.

It is important to note that before each operation, the transitive closure of the graph should be taken, with constraint bounds serving as the weights (can be done by running the Floyd-Warshall algorithm). This will yield more exact DBM matrices. In the example, taking the transitive closure will yield that $V_1 - V_2 \leq 2$.

The closure's importance is in finding a normal form for the matrix. It is also important in order to define the $\perp$ value. Any other definition would yield an incomplete lattice. It is also required in order to define an optimal join operator - whose result will map under γ to the smallest set of states containing both operands' concretizations. Take for example this program:

```
x := Random([-4,-1]) // Pick random int in [-4,-1]
if (x == -4) {
        repeat
                y := Random[-3,-1]
        until (y <= x + 1)
} else {
        repeat
                y := Random[-4,-1]
        until (|x-y| <= 1)
}
assert (x,y) != (-1,-4)
```

The assert can be easily verified by hand - x is chosen first, so if x==-4, the assert will be surely false. Otherwise, assuming x=-1, y is picked in the range [-4,-1] such that the absolute value of the difference from x is not larget than 1. So y can't be equal to -4.

However, if we use the zone domain abstraction, we do not know which branch is taken. So upon reaching the assert, we're actually joining the constraints from both branches. In the x==-4 branch we have:

$$A = \begin{matrix} \infty & 4 & 3 \\ -4 & \infty & \infty \\ -1 & 1 & \infty \end{matrix}$$

Whose transitive closure is:

$$A^* = \begin{matrix} \infty & 4 & 3 \\ -4 & \infty & 0 \\ -1 & 1 & \infty \end{matrix}$$

as we have a path from x to y with weight 0 (x to V_0 through an edge with weight -4, and from V_0 to y through an edge with weight 4).

On the else branch, we assume $-x \leq 3$ (the other option, $x \leq -5$, is impossible as it will yield a negative cycle, thus $\perp$). The DBM matrix in the end of the else branch is:

$$B = \begin{matrix} \infty & 3 & 4 \\ -1 & \infty & 1 \\ -1 & 1 & \infty \end{matrix} = B^*$$

So in the assert, we take the join. Below are the join with and without transitive closures taken:

$$A \sqcup B = \begin{matrix} \infty & 4 & 4 \\ -1 & \infty & \infty \\ -1 & 1 & \infty \end{matrix}$$

$$A^* \sqcup B^* = \begin{matrix} \infty & 4 & 4 \\ -1 & \infty & 1 \\ -1 & 1 & \infty \end{matrix}$$

The only difference between the matrices is the $x - y \leq 1$ constraint. This is the set of constraints shared between both $A \sqcup B, A^* \sqcup B^*$:

$$-4 \leq x \leq -1$$

$$-4 \leq y \leq -1$$

$$y - x \leq 1$$

In this constraint system, if $x = -1, y = -4$ we have fulfillment of all 3 constraints, thus leading to a failure of the assert, a false alarm. With the added constraint $x - y \leq 1$ coming from the correct join, it is impossible and the assert is correctly verified.

4.6.2 The Octagon Domain

The Octagon domain extends the zone domain by allowing greater flexibility with the constraints. Instead of just bounding variables' differences ($V_i - V_j \leq c$), the following inequalities are also kept:

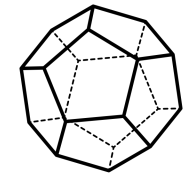
$$\pm V_i \pm V_j \leq c$$

4.6.3 The Polyhedra Domain

The polyhedra domain handles constraints of a much more general form:

$$\wedge_i \sum_j a_{i,j} x_{i,j} \geq c_i$$

Note that the number of variables participating in the constraint is not bounded.



Created by Michael Senkow
from Noun Project

The polyhedra constraints provide better precision but with the cost of harder to compute operations. These constraints need to be constantly transformed to some normal form in order to make comparisons and perform join/meet operations.

The join operation is visualized as the convex hull of the shapes in R^n space. Intuitively, this means taking the smallest convex shape containing all the shapes provided, even if disjoint. It is an overapproximation of the shape, so the analogy to sound static analysis is clear:

The polyhedra domain has two dual representations: the constraint representation (as above) and the generator representation (the convex hull). The intuition is that the polyhedron shape can be also defined by its generators: a set of vectors consisting of: points, which are the polyhedron's vertices, and for unbounded polyhedra, rays. The duality theorem of Weyl-Minkowsky claims that every polyhedron is finitely generated (thus representable by such points and rays) and every finitely generated set is a polyhedron (i.e. its points form a convex hull). Furthermore, we can move between the 2 representations without losing precision. However, there is a trade-off in performance, as the algorithm used to constantly transform between the two representations (Chernikova's algorithm) is costly, and the generator representation's size is exponential in the size of the original constraint system. The alternative is to use only the constraint representation, while using linear programming to minimizing the constraint system (which may still be a challenging task performance-wise).

A convenient method to experiment with the polyhedra domain is the Interproc analyzer. Below is an example of a very simple program where we see that a non-intuitive change of the condition leads to a better, more precise analysis in the polyhedra abstract domain:

```
1:      var x : int, y : int, res : int;
2:      begin
3:              assume x >= 0 and y >= 0 ;
4:              if x == 0 then
5:                      res = y + 1;
6:              else
7:                      res = y - 1;
8:              endif;
9:      end
```

The constraint representation of the above program after each step (before and after minimization, where applicable) is:

$3 : \{x \geq 0, y \geq 0\}$

$4 : \{x = 0, y \geq 0\}$

$5 : \{x = 0, y \geq 0, res = y + 1\} \rightarrow \{x = 0, y \geq 0, res = y + 1, res \geq 1\}$

$6 : \{x \geq 0, y \geq 0\}$

$7 : \{x \geq 0, y \geq 0, res = y - 1\} \rightarrow \{x \geq 0, y \geq 0, res = y - 1, res \geq -1\}$

$8, 9 : \{x \geq 0, y \geq 0, res \geq y - 1, res \leq y + 1\}$

The dual generator representation is:

```
The vectors represent the variables in the following order: (x,y,res)
3: A vertex at (0,0), and rays in the direction (0,1) and (1,0)
4: A vertex at (0,0), and a ray in the direction (0,1)
5: A vertex at (0,0,1) and a ray in the direction (0,1,1)
6: like 3
7: A vertex at (0, 0, -1), and rays in the direction (1,0,0) and (0,1,1)
```

8,9: Vertices at (0,0,1), (0,0,−1) and rays in the direction (1,0,0) and (0,1,1)

Here are graphic representation of the polyhedrons (generated with Sage):

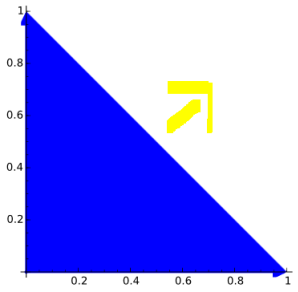


Figure 4.3: Abs. state after 3

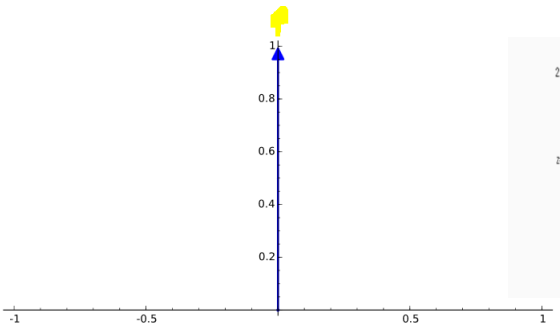


Figure 4.4: Abs. state after 4

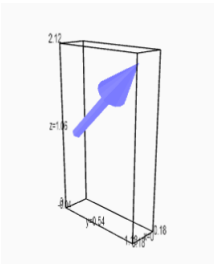


Figure 4.5: Abs. state after 5

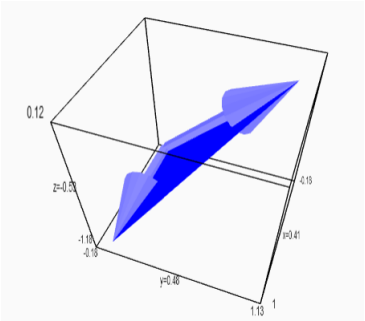


Figure 4.6: Abs. state after 7

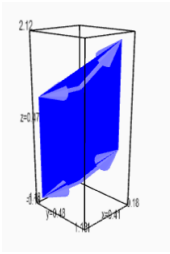


Figure 4.7: Abs. state after 9

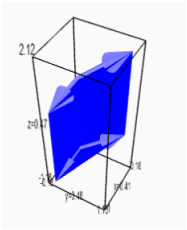


Figure 4.8: Abs. state after 9, different angle

An important note regarding this is that the representation is unable to handle $x \neq 0$ assumption in the else of line no.6, as the values could be over fractions and not just integers.

It is clear, that under the assumption that $x \geq 0$, then:

- in the else, $x \geq 1$.
- *res* is equal to $y + 1$ if $x = 0$, and to $y - 1$ otherwise.

Running the analyzer with the polyhedra abstract domain in Interproc, we get that:

- In the else condition, $x \geq 0$ - but how could x be 0 if we take only non-negative x-s to begin with, and the if itself handles $x = 0$? It seems that the analysis applied meet on the negation of the condition $x = 0$ and with $\top$, instead of with $x \geq 0$.
- In the end of the program, $res \geq y - 1$, which is sound, but does not take into account how the value of x affects res.

A tiny change to the program will give much better results, and it is to change the condition to $x \leq 0$ as follows:

```
var x : int, y : int, res : int;
begin
        assume x >= 0 and y >= 0 ;
        if x <= 0 then                  // Changed line
                res = y + 1;
        else
                res = y - 1;
        endif;
end
```

Here the analysis gives us that:

- In the else condition, $x - 1 \geq 0$, that is $x \geq 1$.
- In the end of the program, we have $res + 2x - y - 1 \geq 0$, or $res \geq y + 1 - 2x$. Which indeed, if $x = 0$, $res \geq y + 1$ is sound, and if $x \neq 0$, it must be that $x \geq 1$, therefore $res \geq y + 1 - 2x$. It is a weaker condition than $res \geq y - 1$ (which is detected correctly even in the "wrong" code), as $y + 1 - 2x$ could be much smaller than $y - 1$, but it shows the correlation of res to the value of x. It unveils the fact that when x is 0, $res = y + 1$, which we could not infer from the analysis of the first program.

This example demonstrates how miniature changes in the code, which do not affect its concrete semantic correctness and meaning, can greatly affect the precision of the analysis.