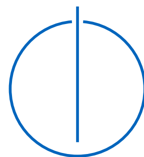
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# Disjunction and Intervals in Industrial-Strength Static Analysis

Sebastian Wilzbach



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

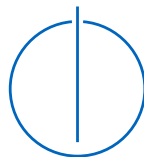
# Disjunction and Intervals in Industrial-Strength Static Analysis

# Disjunktion und Intervalle in statischer Analyse mit industriellem Maßstab



| | |
|---|---|
| Author: | Sebastian Wilzbach |
| Supervisor: | Prof. Dr. Helmut Seidl |
| Advisor: | Prof. Dr. Helmut Seidl |
| Submission Date: | 15.09.2020 |





I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Saarbrücken, 15.09.2020 Sebastian Wilzbach

# Acknowledgments

I would first like to thank my thesis advisor *Prof. Dr. Helmut Seidl* for introducing me to this project as well as providing me with important suggestions in the early stages of this thesis.

Furthermore, I would like to express my gratitude to *Dr. Christian Ferdinand* for offering me this opportunity and *AbsInt Angewandte Informatik GmbH* for their support throughout this thesis.

I would like to offer my special thanks to *Dr. Stephan Wilhelm* for his continued help, advice and encouragements. As a reader of this thesis, I am gratefully indebted to him for his very valuable comments.

I would like to offer my special thanks to *Prof. Dr. Laurent Mauborgne* for his feedback and advice which was greatly appreciated.

I wish to thank *Philipp Albert* for his help in maintaining the technical infrastructure and ensuring working from home was a seamless experience in these difficult times.

Finally, I would like to extend my thanks to all of my other colleagues from *AbsInt Angewandte Informatik GmbH* for their input, help and support.

# Abstract

*Abstract Interpretation* is a powerful technique to statically analyze and prove semantic properties of a program. Finding exact answers to general questions about program semantics is an undecidable problem, hence abstract interpreters approximate answers by *specialized abstract domains*. A common abstract domain is the *interval domain* which stores the range of all potential values of a variable as an interval. However, a frequent pattern in industrial code is to use values outside of the range of the application domain as additional information about a variable (e.g. `MAX_INT` as uninitialized or error state). Such *extremal values* can lead to a high loss of information with the interval domain and similar abstract value domains. In this work we investigate alternative domains to retain precision for extremal values. An implementation of a dedicated domain - a *set of disjunctive intervals* with a user-defined *maximum cardinality* $N$ - is discussed in-depth and has been added to the Astrée Static Analyzer. We evaluated the impact of this new abstract domain on 19 medium-sized ($10 - 100$k analyzed LoC) codes from real-world software (mostly automotive and avionic software) and one large industrial automotive code ($\sim 400$k analyzed LoC) in comparison to Astrée's existing domains. We found that the new abstract domain helped to reduce false run-time errors by on average 1.4% per project in medium-sized codes and 0.6% in large industrial code, increased the number of unreachable blocks by 1.3% per project in medium-sized codes and 5.5% in a large industrial code. Depending on $N$, we measured an increase in memory usage by $5 - 8\%$ per project in medium-sized codes and a decrease in the large code example by $18 - 38\%$ and a run-time duration increase by $7 - 22\%$. Most improvements were observed with $N = 2$ yielding $\sim 50 - 80\%$ of all improvements. $N = 3$ increased the results noticeable ($\sim 10 - 40\%$) whereas any higher $N$ only lead to slight additional improvements ($< 10\%$).

# Contents

**Acknowledgments** **iii**

**Abstract** **iv**

**1 Introduction** **1**

**2 Interval abstract domain** **5**
2.1 Numerical domain . . . . . 5
2.2 Interval operations . . . . . 5
2.3 Partial order . . . . . 6
2.4 Interval lattice . . . . . 7
2.5 Abstract relationship . . . . . 9
2.6 Interval abstract operators . . . . . 11
2.6.1 Forward operators . . . . . 11
2.6.2 Backward operators . . . . . 13
2.7 Fixpoint approximation . . . . . 15
2.7.1 Widening . . . . . 15
2.7.2 Narrowing . . . . . 17

**3 IntervalList abstract domain** **18**
3.1 Definition . . . . . 18
3.2 Partial order . . . . . 20
3.3 Lower and upper bound . . . . . 21
3.4 Hasse diagram . . . . . 21
3.5 Normalization . . . . . 23
3.6 Lattice . . . . . 24
3.7 Abstract and concrete relationships . . . . . 26
3.8 Galois connection . . . . . 26
3.9 Soundness correspondence . . . . . 27

**4 IntervalList operations** **30**
4.1 Relations . . . . . 30
4.1.1 Subset . . . . . 30

4.1.2 Equality . . . . . 31
4.2 Transfer functions . . . . . 31
4.2.1 Union . . . . . 31
4.2.2 Intersection . . . . . 33
4.2.3 Forward arithmetic initialization . . . . . 34
4.2.4 Forward unary arithmetic operations . . . . . 35
4.2.5 Forward binary arithmetic operations . . . . . 38
4.2.6 Backward comparison tests . . . . . 40
4.2.7 Widening . . . . . 45

**5 Results** **52**
5.1 Medium-sized project evaluation . . . . . 53
5.1.1 Unreachable code statements . . . . . 53
5.1.2 Run-time error findings . . . . . 55
5.1.3 Analysis duration and memory usage . . . . . 56
5.2 Large industry example . . . . . 57
5.2.1 Unreachable code statements . . . . . 58
5.2.2 Alarm findings . . . . . 58
5.2.3 Analysis runtime and memory usage . . . . . 60
5.3 Code patterns of improved alarms . . . . . 62
5.3.1 Unreachable code . . . . . 62
5.3.2 Array Out-of-Bounds . . . . . 64
5.3.3 Invariants . . . . . 64
5.3.4 Read of a not written global variable . . . . . 65

**6 Summary** **67**

**List of Figures** **69**

**List of Tables** **70**

**Bibliography** **71**

# 1 Introduction

*Static analysis* of programs is an analysis technique in which program properties are determined without executing the program. *Abstract Interpretation* introduced by [CC77] is a formal static analysis framework that approximates code from its original properties (*concrete domain*) into an alternative universe (*abstract domain*) with its own set of properties. An *abstract interpreter* executes program computations in abstract domains and, thus can approximate program semantics and statically provide answers about certain program properties and run-time behaviors. In general, it is undecidable whether a program given arbitrary input terminates (halting problem), hence an answer from the abstract interpreter cannot necessarily be precise. However, this answer is a *sound approximation* which guarantees to contain the exact answer.

A sound abstract interpreter cannot omit run-time errors (false negatives) from its answer, but can over-approximate and emit *false alarms* (false positives). An abstract interpreter will signal all *potential* and *certain* run-time errors [Cou00]. For a sound abstract interpreter, the absence of alarms in its analysis formally guarantees the *absence of errors at run-time*.

Many abstract domains have been proposed for handling different semantic aspects of programs and these domains are typically combined to achieve a better approximation. In this work we focus on *non-relational abstract domains* which specialize on value properties of numerical variables in code and ignore potential relationships between variables. Consider this illustratory program:

```
int main(int a)
{
  int y = 2; // y = 2
  if (a) { y = 8; }
  return y; // y = ?
}
```

In this example, $y$ could be abstracted by an interval domain ($[2,8]$), a bitfield domain (00001010), a congruence domain ($2 * x + 0$), a finite set domain ($\{2,8\}$), or a non-zero domain ($+$). Each of these abstract domains provide a *trade-off between loss of precision vs. analyzer runtime* and memory consumption of the analysis. In this work, we evaluate the trade-off for a specific subset of programs - programs with

```
unsigned char in_fn() {
  unsigned char x = 255;
  // read input
  if (random > 0) {
    x = (unsigned char) random * 4;
  }
  return x;
}
```

```
void out_fn(unsigned char x) {
  if (x == 255) {
    // special error handling
  } else {
    switch (x) {
      case 0: case 1: case 2: case 3:
      ...
    }
}}
```

```
void main(void) { out_fn(in_fn()); }
```

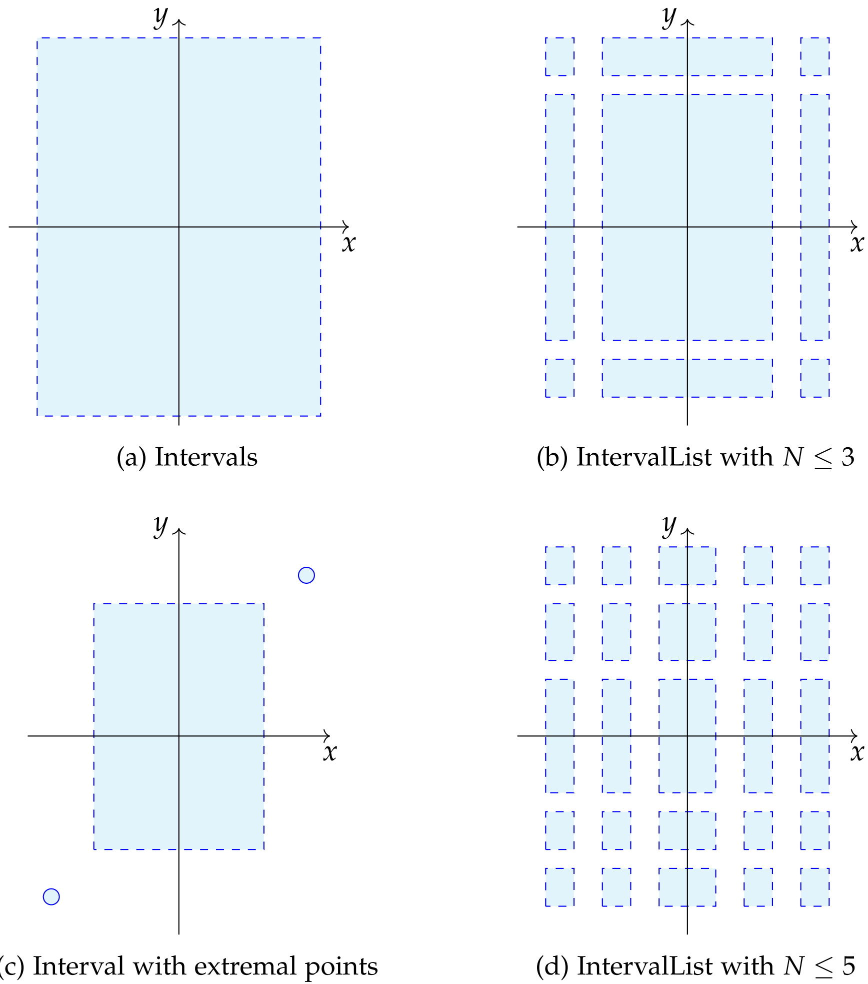
Listing 1.1: Illustratory code with usage of extremal values for representation of additional information. The floating point variable `random` is of range $[0,1[$. The input function `in_fun` can return four valid values $(0,1,2,3)$ and an invalid state (255). The output function `out_fn` checks for the extremal value and handles it differently. In practice, larger number and ranges are used.

*extremal values*. In performance-critical applications programmers often make domain-specific assumptions about the range of their data types and avoid the overhead of a more complex data type by using unused parts of the data type to store additional information. A common example is the use of the maximum value of a data type as an indication that a variable is in an invalid state (e.g. it was not initialized). Consider the simple example program in Listing 1.1 which takes an input value from a sensor via the input function `in_fn` and reports it via an output function `out_fn`. The input function uses the maximum of its range to symbolize that its input value has not been successfully read (here the maximum of `unsigned char` - 255).

This pattern appears frequently in low-level programming, but traditional abstract domains have problems handling this properly. For example, the *interval domain* approximates the range of `x` to $[0,255]$ and, thus after handling of the special value 255, the interval domain's approximation of $x$ is $[0,254]$. It is possible to use the *finite set domain* in this example ($\{0,1,2,3,255\}$), but in practice more numbers are used so that listing all potential values in a set would lead to a drastic increase in memory and runtime consumption. Therefore, in practice finite sets are only used with low cardinalities (e.g. 3 or 5). Another common solution for this problem *partitioning* variables of an analysis. However, when applied to $n$ variables the analysis space and time would grow exponentially and, thus partitioning can only be used to a limited degree in acceptable time and memory bounds.

Hence, there is an active interest from industry to improve abstract interpretation to efficiently cater for this specific programming pattern of extremal variables. This work looks at approaches for solving this problem. Figure 1.1 shows a visual representation of the abstract domains being considered. Specifically, we will focus on a *set of disjunctive intervals with a user-defined maximum cardinality $N$* as a general-purpose solution to this problem.

In Chapter 2 we provide notations used in this work, introduce the traditional interval abstract domain, and provide the principles of the Abstract Interpretation. We then analyze the formal properties of the new *IntervalList* abstract domain in Chapter 3. Afterwards, in Chapter 4 we discuss the individual operators and transformations of this domain. In Chapter 5 we assess its improvements over state-of-the-art abstract domains offered by the Astrée Static Analyzer [Cou+05; Cou+09] and present an evaluation of its impact on real-world industry codes. Finally, we summarize the findings in Chapter 6.




(a) Intervals

(b) IntervalList with $N \leq 3$

(c) Interval with extremal points

(d) IntervalList with $N \leq 5$

Figure 1.1: Visual examples of different improvements to the interval abstract domain (a) - the IntervalList abstract domain with a set of $N$ intervals (b, d) or a specialization with only a single main interval and two extremal points (c).

# 2 Interval abstract domain

In this section we give the notation of intervals and their operators as used in this work. Furthermore, we provide an overview of the interval abstract domain and its operations. Additionally, we summarize key Abstract Interpretation principles used in this and subsequent sections.

## 2.1 Numerical domain

We restrict the discussion to abstract *non-relational* domains operating on *finite integer* data types like `unsigned int` or `long long`. The respective finite integer domain is represented by $\mathbb{T}$ and its product forms the interval domain $\mathbb{I}$ ($\mathbb{T} \times \mathbb{T}$):

$$\begin{aligned} \mathbb{T} &= \{x \in \mathbb{Z} \mid T_{min} \leq x \leq T_{max}\} \\ \mathbb{I} &= \{[l, u] \mid l \in \mathbb{T}, u \in \mathbb{T}, l \leq u\} \end{aligned} \tag{2.1}$$

As a shortcut for *prime intervals* $[x, x]$ we write $\{x\}$. We will use $\underline{x}$ to denote the *lower* endpoint of an interval and $\overline{x}$ to denote the *upper* endpoint of an interval:

$$x = \{e \in \mathbb{Z} \mid \underline{x} \leq e \leq \overline{x}\} \tag{2.2}$$

For individual integer values, the *underline* and *overline* operators are set to the identity function ($\forall x \in \mathbb{T} : \underline{x} = \overline{x} = x$). We will restrict all further discussions to a program space of a single variable as non-relational domains do not consider the relationship between variables. However, it can easily be generalized as all variable states are independent.

## 2.2 Interval operations

Furthermore, we define the *cardinality* of an interval ($a \in \mathbb{I}$) as:

$$|a| \stackrel{\text{def}}{=} \overline{a} - \underline{a} \tag{2.3}$$

Note that $|a|$ can never be negative as such invalid intervals are disallowed by definition (Section 2.1). In addition, with definition 2.2 of the underline and overline operator above this cardinality applies to integers as well ($\forall x \in \mathbb{T} : |x| = 1$). For completeness, we define *existence* queries for $a \in \mathbb{I}, x \in \mathbb{T}$ as expected:

$$x \in a \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } \underline{a} \leq x \leq \overline{a} \\ \bot & \text{otherwise} \end{cases} \tag{2.4}$$

Moreover, to avoid ambiguity we define the *absolute* value of an interval with `abs` : $\mathbb{I} \mapsto \mathbb{I}$ as:

$$abs(a) \stackrel{\text{def}}{=} \begin{cases} [0, max(C)] & \text{if } 0 \in a \\ [min(C), max(C)] & \text{otherwise} \end{cases} \quad \text{with } C = \{abs(\underline{a}), abs(\overline{a})\} \tag{2.5}$$

Finally, an interval $a \in \mathbb{I}$ can be compared against a number $x \in \mathbb{T}$ with a comparison operator $\square_l = \{<, \leq\}$ and $\square_r = \{>, \geq\}$:

$$\begin{aligned} x \mathbin{\square_l} a &\stackrel{\text{def}}{=} x \mathbin{\square_l} \underline{a} & \qquad a \mathbin{\square_l} x &\stackrel{\text{def}}{=} \overline{a} \mathbin{\square_l} x \\ x \mathbin{\square_r} a &\stackrel{\text{def}}{=} x \mathbin{\square_r} \overline{a} & \qquad a \mathbin{\square_r} x &\stackrel{\text{def}}{=} \underline{a} \mathbin{\square_r} x \end{aligned} \tag{2.6}$$

## 2.3 Partial order

A partial ordering relation $\leq$ on a set $S$ (**poset**) is required to have the following three properties [DP02]:

1. *reflexive*: $\forall x \in S : x \leq x$
2. *transitive*: $\forall x, y, z \in S : x \leq y \wedge y \leq z \implies x \leq z$
3. *anti-symmetric*: $x, y \in S : x \leq y \wedge y \leq x \implies x = y$

A poset $\langle P, \leq \rangle$ is a *chain* if all elements are comparable ($\forall x, y \in P : x \leq y \vee y \leq x$). It is well-known that, for example, the set of integers ($\mathbb{Z}$) forms a chain [DP02]. This applies also for its subset $\mathbb{T}$. We consider this *natural order of intervals* ($x, y \in \mathbb{I}$) as defined in [SWH12]:

$$x \leq^I y \stackrel{\text{def}}{=} \underline{y} \leq \underline{x} \leq \overline{x} \leq \overline{y} \tag{2.7}$$

1) *Reflexivity*:

$$\begin{aligned} a \leq^I a &= (\underline{a} \leq \underline{a}) \wedge (\underline{a} \leq \overline{a}) \wedge (\overline{a} \leq \overline{a}) \quad (\leq \text{ is reflexive in } \mathbb{T}) \\ &= \text{true} \wedge (\underline{a} \leq \overline{a}) \wedge \text{true} \\ &= (\underline{a} \leq \overline{a}) \quad \text{(interval property)} \end{aligned}$$

2) *Transitivity*:

$$\begin{aligned} x \leq^I y \wedge y \leq^\# z &= (\underline{y} \leq \underline{x} \leq \overline{x} \leq \overline{y}) \wedge (\underline{z} \leq \underline{y} \leq \overline{y} \leq \overline{z}) \\ &= (\underline{y} \leq \underline{x} \leq \overline{x} \leq \overline{\mathbf{y}} \leq \overline{\mathbf{z}}) \wedge (\underline{z} \leq \underline{y} \leq \overline{y}) \\ &= \underline{\mathbf{z}} \leq \underline{\mathbf{y}} \leq \underline{x} \leq \overline{x} \leq \overline{y} \leq \overline{z} \quad (\leq \text{ is transitive in } \mathbb{T}) \\ &= \underline{z} \leq \underline{x} \leq \overline{x} \leq \overline{z} = x \leq z \end{aligned}$$

3) *Anti-symmetry*:

$$\begin{aligned} x \leq^I y \wedge x \leq^\# y &= (\underline{y} \leq \underline{x} \leq \overline{x} \leq \overline{y}) \wedge (\underline{x} \leq \underline{y} \leq \overline{y} \leq \overline{x}) \\ &= (\underline{y} \leq \underline{x} \wedge \underline{x} \leq \underline{y}) \wedge (\overline{y} \leq \overline{x} \wedge \overline{x} \leq \overline{y}) \quad (\leq \text{ is anti-symmetric in } \mathbb{T}) \\ &\implies (\underline{x} = \underline{y}) \wedge (\overline{x} = \overline{y}) \\ &= x = y \end{aligned}$$

## 2.4 Interval lattice

We recall the standard definition of a lattice [Bir48]. A poset $(S, \sqsubseteq)$ is a lattice $L$ if it has the following four properties:

I an *upper bound* (ub) $\forall x \in S : x \sqsubseteq ub$

II a *lower bound* (lb) $\forall x \in S : lb \sqsubseteq x$,

III a *least upper bound* (lub) of any $x, y \in S$, s.t. with
$x \sqcup y = a \in S : (x \sqsubseteq a \wedge y \sqsubseteq a) \wedge (\forall b \in S : x \sqsubseteq b \wedge y \sqsubseteq b \implies a \sqsubseteq b)$

IV a *greatest lower bound* (glb) of any $x, y \in S$, s.t. with
$x \sqcap y = a \in S : (a \sqsubseteq x \wedge a \sqsubseteq y) \wedge (\forall b \in S : b \sqsubseteq x \wedge b \sqsubseteq y \implies b \sqsubseteq a)$

Note that (I) and (II) can be easily achieved by adding a new *bottom element* $\bot$ ($\forall x \in S : \bot \sqsubseteq x$) and a new *top element* $\top$ ($\forall x \in S : x \sqsubseteq \top$) to $S$. We define this dedicated domain as $\mathcal{D}^I = \{\mathbb{I}, \bot, \top\}$. Additionally, note that for the integer interval domain $\top$ could be represented by $[T_{min}, T_{max}]$.

An illustratory representation of this partial order is given as a *Hasse diagram* in Figure 2.1. In a Hasse diagram order relations of a domain are depicted by edges. Every edge represents an $a \sqsubseteq b$ ($a, b \in \mathcal{D}^I$) relation of the partial order where in the Hasse graph $a$ is in a lower level than $b$. Transitive order relations are not displayed.

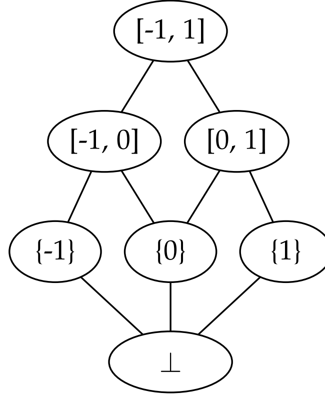



Figure 2.1: Hasse diagram of the interval abstract domain $(\mathcal{D}^I, \sqsubseteq)$ for the subset $\{-1, 0, 1\}$.

The following standard interval *join* and *meet* operations will be used to calculate the least upper bound (lub, $\sqcup$) and greatest lower bound (glb, $\sqcap$) on intervals ($a, b \in \mathbb{I}$):

$$
\begin{aligned}
a \sqcup b &\stackrel{\text{def}}{=} [\min(\underline{a}, \underline{b}), \max(\overline{a}, \overline{b})] \\
a \sqcap b &\stackrel{\text{def}}{=} \begin{cases} [\max(\underline{a}, \underline{b}), \min(\overline{a}, \overline{b})] & \text{if } \max(\underline{a}, \underline{b}) \leq \min(\overline{a}, \overline{b}) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
\tag{2.8}
$$

We extend the defined $\leq^I, \sqcup, \sqcap$ for $\mathcal{D}^I$ as $\leq_I^\#, \sqcup^\#, \sqcap^\#$ and add lifting for $\bot$ and $\top$:

$$a \leq_I^\# b \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } a = \top \wedge b = \top \\ \bot & \text{if } a = \bot \vee b = \bot \\ \bot & \text{if } a = \top \\ \top & \text{if } b = \top \\ a \leq^I b & \text{otherwise} \end{cases}$$
$$a \sqcup^\# b \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } a = \top \vee b = \top \\ \bot & \text{if } a = \bot \wedge b = \bot \\ b & \text{if } a = \bot \\ a & \text{if } b = \bot \\ a \sqcup b & \text{otherwise} \end{cases} \tag{2.9}$$
$$a \sqcap^\# b \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } a = \top \wedge b = \top \\ \bot & \text{if } a = \bot \vee b = \bot \\ b & \text{if } a = \top \\ a & \text{if } b = \top \\ a \sqcap b & \text{otherwise} \end{cases}$$

Hence, the poset $\mathcal{D}^I(\leq_I^\#)$ can form an *interval lattice* $L_I(\leq_I^\#, \top, \bot, \sqcup^\#, \sqcap^\#)$. The *concrete domain* will be noted as $\mathcal{D}^C$ and for this finite integer domain it is given by $\mathbb{T}$. Similarly, its concrete lattice with poset $\mathcal{D}^C(\subseteq)$ is given by $L_C(\subseteq, \mathbb{T}, \emptyset, \cup, \cap)$. Note that both lattices are *complete lattices* - a lattice in which all subsets have a lub and a glb - as every finite lattice is complete [Bir48].

## 2.5 Abstract relationship

In general, we establish a correspondence between the concrete domain $(\mathcal{D}^C, \subseteq)$ and an abstract domain $(\mathcal{D}^\#, \sqsubseteq)$ with an *abstraction* function $\alpha : \mathcal{D}^C \mapsto \mathcal{D}^\#$ and a *concretization* function $\gamma : \mathcal{D}^\# \mapsto \mathcal{D}^C$. Together they form the function pair $\langle \alpha, \gamma \rangle$: $(\mathcal{D}^C, \subseteq) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} (\mathcal{D}^\#, \sqsubseteq)$. Any abstract property $a \in \mathcal{D}^\#$ is a *sound* approximation of a concrete property $x \in \mathcal{D}^C$ if $\alpha(x) \sqsubseteq a$. Moreover, soundness of an abstract property $a \in \mathcal{D}^\#$ requires the concretization to fulfill: $x \subseteq \gamma(a)$. Hence, an abstraction-concretization correspondence is *sound* if $\forall x \in \mathcal{D}^C : x \subseteq \gamma(\alpha(x))$ [CC76].

A *Galois connection* is defined between two complete lattices $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ via a function pair $\langle \alpha, \gamma \rangle$ and requires three properties [CC79]:

1. monotonic $\alpha \in (\mathcal{D}_1 \mapsto \mathcal{D}_2)$
2. monotonic $\gamma \in (\mathcal{D}_2 \mapsto \mathcal{D}_1)$
3. $\forall x \in \mathcal{D}_1 : \forall y^\# \in \mathcal{D}_2 : \alpha(x) \sqsubseteq_2 y^\# \iff x \sqsubseteq_1 \gamma(y^\#)$

In Abstraction Interpretation a Galois connection allows for the establishment of a formal relationship between the concrete domain lattice $(\mathcal{D}^C, \subseteq)$ of a program and an abstract domain lattice $(\mathcal{D}^\#, \sqsubseteq)$. Note that it is only necessary to define either the abstraction $\alpha$ or the concretization $\gamma$ for a Galois connection as $\alpha$ and $\gamma$ can be synthesized from each other [CC92a, Prop. 5]:

$$\alpha(x) \stackrel{\text{def}}{=} \bigsqcap\{y \ \in \mathcal{D}^I \mid x \subseteq \gamma(y)\}$$
$$\gamma(y) \stackrel{\text{def}}{=} \bigsqcup\{x \ \in \mathcal{D}^C \mid \alpha(x) \leq_I^\# y\}$$

We will now consider the interval domain with $(\mathcal{D}^C, \subseteq)$ as concrete domain and $(\mathcal{D}^I, \leq_I^\#)$ as abstract domain and setup a Galois connection. We establish the correspondence between concrete and abstract values with an abstraction function $\alpha : \mathcal{D}^C \mapsto \mathcal{D}^I$:

$$\alpha(X) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } X = \emptyset \\ \top & \text{if } X = \mathbb{T} \\ [min(X), max(X)] & \text{otherwise} \end{cases}$$

Note, that with the definition of $\sqcup$ for intervals (Section 2.4) this implies the following concretization function $\gamma : \mathcal{D}^I \mapsto \mathcal{D}^C$:

$$\gamma(Y) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } Y = \bot \\ \{x \in \mathbb{T}\} & \text{if } Y = \top \\ \{x \in \mathbb{T} \mid \underline{Y} \leq x \leq \overline{Y}\} & \text{otherwise} \end{cases}$$

The *best* approximation is an approximation function which performs the most precise approximation for its domain (see [CC92b, p. 4.24]). The approximation $\alpha$ of the interval domain is the *best* approximation (implied by the Galois connection).

## 2.6 Interval abstract operators

The interval domain in Astrée has been implemented by Miné and most details are provided in [Min04]. As a quick summary we list its abstract operators. Its best join and meet operators are the lub and glb of the interval domain:

$$a^\# \cup^\# b^\# \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } a^\# = \top \wedge b^\# = \top \\ a^\# \sqcup b^\# & \text{if } a^\# \neq \bot \wedge b^\# \neq \bot \\ b^\# & \text{if } a^\# \neq \bot \\ a^\# & \text{if } b^\# \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$a^\# \cap^\# b^\# \stackrel{\text{def}}{=} \begin{cases} b^\# & \text{if } a^\# = \top \wedge b^\# \neq \bot \\ a^\# & \text{if } a^\# \neq \bot \wedge b^\# = \top \\ a^\# \sqcap b^\# & \text{if } a^\# \neq \bot \wedge b^\# \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

### 2.6.1 Forward operators

*Forward semantic operators* determine - given input states with invariants - invariants for the *resulting* states [CC79]. We list all in system programming languages commonly available unary and binary arithmetic operators. However, we omitted handling of the special $\top$ and $\bot$ elements for the individual operations as they can be lifted as follows:

$$\Box^\# a^\# \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } a^\# = \bot \\ \top & \text{if } a^\# = \top \\ \Box^{\#'} a^\# & \text{otherwise} \end{cases}$$

$$a^\# \Box^\# b^\# \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } a^\# = \bot \vee b^\# = \bot \\ \top & \text{if } a^\# = \top \vee b^\# = \top \\ a^\# \Box^{\#'} b^\# & \text{otherwise} \end{cases}$$

where $\Box^{\#'}$ is the respective arithmetic operation before lifting.

#### Forward arithmetic operators

For a monotone n-ary operator $f : \mathbb{T}^n \mapsto \mathbb{T}$ in the concrete domain a *sound abstract transformer* is any $f^\# : (\mathcal{D}^I)^n \mapsto \mathcal{D}^I$ for which $\forall(a_1, ..., a_n \in (\mathcal{D}^I)^n) : \alpha(f(\gamma(a_1), ..., \gamma(a_n))) \sqsubseteq$

$f^\#(a_1, ..., a_n)$ or $f(\gamma(a_1), ..., \gamma(a_n)) \subseteq \gamma(f^\#(a_1, ..., a_n))$.

For binary operators the forward relation is `r = a □ b` and the transfer operator gives an `r`. For unary operators the relation is `r = □ a`. The unary and binary sound abstract transformers of the interval domain are:

$$
\begin{aligned}
-^\# a^\# &\stackrel{\text{def}}{=} [-\overline{a^\#}, -\underline{a^\#}] \\
a^\# +^\# b^\# &\stackrel{\text{def}}{=} [\underline{a^\#} + \underline{b^\#}, \overline{a^\#} + \overline{b^\#}] \\
a^\# -^\# b^\# &\stackrel{\text{def}}{=} [\underline{a^\#} - \overline{b^\#}, \overline{a^\#} - \underline{b^\#}] \\
a^\# \times^\# b &\stackrel{\text{def}}{=} [\min(C^\#), \max(C^\#)] \text{ with } C^\# = \{\underline{a^\#} \times \underline{b^\#}, \underline{a^\#} \times \overline{b^\#}, \overline{a^\#} \times \underline{b^\#}, \overline{a^\#} \times \overline{b^\#}\} \\
a^\# /^\# b^\# &\stackrel{\text{def}}{=} \begin{cases} \top & \text{if } 0 \in b^\# \\ (a^\# \text{ div}^\# \ (b \sqcap [T_{min}, -1]^\#) \sqcup (a^\# \text{ div}^\# \ (b^\# \sqcap [1, T_{max}]^\#) & \text{otherwise} \end{cases} \\
a \text{ div}^\# \ b &\stackrel{\text{def}}{=} [\min(C^\#), \max(C^\#)]^\# \text{ with } C^\# = \{\underline{a^\#}/\underline{b^\#}, \underline{a^\#}/\overline{b^\#}, \overline{a^\#}/\underline{b^\#}, \overline{a^\#}/\overline{b^\#}\} \\
a^\# \ \%^\# \ b^\# &\stackrel{\text{def}}{=} \begin{cases} \top & \text{if } 0 \in b^\# \\ \underline{a^\#} \ \% \ \underline{b^\#} & \text{if } |a^\#| = 1 \wedge |b^\#| = 1 \\ a^\# & \overline{abs(a^\#)} < \underline{b^\#} \\ [0, \overline{abs(b^\#)} - 1] & \text{if } \underline{a^\#} \geq 0 \\ [-\overline{abs(b^\#)} + 1, 0] & \text{if } \overline{a^\#} \leq 0 \\ [-\overline{abs(b^\#)} + 1, \overline{abs(b^\#)} - 1] & \text{otherwise} \end{cases}
\end{aligned}
$$

**Logical bitwise AND**

$$
a^\# \ \&^\# \ b^\# \stackrel{\text{def}}{=} \begin{cases} \underline{a}^\# \ \& \ \underline{b}^\# & \text{if } |a^\#| = 1 \wedge |b^\#| = 1 \\ [0, \min(\overline{a}^\#, \overline{b}^\#)] & \text{if } a^\# \leq [0,1] \wedge b^\# \leq [0,1] \\ a^\# \ \texttt{logand} \ b^\# & \text{if } 0 \leq a^\# \wedge 0 \leq b^\# \\ \top & \text{otherwise} \end{cases}
$$

$$
a^\# \ \texttt{logand} \ b^\# = \begin{cases} [0, a^\# \ \& \ fill(b^\#)] & \text{if } |a^\#| = 1 \\ [0, fill(a^\#) \ \& \ b^\#] & \text{if } |b^\#| = 1 \\ [0, fill(\min(\overline{a}^\#, \overline{b}^\#))] & \text{otherwise} \end{cases}
$$

$$
fill(x) = \begin{cases} 1 & \text{if } y = 0 \\ 2^{y+1} - 1 & \text{if } 2^y = x \\ 2^y - 1 & \text{otherwise} \end{cases} \quad \text{with } y = \lceil log_2 \ x \rceil
$$

**Logical bitwise OR**

$$a^\# \mid^\# b^\# \stackrel{\text{def}}{=} \begin{cases} \underline{a}^\# \mid \underline{b}^\# & \text{if } |a^\#| = 1 \wedge |b^\#| = 1 \\ [\max(\underline{a}^\#, \underline{b}^\#), \max(\overline{a}^\#, \overline{b}^\#)] & \text{if } a^\# \leq [0,1] \wedge b^\# \leq [0,1] \\ a^\# \ \texttt{logor}\ b^\# & \text{if } 0 \leq a^\# \wedge 0 \leq b^\# \\ \top & \text{otherwise} \end{cases}$$

$$a^\# \ \texttt{logor}\ b^\# = \begin{cases} [\underline{a}^\#, c^\#] & \text{if } |a^\#| = 1 \\ [\underline{b}^\#, c^\#] & \text{if } |b^\#| = 1 \\ [0, c^\#] & \text{otherwise} \end{cases} \text{ with } c^\# = \mathit{fill}(\max(\overline{a}^\#, \overline{b}^\#))$$

**Logical bitwise XOR**

$$a^\# \ \oplus^\# \ b^\# \stackrel{\text{def}}{=} \begin{cases} \underline{a}^\# \ \oplus \ \underline{b}^\# & \text{if } |a^\#| = 1 \wedge |b^\#| = 1 \\ [\min(\underline{a}^\#, \underline{b}^\#), \max(\overline{a}^\#, \overline{b}^\#)] & \text{if } a^\# \leq [0,1] \wedge b^\# \leq [0,1] \\ [0, \mathit{fill}(\max(\overline{a}^\#, \overline{b}^\#))] & \text{if } 0 \leq a^\# \wedge 0 \leq b^\# \\ \top & \text{otherwise} \end{cases}$$

**Left and right shift**

`max_shift` is the maximum valid allowed shift size for a specific variable type (typically $\lceil log_2(T_{max} - T_{min}) \rceil - 1$).

$$a^\# \ll b^\# \stackrel{\text{def}}{=} \begin{cases} [min(C^\#), max(C^\#)] & \text{if } b^\# \leq [0, \texttt{max_shift}] \\ & \quad \text{with } C^\# = \{\underline{a^\#}\ll\underline{b^\#}, \underline{a^\#}\ll\overline{b^\#}, \overline{a^\#}\ll\underline{b^\#}, \overline{a^\#}\ll\overline{b^\#}\} \\ \top & \text{otherwise} \end{cases}$$

$$a^\# \gg^\# b^\# \stackrel{\text{def}}{=} \begin{cases} [min(C^\#), max(C^\#)] & \text{if } b^\# \leq [0, \texttt{max_shift}] \\ & \quad \text{with } C^\# = \{\underline{a^\#}\gg\underline{b^\#}, \underline{a^\#}\gg\overline{b^\#}, \overline{a^\#}\gg\underline{b^\#}, \overline{a^\#}\gg\overline{b^\#}\} \\ \top & \text{otherwise} \end{cases}$$

### 2.6.2 Backward operators

*Backward semantic operators* determine - given input states with invariants - the *prior* invariants. They are typically evaluated during a backward program analysis with an inverted program graph [CC79]. For binary operators the forward relation was `r = a` $\square$ `b` and the backward transfer operator is now - given an approximation of `r` - refines $a^\#$ and $b^\#$ further. Hence, a binary backward operator returns the duplet $\mathcal{D}^I \times \mathcal{D}^I$ for

the refined $a^\#, b^\#$. Analogous for unary backward operators the relation was `r = □ a` and only $a^\#$ is refined. Miné [Min04] presented a mechanical way for generic backward arithmetic operator synthesis based on given forward operators:

$$
\begin{aligned}
\overleftarrow{-}^{\#}\,(a^\#, r^\#) &\stackrel{\text{def}}{=} (a^\# \cap (-^\# r^\#)) \\
\overleftarrow{+}^{\#}\,(a^\#, b^\#, r^\#) &\stackrel{\text{def}}{=} (a^\# \cap^\# (r^\# -^\# b^\#),\ b^\# \cap^\# (r^\# -^\# a^\#)) \\
\overleftarrow{-}^{\#}\,(a^\#, b^\#, r^\#) &\stackrel{\text{def}}{=} (a^\# \cap^\# (b^\# +^\# r^\#)),\ b^\# \cap^\# (a^\# -^\# r^\#)) \\
\overleftarrow{\times}^{\#}\,(a^\#, r^\#, b^\#) &\stackrel{\text{def}}{=} (a^\# \cap^\# (r^\# /^\# b^\#),\ b^\# \cap^\# (r^\# /^\# a^\#)) \\
\overleftarrow{/}^{\#}\,(a^\#, b^\#, r^\#) &\stackrel{\text{def}}{=} (a^\# \cap^\# (b^\# \times^\# (r^\# +^\# [-1,1]^\#)), \\
&\qquad b^\# \cap^\# ((a^\# /^\# (r^\# +^\# [-1,1]^\#)) \cup^\# [0,0]^\#)) \\
\overleftarrow{\%}^{\#}\,(a^\#, b^\#, r^\#) &\stackrel{\text{def}}{=} (a^\#,\ b^\#) \\
\overleftarrow{\&}^{\#}\,(a^\#, b^\#, r^\#) &\stackrel{\text{def}}{=} (a^\#,\ b^\#) \\
\overleftarrow{|}^{\#}\,(a^\#, b^\#, r^\#) &\stackrel{\text{def}}{=} (a^\#,\ b^\#) \\
\overleftarrow{\oplus}^{\#}\,(a^\#, b^\#, r^\#) &\stackrel{\text{def}}{=} (a^\# \cap^\# (r^\# \ \oplus^\# \ b^\#), r^\# \ \oplus^\# \ a^\#) \\
\overleftarrow{«}^{\#}\,(a^\#, b^\#, r^\#) &\stackrel{\text{def}}{=} (a^\# \cap^\# (r^\# »^\# b^\#),\ b^\#) \\
\overleftarrow{»}^{\#}\,(a^\#, b^\#, r^\#) &\stackrel{\text{def}}{=} (a^\# \cap^\# ((r^\# «^\# b^\#) +^\# [0, 2^{b^\#} - 1]^\#),\ b^\#)
\end{aligned}
\tag{2.10}
$$

### Backward tests

*Backward tests* allow refinement of variables based on assertions in code. Consider the simple example from Listing 2.1. The invariant $x^\# > 10$ allows the interpreter to refine $x^\#$ in both code branches:

Listing 2.1: Illustratory example for backward comparison refinement

```
if (x > 10) {
  // x ∩ [11,∞]
} else {
  // x ∩ [−∞,10]
}
```

Whenever a backward test yields $\perp$, the abstract interpreter can mark that code branch as unreachable for the current context. A backward comparison test maps an abstract domain duplet $\mathcal{D}^I \times \mathcal{D}^I \mapsto \mathcal{D}^I \times \mathcal{D}^I$ to a duplet which gives the refinement if the comparison was truthy.

$$a^\# \overset{\leftarrow}{=} b^\# \overset{\text{def}}{=} (c^\#, c^\#) \quad \text{with } c^\# = a^\# \sqcap b^\#$$

$$a^\# \overset{\leftarrow}{\leq} b^\# \overset{\text{def}}{=} (a^\# \sqcap [T_{min}, \overline{b^\#}]^\#, b^\# \sqcap [\underline{a}^\#, T_{max}]^\#)$$

$$a^\# \overset{\leftarrow}{<} b^\# \overset{\text{def}}{=} (a^\# \sqcap [T_{min}, \overline{b^\#} - 1]^\#, b^\# \sqcap [\underline{a}^\# + 1, T_{max}]^\#)$$

$$a^\# \overset{\leftarrow}{\neq} b^\# \overset{\text{def}}{=} \begin{cases} (\bot, \bot) & \text{if } |a| = 1 \wedge |b^\#| = 1 \wedge \underline{a}^\# = \underline{b}^\# \\ (a^\#, [\underline{b}^\# + 1, \overline{b}]^\#) & \text{if } |a^\#| = 1 \wedge \underline{a}^\# = \underline{b}^\# \\ (a^\#, [\underline{b}^\#, \overline{b}^\# - 1]^\#) & \text{if } |a^\#| = 1 \wedge \underline{a}^\# = \overline{b}^\# \\ ([\underline{a}^\# + 1, \overline{a}^\#]^\#, b^\#) & \text{if } |b^\#| = 1 \wedge \underline{a}^\# = \underline{b}^\# \\ ([\underline{a}^\#, \overline{a}^\# - 1]^\#, b^\#) & \text{if } |b^\#| = 1 \wedge \overline{a}^\# = \underline{b}^\# \\ (a^\#, b^\#) & \text{otherwise} \end{cases}$$

By duality, the comparison can be inverted if refinement for falsy is desired.

## 2.7 Fixpoint approximation

A *fixpoint* is a point $x$ of a poset $S$ with a monotone mapping function $f : S \mapsto S$ for which $f(x) = x$. We write $FP$ for the set of fixpoints $\{x \in S \mid f(x) = x\}$ of $f$. Then, the *least fixpoint* (lfp) $x$ of $f$ is the fixpoint for which (1) $\forall y \in FP : x \sqsubseteq y$ and (2) $\forall y \in FP : y \not\sqsubseteq x$. Analogous, the *greatest fixpoint* (gfp) $x$ of $f$ is the fixpoint for which (1) $\forall y \in FP : y \sqsubseteq x$ and (2) $\forall y \in FP : x \not\sqsubseteq y$. Additionally, the set of *post-fixpoints* is $\{y \in S \mid f(y) \sqsubseteq y\}$ and the set of *pre-fixpoints* is $\{y \in S \mid y \sqsubseteq f(y)\}$ [CC92a].

### 2.7.1 Widening

For Abstract Interpretation loops are problematic as the evaluation might not terminate if the program state iteration chain $\bot \sqsubseteq f(\bot) \sqsubseteq ... \sqsubseteq f^n(\bot) \sqsubseteq ...$ is infinite.

*Widening* is an *extrapolation* technique which *over-approximates fixpoints* in increasing infinite chains. This over-approximation *accelerates convergence* of an increasing chain and for infinite domains *enforces termination* of the abstract evaluation in finite many iteration steps.

For two complete lattices $D_1$ and $D_2$ which are linked by a Galois connection $\langle \alpha, \gamma \rangle$: $(\mathcal{D}_1, \subseteq) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} (\mathcal{D}_2, \sqsubseteq)$, we know that their their fixpoints have the relation $\text{lfp}(D_1) \subseteq \gamma(\text{lfp}(D_2))$ [CC77]. Hence, a lfp in the abstract domain ($D_2$) will be a sound representation of the program state.

A sound approximation of a lfp in the abstract domain ($D_2$) can be computed with widening. Widening was first introduced by [CC76] and transforms any chain $X^i$ into an increasing chain $Y^i$:

$$Y^i \stackrel{\text{def}}{=} \begin{cases} Y^{i-1} \ \sqcup\!\!\!\!\sqcup \ X^i & \text{if } i > 0 \\ X^i & \text{otherwise} \end{cases}$$

The widening operator $\sqcup\!\!\!\!\sqcup$: $\mathcal{D}^\# \times \mathcal{D}^\# \mapsto \mathcal{D}^\#$ must have two properties:

1. $\forall a, b \in \mathcal{D}^\# : (a \cup b) \leq (a \ \sqcup\!\!\!\!\sqcup \ b)$
2. $n \in [0, z] : Y^{n+1\#} = Y^{n\#} \quad z \in \mathbb{N} : z < \infty$ (not strictly increasing)

The simplest form of interval widening (e.g. given in [SWH12]) is done by extending the domain $\mathcal{D}^I \cup \{-\infty, \infty\} = \mathcal{D}^J$. Note this results in $[-\infty, \infty]$ which could replace the synthetic $\top$ element. Furthermore, all abstract operations need to be adjusted as well. As an example, the partial relation $\leq^\#$ would need to be extended to $\mathcal{D}^J \times \mathcal{D}^J \mapsto \mathcal{D}^J$:

$$a \leq^\# b \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } b = \infty \vee a = -\infty \\ \bot & \text{if } a = \infty \vee b = -\infty \\ a \leq^{\#'} b & \text{otherwise} \end{cases}$$

where $\leq^{\#'}$ is the previous definition of $\leq^\#$.

Subsequently, the widening operator $\sqcup\!\!\!\!\sqcup$: $\mathcal{D}^J \times \mathcal{D}^J \mapsto \mathcal{D}^J$ can be defined as:

$$a \sqcup\!\!\!\!\sqcup b \stackrel{\text{def}}{=} \begin{cases} b & \text{if } a = \bot \\ a & \text{if } b = \bot \\ [l, u] & \text{otherwise} \end{cases} \quad \text{where}$$

$$l = \begin{cases} \underline{a} & \text{if } \underline{a} \leq^\# \underline{b} \\ -\infty & \text{otherwise} \end{cases}$$

$$u = \begin{cases} \overline{a} & \text{if } \overline{a} \geq^\# \overline{b} \\ \infty & \text{otherwise} \end{cases}$$

*Delayed widening* is a trick to improve widening which delays the over-approximating impact of widening steps for fixed set of delay points $DP$ $\{x \in \mathbb{N}\}$ and allows to potentially reduce precision loss of too eager over-approximation:

$$Y^i \stackrel{\text{def}}{=} \begin{cases} Y^{i-1} \sqcup X^i & \text{if } i > 0 \wedge i \in DP \\ Y^{i-1} \sqcup\!\!\!\!\nabla X^i & \text{if } i > 0 \\ X^i & \text{otherwise} \end{cases}$$

The delay points $DP$ highly depend on the analyzed code and are a trade-off between analysis duration and potential higher precision.

### 2.7.2 Narrowing

*Narrowing* as first introduced in [CC76] is another extrapolation technique which attempts to improve the post-fixpoint yielded by widening through *downward iterations*. Given a narrowing operator $\sqcap\!\!\!\!\Delta$: $\mathcal{D}^\# \times \mathcal{D}^\# \mapsto \mathcal{D}^\#$ with following two properties:

1. $\forall a, b \in \mathcal{D}^\# : (b \leq a) \Rightarrow (b \leq (a \sqcap\!\!\!\!\Delta\, b) \leq a)$
2. $n \in [0, z] : Y^{n+1\ \#} = Y^{n\ \#} \quad z \in \mathbb{N} : z < \infty$ (not strictly decreasing)

it is guaranteed that narrowing does not reduce the abstract representation below $\text{lfp}(D_1) \leq \gamma(Y^{n\ \#})$ [CC92a, Prop. 30]. The simplest form of interval narrowing is:

$$a \sqcap\!\!\!\!\Delta\, b \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } a = \bot \vee b = \bot \\ [l, u] & \text{otherwise} \end{cases} \quad \text{where}$$

$$l = \begin{cases} \underline{b} & \text{if } \underline{a} = -\infty \\ \underline{a} & \text{otherwise} \end{cases}$$

$$u = \begin{cases} \overline{b} & \text{if } \overline{a} = \infty \\ \overline{a} & \text{otherwise} \end{cases}$$

However, for finite integer abstract domains narrowing is useless [Ber+10] and we set:

$$X^\# \sqcap\!\!\!\!\Delta\, Y^\# \stackrel{\text{def}}{=} X^\# \cap^\# Y^\#$$

# 3 IntervalList abstract domain

A generalization of the interval abstract domain is to use a *set of disjunctive intervals* which will be discussed in this section. Due to its underlying implementation we call this abstract domain the *IntervalList domain*. We start by providing a formal definition of the IntervalList abstract domain, explore other buildings blocks of this domain (ordering, normalization procedure) and finally discuss properties of the domain.

## 3.1 Definition

An *interval order* is a linearly ordered set of intervals. Interval orders have one problematic property. They can have many distinct representations (e.g. $\{[1,2],[2,4]\} = \{[1,3],[3,4]\} = \{[1,4]\}$). Hence, we first need to definite a stricter representation, s.t. there can only be one *distinct* representation. We define a special IntervalList domain $\mathbb{E}$ which consists of an *linearly ordered set of disjunctive intervals*. The set cardinality cannot be zero and must not be greater than the user-defined maximal cardinality $N \in \mathbb{N}$. The space of $\mathbb{E}$ is $\mathbb{I}^N$. Furthermore, we require a (1) linear order and (2) intervals to neither overlap nor be directly adjacent to ensure a unique representation:

$$
\begin{aligned}
\mathbb{E} = \{e_1, \dots, e_k\} \quad & e_i \in \mathbb{I}, k \in \mathbb{N} : 1 \leq k \leq N \\
& \forall i \in [1, k-1] : \overline{e_i} + 1 < \underline{e_{i+1}}
\end{aligned}
$$

Note that these properties require a specific interval order which results in a unique, distinctive representation as it is no longer possible to form overlapping or adjacent intervals (e.g. $\{[1,4]\}$ cannot be represented as $[1,2],[3,4]$). Hence, we have a list of intervals which is non-overlapping and thus have a linear order. Figure 3.1 shows examples for the individual conditions. The cardinality of this set of intervals must be limited by a constant as otherwise it would grow arbitrarily on every union.

Additionally, we define an unrestricted set of intervals $\mathbb{E}_r = \{e_1, \dots, e_n\}$ without these properties to represent non-normalized lists of intervals. Furthermore, a special bottom and top element are added to construct the abstract domain $\mathcal{D}^E = \{\bot, \mathbb{E}, \top\}$

and $\mathcal{D}^{ER} = \{\bot, \mathbb{E}_r, \top\}$ respectively. Note that this extension is not strictly necessary as $\emptyset$ could have been used as the bottom element and $[T_{min}, T_{max}]$ as the top element, but we opted to use these additional elements as Astrée internally provides interfaces which expect separate $\top$ and $\bot$ elements.

Note that we opted to not include $\infty$ in this representation as (1) in Astrée the interval domain is already using $\pm\infty$ and it would only provide a small additional benefit if either of the ranges is $\pm\infty$, (2) this domain is intended to preserve the extremal values before infinity, and (3) Astrée uses arbitrary-sized integer in its implementation which allows to use $[T_{min}, T_{max}]$ without overflow risks.

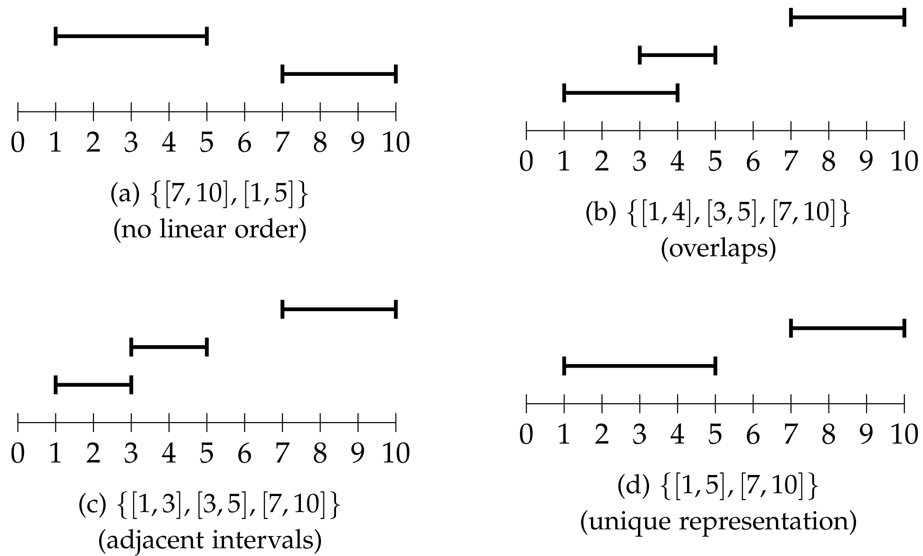



Figure 3.1: Different, non-unique representations of the interval $\{[1,5],[7,10]\}$. (a) is not a linear order, (b) has overlapping intervals, (c) has directly adjacent intervals, (d) is the only and unique representation in this domain.

## 3.2 Partial order

For a better distinction between intervals and lists of intervals, $\leq$ will be used for the partial order of intervals (see Section 2.3) and $\sqsubseteq$ for the partial order of lists of intervals. To construct this partial order, we define a $\sqsubseteq$ relation on $\mathcal{D}^E \times \mathcal{D}^E$ that checks whether all intervals in one list are fully included in one respective interval in the other list:

$$A \sqsubseteq B \stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } A = \bot \vee B = \top \\ \text{false} & \text{if } A = \top \vee B = \bot \\ \forall a \in A : \exists b \in B : a \leq^{\#} b & \text{otherwise} \end{cases} \tag{3.1}$$

Note that as there cannot be two directly adjacent intervals, there can only be at most one interval $b \in B$ that fully includes an interval of $A$. Two neighboring intervals of $B$ need to have a gap of at least 1 and thus can not fully include an interval. The $\sqsubseteq$ relation between list of intervals is a partial order as it fulfills the three properties of a partial order:

**1. Reflexivity**

By definition, $\bot \sqsubseteq \bot$ and $\top \sqsubseteq \top$. For all other elements ($A \neq \bot \wedge A \neq \top$), there exists exactly one element $e$ in $A$ which contains $a$ ($a$ itself). An interval cannot be a subset of another interval, as this would require overlapping intervals which are excluded.

$$\begin{aligned} A \sqsubseteq^{\#} A &= (\forall a \in A : a \leq^{\#} a) \quad \text{Interval reflexivity} \\ &= (\forall a \in A : \text{true}) \end{aligned}$$

**2. Transitivity**

$$\begin{aligned} A \sqsubseteq^{\#} B \wedge B \sqsubseteq^{\#} C &= (\forall a \in A : \exists b \in B : a \leq^{\#} b) \wedge (\forall b \in B : \exists c \in C : b \leq^{\#} c) \\ &= (\forall a \in A : \exists b \in B : (\forall b\ \exists c \in C) : a \leq^{\#} b \wedge b \leq^{\#} c \\ &= (\forall a \in A : \exists b \in B, \exists c \in C : a \leq^{\#} b \wedge b \leq^{\#} c \\ &= (\forall a \in A : \exists b \in B, \exists c \in C : a \leq^{\#} b \leq^{\#} c \quad \text{Interval transivitity} \\ &\implies (\forall a \in A : \exists c \in C : a \leq^{\#} c) \end{aligned}$$

**3. Anti-symmetry**

$$\begin{aligned}
A \sqsubseteq^\# B \land B \sqsubseteq^\# A &= (\forall a \in A : \exists b \in B : a \leq^\# b) \land (\forall b \in B : \exists a \in A : b \leq^\# a) \\
&= \forall a \in A : (\exists b \in B : a \leq^\# b \land \exists c \in A : b \leq^\# c) \land \\
&\quad \forall b \in b : (\exists a \in A : b \leq^\# a \land \exists d \in B : a \leq^\# d) \\
&= \forall a \in A : (\exists b \in B, \exists c \in A : a \leq^\# b \leq^\# c) \land \\
&\quad \forall b \in B : (\exists a \in A, \exists d \in B : b \leq^\# a \leq^\# d)
\end{aligned}$$

As an interval order cannot contain overlapping intervals, $a \leq^\# c$ is only possible iff $c = a$ and thus $b = a$. This applies to all $b$ for all intervals in $A$. Similarly, $b \leq^\# d$ is only possible iff $b = d$ and thus $a = b$. This applies to all $a$ for all intervals in $B$. Hence, all intervals must be identical and both lists are identical ($A = B$).

## 3.3 Lower and upper bound

Lower and upper bounds exist on a list of intervals as well. By the linear order of the list, the lower bound is the lower bound of the first element and the upper bound is the upper bound of the last element in the list. Together they define the *hull* of an IntervalList ($X \in \mathcal{D}^E$):

$$\begin{aligned}
\underline{X} &\stackrel{\text{def}}{=} \underline{\{e_1, \ldots, e_N\}} = \underline{e_1} \\
\overline{X} &\stackrel{\text{def}}{=} \overline{\{e_1, \ldots, e_N\}} = \overline{e_N} \\
\text{hull}(X) &\stackrel{\text{def}}{=} [\underline{X}, \overline{X}]
\end{aligned}$$

## 3.4 Hasse diagram

To obtain a better intuitive understanding of this partial order, a Hasse diagram for a small instance of this domain is provided in Figure 3.2 for a Hasse diagram of $(\{-1, 0, 1\}, \sqsubseteq)$ and Figure 3.3 for $(\{0, 1, 2, 3\}, \sqsubseteq)$. Figure 3.2 is very similar to a corresponding interval Hasse diagram from Figure 2.1 (only the center node $\{\{-1\}, \{1\}\}$ is different). Nodes which exist in the *corresponding interval* Hasse diagram have an *ellipse shape* (interval lists with one interval), new nodes that *cannot occur in the corresponding* interval Hasse diagram use a *box shape* (interval lists with more than one interval).

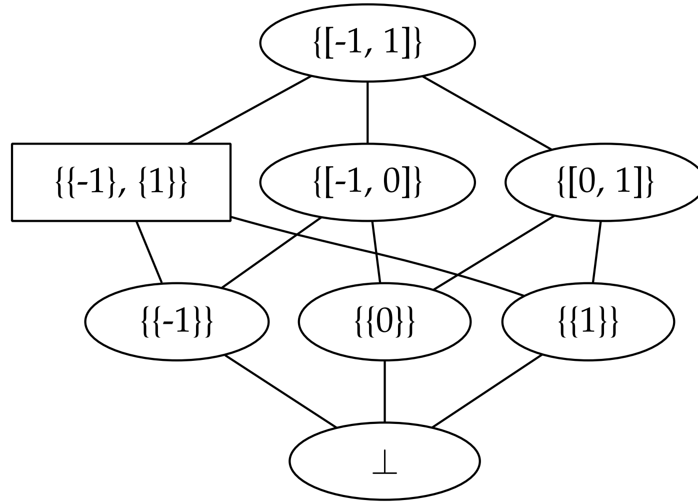



Figure 3.2: Hasse diagram of the IntervalList abstract domain $(\mathcal{D}^E, \sqsubseteq)$ for $\{-1, 0, 1\}$

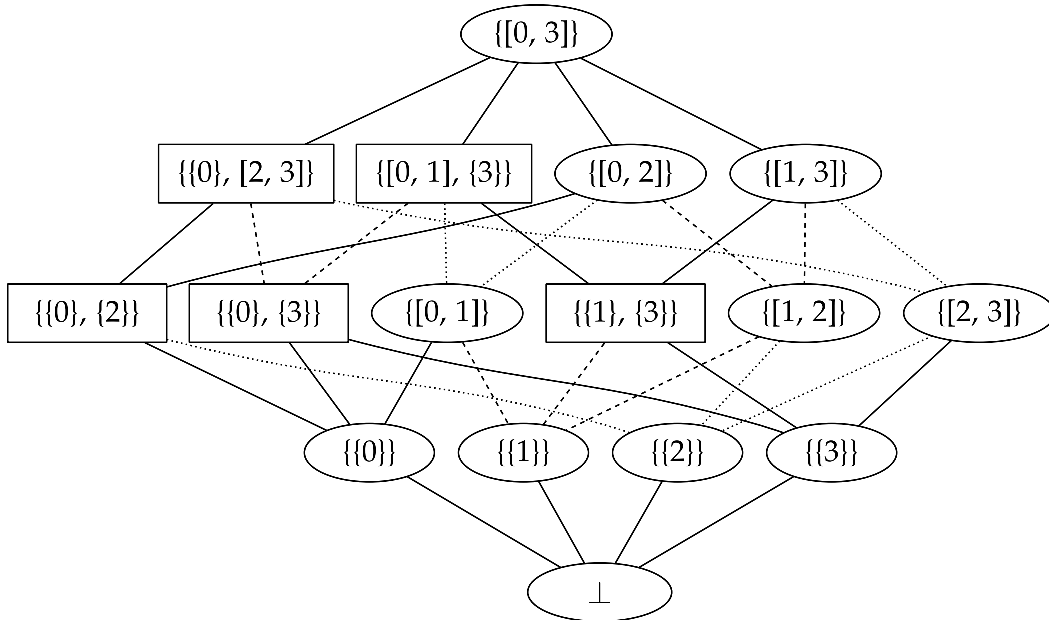



Figure 3.3: Hasse diagram of the IntervalList abstract domain $(\mathcal{D}^E, \sqsubseteq)$ for $\{0, 1, 2, 3\}$

## 3.5 Normalization

It is possible for operations to add or modify intervals so that the list of intervals no longer fulfills the IntervalList properties (i.e. a function with $\mathcal{D}^E \mapsto \mathcal{D}^{ER}$). Hence, we require a procedure (`normalize`) which can modify the list of intervals, such that it is normalized to a unique representation.

For abstract operations which can add intervals and thus increase the set length, we need to have a procedure that reduces the interval set length back to $N$. This is done by `enforceLength` (see below). Similarly, the order invariant and no adjacency invariant are guaranteed by the use of `merge` (defined below). Both procedures do not modify valid input fulfilling all invariants of $\mathbb{E}$, but only repair violations. As this is very useful, we combine both procedures to `normalize` : $\mathcal{D}^{ER} \mapsto \mathcal{D}^E$:

$$\texttt{normalize}(X) = \begin{cases} X & \text{if } X = \top \vee X = \bot \\ \texttt{enforceLength}(\texttt{merge(X)}) & \text{otherwise} \end{cases}$$

The interval merging operation `merge` ($\mathbb{E}_r \mapsto \mathbb{E}_r$) allows to combine overlapping and adjacent intervals. This procedure is done by first sorting the interval set and then checking every neighboring pair of intervals for direct adjacency or overlaps. Such pairs are then merged with the interval join $\sqcup$ (see Section 2.4).

$$\begin{aligned} \texttt{merge}(X) &= \texttt{merge}'(\texttt{sort}(X)) \\ \texttt{sort}(X) &= \{X_1, ..., X_n\} : \forall a, b \in X : a \leq' b \\ a \leq' b &= \underline{a} < \underline{b} \vee (\underline{a} = \underline{b} \wedge \overline{a} \leq \overline{b}) \\ \texttt{merge}'(X) &= \begin{cases} X & \text{if } |X| \leq 1 \\ (X_1 \sqcup X_2) \cup \texttt{merge}'(X \setminus \{X_1, X_2\}) & \text{if } \overline{X_1} + 1 \geq \underline{X_2} \\ \{X_1, X_2\} \cup \texttt{merge}'(X \setminus \{X_1, X_2\}) & \text{otherwise} \end{cases} \end{aligned}$$

`enforceLength` ($\mathbb{E}_r \mapsto \mathcal{D}^E$) recursively finds the closest pair of intervals and merges it until the the number of intervals in the set is lower or equal to $N$:

$$\texttt{enforceLength}(X) = \begin{cases} \bot & \text{if } |X| = 0 \\ X & \text{if } |X| \leq N \\ \texttt{enforceLength}(\texttt{mergePair}(X, & \\ \texttt{closestOverlap}(X))) & \texttt{otherwise} \end{cases}$$

$$\texttt{closestOverlap}(X) = a, b \in X : a \neq b \wedge \nexists c, d \in X : c \neq d \wedge |a - b| > |c - d|$$

$$\texttt{mergePair}(X, a, b) = X \setminus (a, b) \cup (a \sqcup b)$$

An important detail in the implementation of `closestOverlap` is which interval pair to pick when *more than one interval pair with the same, smallest distance occurs* in the list. Our choice was to pick the interval pair which is closer to list center and was motivated by the problem description of preserving extremal values.

One invariant of the abstract domain is to restrict the set cardinality within $[1, N]$. Instead of returning an empty set, operations on the IntervalList domain return $\bot$. Thus, all abstract operations can expect their input to conform to this invariant and in particular do not need to handle the edge case $E \in \mathbb{E} : |E| = 0$.

Note that `normalize` is *surjective* (there is only one unique possible normalization of $x$), but it is not *order-preserving* (i.e. monotonically increasing) which would mean $\forall x, y \in \mathbb{E}, x \subseteq y \Rightarrow \texttt{normalize}(x) \sqsubseteq \texttt{normalize}(y)$. This can shown by a negative example with $N = 3$: $\{3, 6, 8, 10\} \subseteq \{3, 6, 8, 10, 11\}$, but:

$$\texttt{normalize}(\{2, 4, 8, 10\}) = \{[2, 4], 8, 10\}$$
$$\not\subseteq \texttt{normalize}(\{2, 4, 8, 9, 10\}) = \texttt{normalize}(\{2, 4, [8, 9], 10\}) = \{2, 4, [8, 10]\}$$

However, as `normalize` never removes values and only might merges overlaps, it is *extensive* ($\forall x \in \mathcal{D}^E : x \sqsubseteq \texttt{normalize}(x)$). Furthermore, note that in practice an implementation does not need to recursively find the closest overlaps as the distance will not change by merging intervals. Thus, a more efficient implementation can either compute the interval distances once, sort, and store them for subsequent iterations or alternatively it could even store distance information as part of the interval set in a sorted data structure.

## 3.6 Lattice

In this section we attempt to create a lattice $L$ for this domain (recall Section 2.4 for the interval lattice).

We try to define a interval list lattice as $L = (\mathcal{D}^E, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. The lower bound $\bot$ and upper bound $\top$ were explained in Section 3.2. Hence, we attempt to define the least upper bound $\sqcup$ (1) and the great lower bound $\sqcap$ (2).

1) **Join semi-lattice** $(\mathcal{D}^E, \sqsubseteq, \sqcup)$, s.t. $\forall x, y \in \mathcal{D}^E : x \sqcup y$ (lub)

$$x \sqcup y = \begin{cases} \top & \text{if } x = \top \vee y = \top \\ y & \text{if } x = \bot \\ x & \text{if } y = \bot \\ \texttt{normalize}(\{x_1, ..., x_n, y_1, ..., y_m\}) & \text{otherwise} \end{cases}$$

The least upper bound of $\top$ can only be $\top$ (case I). Whereas the least upper bound for any element and $\bot$ is the respective element (case II, III). For an arbitrary pair (case IV) the least upper bound is exactly the unique representation of the join of both interval lists as no smaller representation is valid. A valid solution can be found via normalization (see Section 3.5). However, note that there might be multiple, incomparable solutions and thus that this *does not construct a join semi-lattice*. For example, consider $\{[10, 20], [30, 40]\} \sqcup \{[50, 60], [70, 80]\}$. As the location of the interval merge could occur anywhere there a three potential results for the lub: $\{[10, 40], [50, 60], [70, 80]\}$, $\{[10, 20], [30, 60], [70, 80]\}$, and $\{[10, 20], [30, 40], [50, 80]\}$. However, the lub, by its definition, must be unique.

2) **Meet semi-lattice** $(\mathcal{D}^E, \sqsubseteq, \sqcup)$, s.t. $\forall x, y \in \mathcal{D}^E : x \sqcap y$ (glb)

$$x \sqcap y = \begin{cases} \bot & \text{if } x = \bot \vee y = \bot \\ y & \text{if } x = \top \\ x & \text{if } y = \top \\ \texttt{enforceLength}(\texttt{sort}(\{x \sqcap^{\#} y \mid x \in X, y \in Y\})) & \text{otherwise} \end{cases}$$

The greatest lower bound of $\bot$ can only be $\bot$ (case I). Whereas the greatest lower bound for any element and $\top$ is the respective element (case II, III). For an arbitrary pair (case IV) the greatest lower bound is the unique representation of the intersection of both interval lists. The intersection of two interval lists cannot introduce new overlapping or directly adjacent intervals as neither of the interval lists contains overlapping or directly adjacent intervals. It is possible, however, to have up to $N - 1$ new intervals after the pairwise intersection of all intervals (see Figure 3.4 for an example) which is why normalization in respect to the length is required.

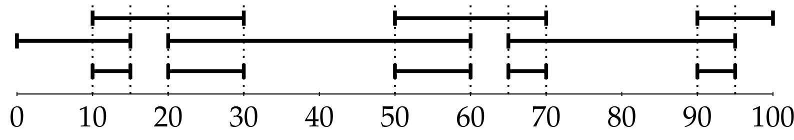



Figure 3.4: Pairwise intersection of $\{[10, 30], [50, 70], [90, 100]\} \cap \{[10, 15], [20, 60], [65, 95]\}$ resulting in $\{[10, 15], [20, 30], [50, 60], [65, 70], [90, 95]\}$.

However, the merge of intervals, in general, might result in different sets. For example, consider $\{[0, 30], [40, 70], [80, 90]\} \cap \{[0, 10], [20, 70], [80, 90]\}$ resulting in $\{[0, 10], [20, 30], [40, 70], [80, 90]\}$ which can be normalized as either $\{[0, 30], [40, 70], [80, 90]\}$, $\{[0, 10], [20, 70], [80, 90]\}$, or $\{[0, 10], [20, 30], [40, 90]\}$. Thus, this definition *does not construct a meet semi-lattice* and in general it is not possible when merging intervals semi-arbitrarily to a fixed, maximum number of intervals.

## 3.7 Abstract and concrete relationships

Similar to Section 2.5 we establish the correspondence between concrete and abstract values with an abstraction function $\alpha : \mathcal{D}^C \mapsto \mathcal{D}^E$:

$$\alpha(X) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } X = \emptyset \\ \top & \text{if } X = \mathbb{T} \\ \texttt{normalize}(\{[x, x] \mid x \in X\}) & \text{otherwise} \end{cases}$$

Example: $\alpha(\{2, 4, 6\}) = \{[2, 2], [4, 4], [6, 6]\}$, $\alpha(\{2, 4, 8, 20\} = \{[2, 4], [8, 8], [20, 20]\}$. Note that as `normalize` is not monotonic (see Section 3.5), $\alpha$ cannot be monotonic either. This will be relevant when looking at a potential Galois connection for this domain in the next subsection. Following, we define a concretization function $\gamma : \mathcal{D}^E \mapsto \mathcal{D}^C$:

$$\gamma(Y) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } y = \bot \\ \{x \in \mathbb{T}\} & \text{if } y = \top \\ \{x \in \mathbb{T} \mid \forall y \in Y : \underline{y_1} \leq x \leq \overline{y_1} \vee \cdots \vee \underline{y_N} \leq x \leq \overline{y_N}\} & \text{otherwise} \end{cases}$$

## 3.8 Galois connection

We try to define a Galois connection from the concrete domain $(\mathbb{T}, \subseteq)$ to the abstract domain $(\mathcal{D}^E, \sqsubseteq)$ with the function pair $\langle \alpha, \gamma \rangle$: $(\mathbb{T}, \subseteq) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} (\mathcal{D}^E, \sqsubseteq)$. Recall, that Galois

connection are defined as [CC92b]:

$$\forall x \in D : \forall y^{\#} \in D^{\#} : \alpha(x) \sqsubseteq^{\#} y^{\#} \iff x \sqsubseteq \gamma(y^{\#})$$

After [CC93] the following five properties are needed to establish a Galois connection between $\mathcal{D}$ and $\mathcal{D}^E$ (and equivalent to the concise definition):

1. Partial ordering in the concrete $\subseteq$ (given by $\mathbb{T}$)
2. Partial ordering in the abstract $\sqsubseteq$ (see Section 3.2)
3. Monotonic concretization $\gamma$: $a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a')$ (follows directly from $\gamma$ being a one-to-one isomorphic projection).
4. Sound approximation (see Section 3.9 below)
5. Most precise approximation $\forall c \in \mathcal{D} : \forall a \in \mathcal{D}^E : c \subseteq \gamma(a) \Rightarrow \alpha(c) \sqsubseteq a$

As the approximation is not the most precise approximation condition (5) is not fulfilled. This can be shown by a negative example:

$$\begin{aligned} c = \{1,3,7,11\} &\sqsubseteq \{1,3,[7,11]\} = \gamma(a) \\ \alpha(c) = \alpha(\{1,3,7,11\}) = \{[1,3],7,11\} &\not\sqsubseteq \{1,3,[7,11]\} = a \end{aligned}$$

Hence, with the defined abstract approximation and concretization function a Galois connection for our domain cannot be obtained.

## 3.9 Soundness correspondence

Our desired approximation is not a Galois connection and, as a result, we loose many interesting properties of Galois connections. Notwithstanding, the best abstract approximation into $D^{\#}$ can be created by ignoring the constraints defined in Section 3.1 as then we can use a one-to-one morphism $\alpha'(X) = \{[x,x] \mid x \in X\}$. However, as this would require vastly more space, the IntervalList domain normalizes this best abstract approximation and tries to creates a *"good-enough" approximation* by trading precision for reduced memory and increased speed without sacrificing soundness.

Recall that an abstraction is *sound* if no concrete values will be lost by neither the abstract approximation nor concretization ($\forall c \in \mathcal{D} : c \sqsubseteq \gamma(\alpha(c))$). This is true for the chosen approximation and concretization:

$$\begin{aligned}\{X_1, ..., X_k\} &\subseteq \gamma(\alpha(\{X_1, ..., X_k\})) \\ &= \gamma(\texttt{normalize}(\{X_1, ..., X_k\})) \quad \forall x \in \mathcal{D}^{ER} : x \sqsubseteq \texttt{normalize}(x) \\ &\subseteq \gamma(\{Y_1^\#, ..., Y_k^\#\}) \\ &= \{x \in \mathbb{T} \mid \underline{y_1^\#} \leq x_1 \leq \overline{y_1^\#} \vee \cdots \vee \underline{y_N^\#} \leq x_k \leq \overline{y_N^\#}\}\end{aligned}$$

By definition of the interval list normalization (see Section 3.5) is *extensive* (intervals can only be merged into bigger intervals, but not removed). Hence, the abstract representation will always contain all elements of $X$, but may add more elements. The chosen $\alpha$ is not a best approximation, but a *sound over-approximation*. Furthermore, it fulfills the *abstract minimality assumption* (as stated in [CC92b], 4.23): $\alpha = \{\langle c, a\rangle \mid \forall a' \in \mathcal{D}^E : (\langle c, a'\rangle \in \sigma : a' \sqsubseteq a) \Rightarrow (a \sqsubseteq a')\}$ as there cannot be a better approximation of a respective interval merge. An interval merge is optimal, but its location may not (especially when considering equi-distant interval pairs). Hence, for all possible variants $a, b$ resulting from different interval pairs both a and b contain new elements as part of their specific interval merge and thus : $a \not\sqsubseteq b \wedge b \not\sqsubseteq a$.

This correspondence is defined as a *weak abstraction/concretization connection* and any soundness correspondence which fullfils the abstract minimality assumption is a weak abstraction/concretization soundness correspondence (see [CC92b, Prop 8.1]). For reference, the four properties of a weak abstraction connection are:

1. $\forall c \in \mathcal{D} : c \subseteq \gamma(\alpha(c))$ (see above)
2. $\gamma{\sim}\gamma \circ \alpha \circ \gamma$ ($a, b \in \mathcal{D} : a{\sim}b \overset{\text{def}}{=} a \subseteq b \wedge b \subseteq a$)
3. $\forall c \in \mathcal{D} : \forall a \in \mathcal{D}^E : \alpha(c) \sqsubseteq a \implies c \subseteq \gamma(a) \not\Leftarrow \alpha(c) \sqsubseteq a$ (first assumption follows from extensiveness of $\alpha$ ($c \sqsubseteq \alpha(c)$) and $\gamma$: being a one-to-one mapping ($\gamma(\alpha(c)) \subseteq \gamma(a) \wedge \alpha(c) \subseteq \gamma(\alpha(c))$), for a negative example for the second part see Section 3.8 )
4. $\forall c, c' \in \mathcal{D} : c \subseteq c' \not\Leftarrow \alpha(c) \sqsubseteq \alpha(c')$

For the second property, in this case equality could be used as it is guaranteed that there is only one unique representation. Furthermore, the input of any concretization $\gamma$ must have been valid in the abstract domain and as $\gamma$ is a one-to-one isomorphic projection, it does not add or remove any values and neither can the subsequent abstract approximation, hence $\gamma = \gamma \circ \alpha \circ \gamma$.

An example for (4) is:

$$\begin{aligned} c = \{0,3,7,11\} &\subseteq \{0,3,7,9,11\} = c' \\ \alpha(c) = \alpha(\{0,3,7,11\}) = \{[0,3],7,11\} &\not\sqsubseteq \{0,3,[7,11]\} = \alpha(c') \end{aligned}$$

Finally, it should be noted that due to the *abstract soundness assumption* from [CC92b] (4.19) any approximation which contains a sound approximation is sound as well:

$$\forall c \in \mathcal{D} : \forall a, a' \in \mathcal{D}^\# : \big(\langle c, a\rangle \in \sigma \wedge a \sqsubseteq^\# a' \Rightarrow \langle c, a'\rangle \in \sigma\big)$$

Reconsider the best abstraction $\alpha' : \mathbb{T} \mapsto \mathcal{D}^{ER}$:

$$\alpha'(X) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } X = \emptyset \\ \top & \text{if } X = \mathbb{T} \\ \{x \in X \mid [x,x]\} & \text{otherwise} \end{cases}$$

This abstraction $\alpha'$ with the same concretization $\gamma$ is a Galois connection as both abstraction and concretization are one-to-one projections and hence $\forall x \in D : \forall y^\# \in D^\# : \alpha(x) =^\# y^\# \iff x = \gamma(y^\#)$.

For the equivalent special cases $\forall a \in \mathcal{D}^E : \alpha'(a) = \alpha(a)$ is trivially given. Consider the main case - $\alpha(a) = \texttt{normalize}(\{[x,x] \mid x \in X\}) = \texttt{normalize}(\alpha'(a))$. As `normalize` is extensive, it can be seen that $\forall a \in \mathcal{D}^E : \alpha'(a) \sqsubseteq \alpha(a)$. Thus, the Galois connection with abstract approximation $\alpha'$ equally implies soundness for the choosen abstract approximation of the IntervalList domain.

# 4 IntervalList operations

In this section the individual operations that are part of a non-relational domain in Astrée are described in detail. For all operations below: $X^\#, Y^\# \in \mathcal{D}^E$. The interval operator as defined in Section 2.6 often build the underlying building blocks. Furthermore, recall that for a monotone n-ary operator $f : \mathbb{T}^n \mapsto \mathbb{T}$ in the concrete domain a *sound abstract transformer* is any $f^\# : (\mathcal{D}^E)^n \mapsto \mathcal{D}^I$ for which $\forall (a_1, ..., a_n \in (\mathcal{D}^E)^n) :$ $\alpha(f(\gamma(a_1), ..., \gamma(a_n))) \sqsubseteq f^\#(a_1, ..., a_n)$ or $f(\gamma(a_1), ..., \gamma(a_n)) \subseteq \gamma(f^\#(a_1, ..., a_n))$. We list all non-trivial operations which are part of the implementation in Astrée.

## 4.1 Relations

Astrée uses subset or equality (`subseteq`) and equality (`equal`) during fixpoint iterations. These functions are only valid in the abstract domain, but cannot be transferred into the concrete. However, they can be used to determine if a fixpoint has been reached.

### 4.1.1 Subset or equality

Subset or equality uses the previously introduced $\sqsubseteq^\#$ operator from partial order (see Section 3.2).

$$[[X \subseteq Y]]^\# \stackrel{\text{def}}{=} \begin{cases} \texttt{true} & X^\# \sqsubseteq^\# Y^\# \\ \texttt{false} & \text{otherwise} \end{cases}$$

Consider the most common case $X^\# \neq \bot \wedge X^\# \neq \top$, $Y^\# \neq \bot \wedge Y^\# \neq \top$:

$$\begin{aligned} [[X \subseteq Y]]^\# &\iff \alpha(\gamma(X^\#) \subseteq \gamma(Y^\#)) \\ &= \alpha\left(\bigwedge_{i=1}^{|X|} X_i \subseteq \bigcup_{i=1}^{|Y|} Y_i\right) \\ &\not\subseteq (X^\# \sqsubseteq^\# Y^\#) \end{aligned}$$

This operation is not a transfer function. For example, this is possible:

$$X = \{1, 5, 9\} \Rightarrow X^{\#} = \{1, 5, 9\}$$
$$Y = \{0, 11, 30, 40\} \Rightarrow Y^{\#} = \{[0, 11], 30, 40\}$$

Here, $X^{\#} \sqsubseteq^{\#} Y^{\#}$, but $X \not\subseteq Y$.

### 4.1.2 Equality

Equality can be defined as $[[X =^{\#} Y]] = X \subseteq Y \wedge Y \subseteq X$, but for the implementation it is more efficiently defined as:

$$[[X =^{\#} Y]] \triangleq (|X^{\#}| = |Y^{\#}|) \wedge \bigwedge_{i=1}^{|X^{\#}|} X_i^{\#} = Y_i^{\#}$$

This operation is not transfer function. For example, this is possible:

$$X = \{0, 5, 10, 30, 40\} \Rightarrow X^{\#} = \{[0, 10], 30, 40\}$$
$$Y = \{0, 6, 10, 30, 40\} \Rightarrow Y^{\#} = \{[0, 10], 30, 40\}$$

Here, $X^{\#} =^{\#} Y^{\#}$, but $X \neq Y$.

## 4.2 Transfer functions

We provide abstractions for the union (subsection 4.2.1), intersection (subsection 4.2.2), all arithmetic forward operators (subsection 4.2.3, subsection 4.2.4, subsection 4.2.5), comparison backwards tests (subsection 4.2.6) and widening (subsection 4.2.7).

### 4.2.1 Union

The union of two interval lists concats all intervals and finds a valid abstraction (similar to Section 3.6).

$$[[X \cup Y]]^{\#} \triangleq \begin{cases} \top & \text{if } X^{\#} = \top \vee Y^{\#} = \top \\ X^{\#} & \text{if } X^{\#} \neq \bot \wedge Y^{\#} = \bot \\ Y^{\#} & \text{if } X^{\#} = \bot \wedge Y^{\#} \neq \bot \\ \bot & \text{if } X^{\#} = \bot \wedge Y^{\#} = \bot \\ \text{normalize}(X^{\#} \cup Y^{\#}) & \text{if } X^{\#} \neq \bot \wedge Y^{\#} \neq \bot \end{cases}$$

**Proof:**
I) if $X^\# = \top$ (and similar for $Y^\# = \top$):

$$
\begin{aligned}
[[X \cup Y]]^\# \iff & \alpha(\gamma(X^\#) \cup \gamma(Y^\#)) \\
&= \alpha\left(\{x \mid x \in \mathbb{T}\} \cup \bigcup_{i=1}^{|Y|} Y_i\right) \\
&= \alpha\left(\{x \mid x \in \mathbb{T}\}\right) = \top \\
&\sqsubseteq (X^\# \cup^\# Y^\#) = \top
\end{aligned}
$$

II, III) if $X^\# = \bot$ (and similar for $Y^\# = \bot$):

$$
\begin{aligned}
[[X \cup Y]]^\# \iff & \alpha(\gamma(X^\#) \cup \gamma(Y^\#)) \\
&= \alpha\left(\emptyset \cup \bigcup_{i=1}^{|Y|} Y_i\right) \\
&= \text{normalize}\left(\{Y_1^\#, ..., Y_N^\#\}\right) = Y^\# \\
&\sqsubseteq (X^\# \cup^\# Y^\#) = (\emptyset \cup^\# Y^\#) = Y^\#
\end{aligned}
$$

IV) if $X^\# = \bot \wedge Y^\# = \bot$:

$$
\begin{aligned}
[[X \cup Y]]^\# \iff & \alpha(\gamma(X^\#) \cup \gamma(Y^\#)) \\
&= \alpha(\emptyset \cup \emptyset) = \alpha(\emptyset) = \bot \\
&\sqsubseteq (X^\# \cup^\# Y^\#) = \bot
\end{aligned}
$$

V) if $X^\# \neq \bot \wedge Y^\# \neq \bot$:

$$
\begin{aligned}
[[X \cup Y]]^\# \iff & \alpha(\gamma(X^\#) \cup \gamma(Y^\#)) \\
&= \alpha\left(\bigcup_{i=1}^{|X|} X_i \cup \bigcup_{i=1}^{|Y|} Y_i\right) \\
&= \text{normalize}\left(\{X_1^\#, ..., X_N^\#, ..., Y_1^\#, ..., Y_N^\#\}\right) \\
&\sqsubseteq (X^\# \cup^\# Y^\#) \\
&= \text{normalize}\left(\{X_1^\#, ..., X_N^\#, ..., Y_1^\#, ..., Y_N^\#\}\right)
\end{aligned}
$$

Note that due to the sortedness of both lists, merge-sorting two interval lists can be done in $\mathcal{O}(N)$. Merging the closest interval pairs for `enforceLength` during normalization,

however, is more expensive and might take up to $\mathcal{O}(N^2)$ (with a recursive find) or $\mathcal{O}(N * log(N))$ with a sorted tree or heap of closest interval pairs. For small $N$ this is not problematic, but if a higher $N$ is used, it is worthwhile to consider a trade-off with a less optimal heuristic to trade performance for a slight loss in precision. Such a heuristic can be:

I apply `merge` with a lower $N$ before the union and normalization. For example, for a $N = 5$ interval list, first merge the closest overlapping intervals with $N = 3$ and then join both three-element interval lists.

II pick the lowest and highest interval (due to the sortedness of both lists the lowest and highest interval can be selected in $\mathcal{O}(1)$) and merge all other intervals to the center interval.

III randomly select $N$ intervals from the union of both lists and grow these randomly selected intervals by the process of merging the non-selected intervals with its closest interval.

IV select $N$ partitioning intervals from the union and merge non-selected intervals into the closest partitioning. For example, such partition could retain the extremal values ($I_1$, $N/2$, $I_N$) or ($I_1$, $closest(I_1 + d/2)$, $I_N$), but also be a general partitioning of the list ($closest(I_1 + d/4)$, $closest(I_1 + 2d/4)$, $closest(I_1 + 3d/4)$) ($d = I_N - I_1$).

### 4.2.2 Intersection

For the intersection of two interval lists we intersect every interval of $X^\#$ with every interval of $Y^\#$ (similar to Section 3.6).

$$[[X \cap^\# Y]] \triangleq \begin{cases} X^\# & \text{if } Y^\# = \top \\ Y^\# & \text{if } X^\# = \top \\ \texttt{enforceLength}(\texttt{sort}(C^\#)) & \text{if } X^\# \neq \bot \wedge Y^\# \neq \bot \wedge |C^\#| \geq 1 \\ & \quad C^\# = \{x^\# \sqcap^\# y^\# \mid x^\# \in X^\#,\ y^\# \in Y^\#\} \\ \bot & \text{otherwise} \end{cases}$$

I, II) $X^\# = \top$ (or $Y^\# = \top$)

$$\begin{aligned} [[X \cap Y]]^\# &\iff \alpha(\gamma(X^\#) \cap \gamma(Y^\#)) \\ &= \alpha\left(\{x \in \mathbb{T}\} \cap \gamma(Y^\#)\right) \\ &= \alpha(\gamma(Y^\#)) \\ &\sqsubseteq (X^\# \cap^\# Y^\#) = Y^\# \end{aligned}$$

IV) $X^\# = \bot$ (or $Y^\# = \bot$, or $X \cap Y = \emptyset$)

$$\begin{aligned}
[[X \cap Y]]^\# \iff & \alpha(\gamma(X^\#) \cap \gamma(Y^\#)) \\
&= \alpha\left(\emptyset \cap \bigcup_{i=1}^{|Y|} Y_i\right) \\
&= \alpha(\emptyset) = \bot \\
&\sqsubseteq (X^\# \cap^\# Y^\#) = \bot
\end{aligned}$$

III) $X^\# \neq \bot \wedge Y^\# \neq \bot$

$$\begin{aligned}
[[X \cap Y]]^\# \iff & \alpha(\gamma(X^\#) \cap \gamma(Y^\#)) \\
&= \alpha\left(\bigcup_{i=1}^{|X|} X_i \cap \bigcup_{i=1}^{|Y|} Y_i\right) \\
&= \alpha\left(\bigcup_{i=1}^{|X|}\bigcup_{i=1}^{|Y|} X_i \cap Y_i\right) \\
&= \texttt{normalize}\left(\{x^\# \cap y^\# \mid x^\# \in X^\#, y^\# \in Y^\#\}\right) \\
&\sqsubseteq (X^\# \cap^\# Y^\#) \\
&= \texttt{enforceLength}(\texttt{sort}(\{x^\# \cap^\# y^\# \mid x^\# \in X^\#, y^\# \in Y^\#\}))
\end{aligned}$$

The intersection of two interval lists cannot introduce new overlapping or directly adjacent intervals as neither of the interval lists contains overlapping or directly adjacent intervals. Hence, merge cannot modify the list and (1) sorting, and (2) length enforcement is equivalent to normalization. Furthermore, it is possible to avoid the sorting as both interval lists are sorted and by intersecting every element of list $X^\#$ with all elements of $Y^\#$, the respective elements of list $X^\#$ cannot increase and thus the original list order is preserved. The complexity of the intersection as implemented is, however, $\mathcal{O}(N^2 + N^2) = \mathcal{O}(N^2)$ as every element of $X^\#$ might require a full list traversal and the resulting list might have up to $N - 1$ new intervals which need to be merged via `enforceLength`.

### 4.2.3 Forward arithmetic initialization

Astrée has three types of initialization operations: `TOP` (the full domain), `BOT` (no value of the domain) and `RANGE` for an interval or point initialization. They are stated here for completeness.

$$[[op^\#]] = \begin{cases} \top & \text{if } op^\# = \texttt{TOP} \\ \bot & \text{if } op^\# = \texttt{BOT} \\ \{[l,h]\} & \text{if } op^\# = \texttt{RANGE}(l,h) \end{cases}$$

$$\begin{aligned} [[\texttt{TOP}]]^\# \iff & \alpha(\gamma(\top)) \\ & \alpha(\{x \in \mathbb{T}\}) = \top \\ & \sqsubseteq (\texttt{TOP}^\#) = \top \end{aligned}$$

$$\begin{aligned} [[\texttt{BOT}]]^\# \iff & \alpha(\gamma(\bot)) \\ & \alpha(\emptyset) = \bot \\ & \sqsubseteq (\texttt{BOT}^\#) = \bot \end{aligned}$$

$$\begin{aligned} [[\texttt{RANGE}(l,h)]]^\# \iff & \alpha(\gamma(\{[l,h]\})) \\ & \alpha(\{x \in \mathbb{T} \mid l \leq x \leq h\}) = \{[l,h]\} \\ & \sqsubseteq \texttt{RANGE}^\#(l,h) = \{[l,h]\} \end{aligned}$$

Normalization is not necessary for RANGE as the set contains only a single interval.

### 4.2.4 Forward unary arithmetic operations

Astrée implements a few unary operations. Most notably, negation (4.2.4) and bound-checking (4.2.4).

$$[[X \; \square^\#]] = \begin{cases} \top & \text{if } X = \top \\ \bot & \text{if } X = \bot \\ \texttt{neg}^\# \; X^\# & \text{if } \square^\# = \texttt{NEG} \\ \texttt{BOUND_CHECK}^\# \; X^\# & \text{if } \square^\# = \texttt{BOUND_CHECK op}^\# \end{cases}$$

#### Negation

As the interval list is guaranteed to be sorted, the negation of it only requires to negate each interval individually and reverse the interval list:

$$\texttt{neg}^\# \; X^\# = \texttt{reverse}([-y^\#, -x^\#] \mid [x^\#, y^\#] \in X^\#)$$

Hence, for negation `normalize` is equivalent to `reverse`:

$$
\begin{aligned}
[[\texttt{NEG}\ X]]^{\#} \iff\ & \alpha(-\gamma(X^{\#})) \\
&= \alpha\left(-\bigcup_{i=1}^{|X|} X_i\right) \\
&= \alpha\left(\bigcup_{i=1}^{|X|} -X_i\right) \\
&= \text{normalize}\left(\{[-\overline{x}^{\#}, -\underline{x}^{\#}] \mid x^{\#} \in X^{\#}\}\right) \\
&\sqsubseteq (\texttt{NEG}\ X^{\#}) = \texttt{reverse}(\{[-b^{\#}, -a^{\#}] \mid [a^{\#}, b^{\#}] \in X^{\#}\})
\end{aligned}
$$

### Bound-checking

Bound-checking can result from a boolean coercion from the program (bool), an up- or down-cast into a different data type (mod) or partitioning (intersect). The range and keep bound-checking operations are only mentioned for completeness.

$$
\begin{aligned}
\texttt{BOUND_CHECK}^{\#}\ X^{\#}\ \texttt{op}^{\#} &= \begin{cases} X^{\#} & \text{if op}^{\#} = \texttt{Keep} \\ \texttt{bool}\ X^{\#} & \text{if op}^{\#} = \texttt{Bool} \\ X^{\#}\ \text{mod}^{\#}\ m & \text{if op}^{\#} = \texttt{Mod}\ m \\ \{[l,h]\} & \text{if op}^{\#} = \texttt{Range}(l,h) \\ X^{\#}\ \cap^{\#}\ \{[l,h]\} & \text{if op}^{\#} = \texttt{Intersect}(l,h) \end{cases} \\
\texttt{bool}\ X^{\#} &= \begin{cases} \{[0,1]\} & \text{if } X^{\#} = \top \\ \{[0,0]\} & \text{if } X^{\#} = \{[0,0]\} \\ \{[0,1]\} & \text{if } 0 \in X^{\#} \\ \{[1,1]\} & \text{otherwise} \end{cases} \\
X^{\#}\ \texttt{mod}^{\#}\ m &= \text{normalize}\ (x^{\#}\ \texttt{mod}\ m \mid x \in X^{\#}) \\
x^{\#}\ \texttt{mod}\ m &= \begin{cases} [\underline{m}, \overline{m}] & \text{if } \overline{x}^{\#} - \underline{x}^{\#} > \overline{m} - \underline{m} \vee \\ & (\underline{x}^{\#} - \underline{m}) \text{ mod } z > (\overline{x}^{\#} - \underline{m}) \text{ mod } z \\ [\underline{m} + (\underline{x}^{\#} - \underline{m}) \text{ mod } z, & \\ \underline{m} + (\overline{x}^{\#} - \underline{m}) \text{ mod } z] & \text{otherwise} \\ & \textit{with } z = \overline{m} - \underline{m} + 1 \end{cases}
\end{aligned}
$$

In particular for `mod`:

$$
\begin{aligned}
[[X \texttt{ mod } m]]^{\#} \iff & \alpha(\gamma(X^{\#}) \texttt{ mod } m) \\
& = \alpha\left(\left(\bigcup_{i=1}^{|X|} X_i\right) \texttt{ mod } m\right) \\
& = \alpha\left(\bigcup_{i=1}^{|X|} X_i \texttt{ mod } m\right) \\
& = \text{normalize}\left(\{(x^{\#} \texttt{ mod } \overline{m}) + \underline{m} \mid x \in X^{\#}\}\right) \\
& \sqsubseteq (X^{\#} \texttt{ mod}^{\#} m) = \text{normalize}(\{x^{\#} \texttt{ mod}^{\#} m \mid x^{\#} \in X^{\#}\})
\end{aligned}
$$

Consequently, we need to ensure that $\texttt{mod}^{\#}$ is sound (i.e. it does not loose any values created by `mod` in the concrete). For this, we consider its cases individually:

**Case Ia)**: $\overline{x}^{\#} - \underline{x}^{\#} > \overline{m} - \underline{m}$ ($x^{\#}$ contains more values than the new bound)

$$
\begin{aligned}
& = \text{normalize}\left(\{(x^{\#} \texttt{ mod } \overline{m}) + \underline{m} \mid x^{\#} \in X^{\#}\}\right) \\
& \sqsubseteq (X^{\#} \texttt{ mod}^{\#} m) = [\underline{m}, \overline{m}]
\end{aligned}
$$

When $X^{\#}$ might overflow, returning all possible concrete values $[\underline{m}, \overline{m}]$ in abstract domain is a sound approximation.

**Case Ib)**: $(\underline{x}^{\#} - \underline{m}) \textit{ mod } z > (\overline{x}^{\#} - \underline{m}) \textit{ mod } z$

Again, $x^{\#}$ might overflow partially and hence similar to (Ia) the approximation returns the full potential range.

The precision loss is only smaller for case (II). This means that for each interval the modulo operation can be done individually:

$$
\begin{aligned}
& = \text{normalize}\left(\{(x^{\#} \texttt{ mod } \overline{m}) + \underline{m} \mid x^{\#} \in X^{\#}\}\right) \\
& \sqsubseteq (X^{\#} \texttt{ mod}^{\#} m) \\
& = \text{normalize}(\{[\underline{m} + (\underline{x}^{\#} - \underline{m}) \text{ mod } z, \underline{m} + (\overline{x}^{\#} - \underline{m}) \text{ mod } z] \mid x^{\#} \in X^{\#}\})
\end{aligned}
$$

The resulting interval is at least as big as the concrete $x^{\#}$ as the size of $x^{\#}$ is guaranteed to be lower than of $m$ (Ia) and is monotonic (Ib). Hence, modulo on the lowest and highest value of the interval will enclose all points in-between.

### 4.2.5 Forward binary arithmetic operations

All common arithmetic operations (addition, subtraction, multiplication, division, modulo, bitwise and, bitwise or, bitwise xor, left shift, and right shift) can be abstracted for the IntervalList domain. For the IntervalList domain the basic operation strategy is for each interval of the first operand to apply the operation against all intervals of the second operand using the interval operations defined in Section 2.6. Furthermore, it needs to be ensured that there can be no underflows or overflows during any operation and the result needs to be normalized afterwards. For some operations extra care must be taken to avoid undefined operations like a division by zero.

$$
[[A \; \Box \; B]]^\# = \begin{cases} \top & \text{if } A^\# = \top \lor B^\# = \top \\ \bot & \text{if } A^\# = \bot \lor B^\# = \bot \\ [T_{min}, T_{max}] & \text{if } \Box = \{/, \%\} \land 0 \in B^\# \\ (\text{map } /^\# \; A^\# \; (B^\# \cap^\# [\mathbb{T}_{min}, -1])) \cup^\# & \text{if } \Box = / \land min(B^\#) < 0 \land \\ (\text{map } /^\# \; A^\# \; (B^\# \cap^\# [1, \mathbb{T}_{max}])) & \quad max(B^\#) > 0 \\ \text{map } \Box^\# \; A^\# \; B^\# & \text{otherwise} \end{cases} \quad \text{s. t.}
$$

$$
\text{map } \Box^\# \; A^\# \; B^\# = \begin{cases} \text{normalize}(C^\#) & \text{if } |C^\#| > 0 \\ \top & \text{if } |C^\#| = 0 \lor \top \in C^\# \end{cases}
$$

$$
C^\# = \left\{ \begin{cases} \top & \text{if } \underline{c}^\# < \mathbb{T}_{min} \lor \overline{c}^\# > \mathbb{T}_{max} \\ c^\# & \text{otherwise} \end{cases} \middle| \; a^\# \in A^\#, b^\# \in B^\# \right\}
$$

$$
\text{with } c^\# = a^\# \; \Box^\# \; b^\#
$$

For most operations $\Box^\#$ can be defined as the minimum and maximum of all possible values of the interval endpoints:

$$
\begin{aligned} a^\# \; \Box^\# \; b^\# &= [min(C^\#), max(C^\#)] \\ C^\# &= \{\underline{a}^\# \; \Box^\# \; \underline{b}^\#, \overline{a}^\# \; \Box^\# \; \underline{b}^\#, \underline{a}^\# \; \Box^\# \; \overline{b}^\#, \overline{a}^\# \; \Box^\# \; \overline{b}^\#\} \end{aligned}
$$

As we have seen in Section 2.6 for certain operations like addition or subtraction this can be simplified to two operations:

$$
\begin{aligned} a^\# \; +^\# \; b^\# &= [\underline{a}^\# + \underline{b}^\#, \overline{a}^\# + \overline{b}^\#] \\ a^\# \; -^\# \; b^\# &= [\underline{a}^\# - \overline{b}^\#, \overline{a}^\# - \underline{b}^\#] \end{aligned}
$$

The soundness proof is similar to the unary operations. For the general case with $X^\# \neq \top \wedge X^\# \neq \bot, Y^\# \neq \top \wedge Y^\# \neq \bot$:

$$
\begin{aligned}
[[X \,\square\, Y]]^\# &\iff \alpha(\gamma(X^\#) \,\square\, \gamma(Y^\#)) \\
&= \alpha\left(\bigcup_{i=1}^{|X|} X_i \,\square\, \bigcup_{i=1}^{|Y|} Y_i\right) \\
&= \alpha\left(\bigcup_{i=1}^{|X|}\bigcup_{i=1}^{|Y|} X_i \,\square\, Y_i\right) \\
&= \text{normalize}\left(\{x^\# \,\square\, y^\# \mid x^\# \in X^\#, y^\# \in Y^\#\}\right) \\
&\sqsubseteq (X^\# \,\square^\#\, Y^\#) = \text{normalize}(\{x^\# \,\square\, y^\# \mid x^\# \in X^\#, y^\# \in Y^\#\})
\end{aligned}
$$

Hence, arithmetic operations are sound as long as $x \,\square\, y \subseteq x^\# \,\square^\#\, y^\#$. With the definition of $\square^\#$ above this is the case if the arithmetic operator is monotonic as the bounds are kept $(a \in S : \forall b \in S : a \leq b) \Rightarrow (\forall b \in S : f(a) \leq (b))$.

The main difference of division is that it needs to partition the divisor into a positive part and negative part if both can occur and join both results afterwards. However, if it is possible for zero to be part of the divisor the operation is not defined and Astrée triggers a division by zero alarms. Subsequently, to allow for a partial recovery of the static analyzer and discovery of more alarms in the same run, the analysis is set to $\top$ which is done in case (III) by the full range of the data type.

#### $a^\# \,\square^\#\, b^\#$ special case for $|a| = 2 \vee |b| = 2$

The IntervalList domain can introduce one important special case: if either of the operands is an interval of length 2, the resulting list will contain two intervals as both values of the intervals are used to build a new interval, i.e. if $|a| = 2$, then $a^\# \,\square^\#\, b^\# = \{[\underline{a}^\#, \underline{a}^\#] \,\square^\#\, b^\#, [\overline{a}^\#, \overline{a}^\#] \,\square^\#\, b^\#\}$. This can reduce the loss of precision for a few cases, e.g. `x<<y` with $x$ in $[5,8]$ and $y$ in $[2,3]$. Here with $minmax(X) = [min(X), max(X)]$ we get $\{[5 << 2, 5 << 3], [8 << 2, 8 << 3]\} = \{[20,32],[40,64]\}$ instead of $\{minmax(\{5 << 2, 5 << 3, 8 << 2, 8 << 3\})\} = \{minmax(\{20,40,32,64\})\} = \{[20,64]\}$. However, during preliminary evaluations we found almost no improvements for bigger code whereas it did result in a significant increase in memory usage as for the most common arithmetic operations it does not improve the result, but increases list size and allocations.

Similar to the intersection, every element of the interval list of the first operand requires a traversal of every element in the interval list of the second operand ($\mathcal{O}(N^2)$).

However, even without the above mentioned modification for interval lists of length 2, the resulting list size can be up to $N^2$ and thus requires $N^2 - N$ merge operations which with the linear recursive merge in $\mathcal{O}(N)$ leads to an overall $\mathcal{O}(N^3)$. In practice, the used $N$ of this domain were sufficiently small ($\leq 7$). However, for bigger $N$ it might be more feasible to merge the interval list of both operands to a smaller $N$ before computing the arithmetic operations (see the performance consideration of the IntervalList union in subsection 4.2.5 for details).

### 4.2.6 Backward comparison tests

Unary and binary backward arithmetic operators were derived mechanically after Equation 2.10. Hence, we only provide backward comparison tests for equivalence, in-equivalence and the standard ordering operations ($<, \leq, >, \geq$). Analogous to subsection 2.6.2 we define the backward comparison test $f^\# : \mathcal{D}^E \times \mathcal{D}^E \mapsto \mathcal{D}^E \times \mathcal{D}^E$ where the resulting domain duplet represent the respective refined variables for a truthy comparison.

#### Equality

Equality can be defined by re-using the previously defined intersection $\cap^\#$ (see subsection 4.2.2). Furthermore, intersection is commutative ($Y^\# \cap^\# X^\# = X^\# \cap^\# Y^\#$):

$$[[X \overset{\leftarrow}{=} Y]]^\# \triangleq \begin{cases} \top & \text{if } X^\# = \top \vee Y^\# = \top \\ (Z^\#, Z^\#) & \text{if } Z^\# \neq \bot \ \textit{with}\ Z^\# = X^\# \cap^\# Y^\# \\ \bot & \text{otherwise} \end{cases}$$

$\top$ must be lifted (I), $\bot$ is lifted by (II). In addition, if $X^\#$ and $Y^\#$ do not overlap in the abstract domain they can never be equal in the concrete as by the soundness property $x \sqsubseteq \gamma(\alpha(x))$ with a one-to-one concretization mapping the abstract space must always be a superset (III). Finally, we look closer at (II) for a single $Z^\#$:

$$\begin{aligned}
[[X \overset{\leftarrow}{=} Y]]^\# \iff & \alpha(\gamma(X^\#) \ = \ \gamma(Y^\#)) \\
&= \alpha\left(\bigcup_{i=1}^{|X|} X_i \ = \ \bigcup_{i=1}^{|Y|} Y_i\right) \\
&= \texttt{normalize}\left(\bigcup_{i=1}^{|X^\#|}\bigcup_{i=1}^{|Y^\#|} X_i^\# \cap Y_i^\#\right) \\
&\sqsubseteq (X^\# =^\# Y^\#) = X^\# \cap^\# Y^\# = \{x^\# \cap^\# y^\# \mid x^\# \in X^\#, y^\# \in Y^\#\}
\end{aligned}$$

In subsection 4.2.2 we have shown that $X \cap Y \sqsubseteq X^\# \cap^\# Y^\#$.

**Inequality**

For inequality we can only refine variables further if either of intervals is a single prime interval (point):

$$[[X \overleftarrow{\neq} Y]]^\# \triangleq \begin{cases} \top & \text{if } X^\# = \top \vee Y^\# = \top \\ \bot & \text{if } X^\# = \bot \vee Y^\# = \bot \\ \bot & \text{if } \texttt{equal}(X^\#, Y^\#) \wedge \\ & \quad \forall x^\# \in X^\# : |x^\#| = 1 \wedge \\ & \quad \forall y^\# \in X^\# : |y^\#| = 1 \\ (B^\#, A^\#) \textit{ with} & \\ \quad (A^\#, B^\#_{1_1}) = \text{ineq}(Y^\#, X^\#) & \text{if } \texttt{isPoint}(X^\#) \\ \text{ineq}(X^\#, Y^\#_{1_1}) & \text{if } \texttt{isPoint}(Y^\#) \\ (X^\#, Y^\#) & \text{otherwise} \end{cases}$$

$$\texttt{isPoint}(A^\#) = \begin{cases} \underline{A_1}^\# = \overline{A_1}^\# & \text{if } |A^\#| = 1 \\ \texttt{false} & \text{otherwise} \end{cases}$$

(I) and (II) are the usual lifting of unknown or impossible states. Note for (III) that we cannot infer from equality in the abstract domain (see `equal` from subsection 4.1.2) that they are equal in the concrete space in the general case. However, if all intervals are points we have a special case as no normalization nor precision loss can have happened yet.

$$\begin{aligned} [[X \overleftarrow{\neq} Y]]^\# &\iff \alpha(\gamma(X^\#) \;\neq\; \gamma(Y^\#)) \\ &= \alpha\left(\bigcup_{i=1}^{|X|} X_i \;\neq \bigcup_{i=1}^{|Y|} Y_i\right) \\ &= \alpha(\emptyset) = \bot \\ &\sqsubseteq (X^\# \neq^\# Y^\#) = \bot \end{aligned}$$

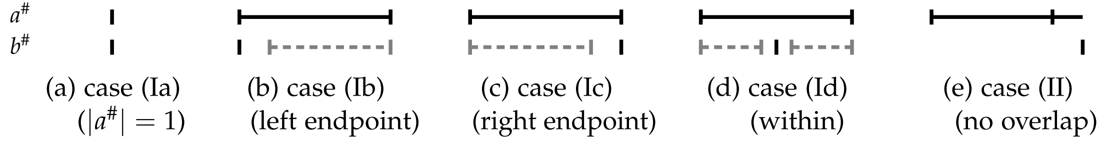
Lastly, for (IV) if either of the interval list is a single value, further refinement can be done with `ineq` ($\mathbb{E} \times \mathbb{T} \mapsto \mathbb{E}$). `ineq` removes the interval list consisting of single value $b^\#$ whenever possible from the intervals of $A^\#$. Note that `enforceLength` is only required due to the interval splitting in case IV of `ineq`".

$$\text{ineq}(A^\#, b^\#) = \texttt{enforceLength}(\text{ineq}'(A^\#, b^\#))$$

$$\text{ineq}'(A^\#, b^\#) = \begin{cases} \emptyset & \text{if } |A^\#| = 0 \\ (\text{ineq}''(A_1^\#, b^\#), \text{ineq}'(A^\# \setminus A_1^\#, b^\#)) & \text{if } |A^\#| \geq 1 \end{cases}$$

$$\text{ineq}''(a^\#, b^\#) = \begin{cases} \begin{cases} \emptyset & \text{if } \underline{a}^\# = \overline{a}^\# \\ [\underline{a}^\# + 1, \overline{a}^\#] & \text{if } \underline{a}^\# = b^\# \\ [\underline{a}^\#, \overline{a}^\# - 1] & \text{if } \overline{a}^\# = b^\# \\ ([\underline{a}^\#, b^\# - 1], [b^\# + 1, \overline{a}^\#]) & \text{otherwise} \end{cases} & \text{if } \underline{a}^\# < b^\# \wedge b^\# < \overline{a}^\# \\ a^\# & \text{otherwise} \end{cases}$$

In general, there are two different scenarios for inequality of an interval $a^\#$ with a point $b^\#$ to consider: $b^\#$ overlaps with $a^\#$ (case I) or it does not (case II). In latter case (II) we cannot perform refinement. Otherwise, we can consider the type of the overlap and remove $b^\#$ from the interval. Case (II) and all four potential sub cases of (I) are illustrated in Figure 4.1.




(a) case (Ia) ($|a^\#| = 1$) (b) case (Ib) (left endpoint) (c) case (Ic) (right endpoint) (d) case (Id) (within) (e) case (II) (no overlap)

Figure 4.1: Backward inequality test examples for the IntervalList. All five potential cases are listed. $a^\#$ and $b^\#$ are represented by black, solid intervals. Grey, dotted intervals represent the value of $a^\#$ after refinement.

If an overlapping interval of $A^\#$ consists of a single value as well (i.e. this value is the same as $b^\#$) (Ia), we know that this interval cannot be equal to $B^\#$ and can remove it from the list. If the overlap happens at the lower (Ib) or upper (Ic) border of $a^\#$, we can remove the respective endpoint from the interval $a^\#$. Finally, if the overlap is within $a^\#$ (Id), $a^\#$ can be partitioned into two new intervals. Hence, overall case (III) and (IV) of inequality with ($|X^\#| = 1$, similar for $|Y^\#| = 1$) are:

$$\begin{aligned} [[X \overleftarrow{\neq} Y]]^\# &\iff \alpha(\gamma(X^\#) \;\neq\; \gamma(Y^\#)) \\ &= \alpha\left(\underline{x} \neq \bigcup_{i=1}^{|Y|} Y_i\right) \\ &= \text{normalize}\left(\{y^\# \setminus \underline{x}^\# \mid y^\# \in Y^\#\}\right) \\ &\sqsubseteq (X^\# \neq^\# Y^\#) = \text{normalize}(\{ineq'(Y_1^\#, \underline{X_1}^\#), ..., ineq'(Y_N^\#, \underline{X_1}^\#)\}) \end{aligned}$$

ineq′ guarantees to only remove at most the single value interval list $X$ and thus $Y_k \setminus \underline{X_1} \sqsubseteq ineq'(Y_k^\#, \underline{X_1}^\#)$. It is worth mentioning that case (IV) of ineq″ is the major difference over the normal interval domain and allows to create holes in existing intervals.

**Comparison**

For the remaining operators $\Box_{cmp} \in \{<, \leq, >, \geq\}$ each variable needs to be compared against each other as cmp is not symmetric. The duality function revComp: $\{<, \leq, >, \geq\} \mapsto \{<, \leq, >, \geq\}$ maps each comparison operator to its matching opposite and allows the comparison refinement definition to be re-used for the variable $Y$. The refinement for $\Box_{cmp}$ is based on cutting off the lower (or upper) part of a variable $X^\#$ by the upper (or lower) bound of $Y$ in cmp.

$$[[X \overleftarrow{\Box_{cmp}} Y]]^\# = \begin{cases} \top & \text{if } X^\# = \top \vee Y^\# = \top \\ (A^\#, B^\#) & \text{if } A^\# \neq \bot \wedge B^\# \neq \bot \text{ with} \\ & \quad A^\# = \texttt{cmp}(X^\#, Y^\#, \Box^\#_{cmp}) \\ & \quad B^\# = \texttt{cmp}(Y^\#, X^\#, \texttt{revCmp}(\Box^\#_{cmp})) \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{revCmp}(\Box^\#_{cmp}) = \begin{cases} < & \text{if } \Box^\#_{cmp} = > \\ \leq & \text{if } \Box^\#_{cmp} = \geq \\ > & \text{if } \Box^\#_{cmp} = < \\ \geq & \text{if } \Box^\#_{cmp} = \leq \end{cases}$$

$$\texttt{cmp}(X^\#, Y^\#, \Box^\#_{cmp}) = \begin{cases} A^\# & \text{if } |A^\#| > 0 \text{ with } A^\# = \{z^\# \mid e^\# \in X^\# \wedge z^\# \neq \bot\} \\ & \qquad \text{with } z^\# = \texttt{cut}(e^\#, Y^\#, \Box^\#_{cmp}) \\ \bot & \text{otherwise} \end{cases}$$

The cut operation $\mathbb{I} \times \mathbb{E} \mapsto \mathbb{I}$ will cut the interval $x^\#$ against the hull of $Y^\#$ or remove the interval $x^\#$ entirely:

$$\texttt{cut}(x^\#, Y^\#, \Box^\#_{cmp}) = \begin{cases} [\underline{x}^\#, min(\overline{x}^\#, \overline{Y^\#} - 1)] & \text{if } \Box^\#_{cmp} = < \wedge\ \underline{x}^\# < \overline{Y^\#} \\ [\underline{x}^\#, min(\overline{x}^\#, \overline{Y^\#})] & \text{if } \Box^\#_{cmp} = \leq \wedge\ \underline{x}^\# \leq \overline{Y^\#} \\ [max(\underline{x}^\#, \underline{Y^\#} + 1), \overline{x}^\#] & \text{if } \Box^\#_{cmp} = > \wedge\ \overline{x}^\# > \underline{Y^\#} \\ [max(\underline{x}^\#, \underline{Y^\#}), \overline{x}^\#] & \text{if } \Box^\#_{cmp} = \geq \wedge\ \overline{x}^\# \leq \underline{Y^\#} \\ \bot & \text{otherwise} \end{cases}$$

Overall for the comparison, case (I) is the lifting of unknown information. Hence, we will look closer at case (II) and (III), but only consider one side and comparison operator $\square_{cmp} = <$.

$$
\begin{aligned}
[[X \overset{\leftarrow}{<} Y]]^{\#} \iff & \alpha(\gamma(X^{\#}) \ < \ \gamma(Y^{\#})) \\
&= \alpha\left(\bigcup_{i=1}^{|X|} X_i < \bigcup_{i=1}^{|Y|} Y_i\right) \\
&= \alpha\left(\bigcup_{i=1}^{|X|} X_i < \ max(Y)\right) \quad \text{sortedness} \implies \text{last interval has highest element} \\
&= \alpha\left(\bigcup_{i=1}^{|X|} X_i < \ max(Y_N)\right) \\
&= \text{normalize}\left(\{x^{\#} < max(Y_N^{\#}) \mid x^{\#} \in X^{\#}\}\right) \quad \text{No new nor growing intervals} \\
&= \{x^{\#} < max(Y_N^{\#}) \mid x^{\#} \in X^{\#}\} \\
&\sqsubseteq X^{\#} \ <^{\#} \ Y^{\#} = \texttt{cmp}(X^{\#}, Y^{\#}, <^{\#}) \\
&= \left\{ \begin{cases} min(\underline{x}^{\#}, \overline{Y_n}^{\#} - 1), min(\overline{x}^{\#}, \overline{Y_n}^{\#} - 1) & \text{if } \underline{x}^{\#} < \overline{Y_n}^{\#} \\ \bot & \text{otherwise} \end{cases} \ \middle| \ x^{\#} \ \in X^{\#} \right\}
\end{aligned}
$$

Normalization will not result in any change as $X^{\#}$ must have had valid IntervalList properties (e.g. sortedness) and intervals can only either be removed fully (case (V) of comp') or be shrunken (case (I) - (IV) of `cut`).

Moreover, as part of the abstract approximation soundness we know that the abstract properties of $Y$ must contain all elements of the concrete, i.e. $\gamma(\underline{Y_1}^{\#}) \leq min(Y_1)$ and $max(Y_N) \leq \gamma(\overline{Y_N}^{\#})$. If an entire interval $x^{\#}$ is strictly bigger than $Y^{\#}$ this implies that $\forall a \in x : a > \gamma(min(Y_N))$, but only by the assertion $\forall a \in X : a < Y$. Hence, this comparison results in an empty ($\bot$) interval list and will be removed. If all intervals are strictly bigger than $Y^{\#}$ (case II of `cmp`), the entire test can never be valid as the abstract domain is required to fully include all concrete elements. Therefore, all concrete values must have been removed as well and the assertion cannot be valid ($\bot$).

Otherwise, in case (I) of `cmp` for all abstract intervals of $X^{\#}$ we can soundly cut them at $Y_n^{\#}$ as every value of $X^{\#}$ must be lower than $Y_n^{\#}$). Thus, $\forall x \in X : x < max(Y_N) \sqsubseteq \gamma([\underline{x}^{\#}, min(\overline{x}^{\#}, \overline{Y_n}^{\#} - 1)])$.

Apart from the equality backward test which uses list intersection (subsection 4.2.2),

these test can be done in $\mathcal{O}(N)$ as no normalization needs to be performed and every interval list operand needs to be traversed exactly once and the underlying `cut`, `ineq`'' per interval can be done in $\mathcal{O}(1)$.

### 4.2.7 Widening

As we have seen in Section 2.7, *widening* is a fixpoint approximation technique. In this section we have seen that a Galois connection between two lattices $(D_1, \subseteq)$ and $(D_2, \sqsubseteq)$ implies $\mathrm{lfp}(D_1) \subseteq \gamma(\mathrm{lfp}(D_2))$. After [CC92b, Prop 6.10] we can replace a widening with a *coarser widening* without affecting soundness if (1) abstract minimality is satisfied and (2) convergence of abstract iteration sequences exist, their limits are: $\alpha(F) \sqsubseteq F^{\#}$. The first assumption was shown in Section 3.9, and thus for overall soundness of IntervalList widening we only need to ensure that widening operator is (a) sound and (b) convergent. As $\mathcal{D}^I$ is a finite domain, termination and convergence (b) at a fixpoint must happen in finite steps. However, it is desirable to converge on a fixpoint in a reasonable amount of iterations. Moreover, it should be noted that the IntervalList abstract approximation is not an best approximation and neither is its widening. Hence, its widening only be guarantees to result in a post-fixpoint.

In general, the widening of the IntervalList domain tries to *preserve existing extremal points*. If possible, it will ignore intervals which occur unchanged in both $X^{\#}$ and $Y^{\#}$ and *only widen new or changed intervals*.

Astrée uses a few widening strategies: (I) widening with a *dynamic delay counter* $\delta$, (II) widening with a *static ramp*, (III) widening with a *dynamic ramp*, and (IV) widening with a *dynamic range*. Thus, its widening relation provides dynamic parameters which are forwarded to the next widening iteration: $\sqcup$: $\mathcal{D}^E \times \mathcal{D}^E \times \mathcal{K} \mapsto \mathcal{D}^E \times \mathcal{K}$. The dynamic information for the IntervalList domain $\mathcal{K}$ is a loop iteration counter of $\mathbb{N}$, a static ramp set $R_S$ in $\mathbb{P}(\mathbb{T})$ with a fixed maximum cardinality $|R_S| < k$, a dynamic threshold $R_T$ of $\mathbb{P}(\mathbb{T})$ with a fixed maximum cardinality $|R_T| < i$, and a dynamic range interval $R_D$ of $\mathbb{I}$. We summarize all ramp parameters as $R = (R_S, R_T, R_R)$. Hence the extra information $\mathcal{K}$ is $\mathbb{N} \times R$. The widening for a single variable in the IntervalList domain is:

$$[[X \;\sqcup\; Y]]^{\#}\,(\delta, R) \stackrel{\text{def}}{=} \begin{cases} \top\;(\delta, R) & \text{if } X^{\#} = \top \vee Y^{\#} = \top \\ X^{\#}\;(\delta, R) & \text{if } Y^{\#} \sqsubseteq X^{\#} \\ ((X^{\#} \sqcup^{\#} Y^{\#})R)\;(\delta+1, R) & \text{if } \texttt{delay}(\delta) \\ ((X^{\#} \sqcup'^{\#} Y^{\#})R)\;(\delta+1, R) & \text{otherwise} \end{cases}$$

The delay counter $\delta$ is incremented with every widening iteration except for the first and second case (widening terminations) in which no iterations are required. The

number of widening steps is limited by the number of delays allowed by `delay`$(\delta)$ and the number of ramp steps. Astrée allows the user to choose his trade-off of (a) widening with few ramp parameters ("quick widening") with a potentially higher loss in precision, but faster runtime and (b) widening with more ramp parameters, but higher runtime. In practice, the latter option is typically used for smaller codebases and only reduced to "quick widening" for large codebases with high runtimes of their code analysis.

As stated above, correctness for widening implies that (1) the widening chain is stable in finite time and (2) every iteration is a superset of its previous widening iteration:

$$\forall a^\#, b^\# \in A^\# : \gamma(a^\#) \subseteq \gamma(a^\# \sqcup b^\#) \ \wedge \ \gamma(b^\#) \subseteq \gamma(a^\# \sqcup b^\#)$$

The **termination cases** (I, II) are sound as $\top$ is the superset of all elements (I) and in the second case (II) a fixpoint has been reached ($Y^\# \sqsubseteq X^\#$ by definition and thus $X^\# \sqsubseteq X^\#$).

**Delaying iterations** (III). A special delay function can decide on whether to delay the widening and instead perform a union. This operation is sound by the definition of the union and as long as `delay`$(\delta)$ is truthy for only a constant of values termination and convergence is guaranteed. However, the trick is to find the balance between runtime and loss of precision for a specific application. An example of widening with delays can be seen in Figure 4.2.

Finally, we need to consider **actual widening (**IV). For this we consider the individual intervals and *partition into identical ($X^\#_{eq}$) and non-identical intervals* ($X^\#_{neq}, Y^\#_{neq}$). The underlying idea is to preserve and avoid widening unchanged extrema intervals $X^\#_{eq}$. Widening should only be performed on intervals which have been altered in the current iteration step, i.e. on partially matching or non-matching intervals (see Figure 4.3 for examples). The final widening is the union of the non-widened intervals $X^\#_{eq}$ and the widened intervals ($X^\#_{neq}, Y^\#_{neq}$):

$$\begin{aligned} X^\#_{eq} = Y^\#_{eq} &\stackrel{\text{def}}{=} \bigcup_{i=1}^{|X^\#|} (X^\#_i =^\# Y^\#_i) \\ X^\#_{neq} &= X^\# \setminus X^\#_{eq} \\ Y^\#_{neq} &= Y^\# \setminus X^\#_{eq} \\ (X^\# \sqcup'^\# Y^\#)R &= X^\#_{eq} \sqcup^\# ((X^\#_{neq} \sqcup''^\# Y^\#_{neq})R) \end{aligned} \tag{4.1}$$

```
int x = 0;
if (random) { x = 444; }
while (x < 300) {
  x += 5;
}
```

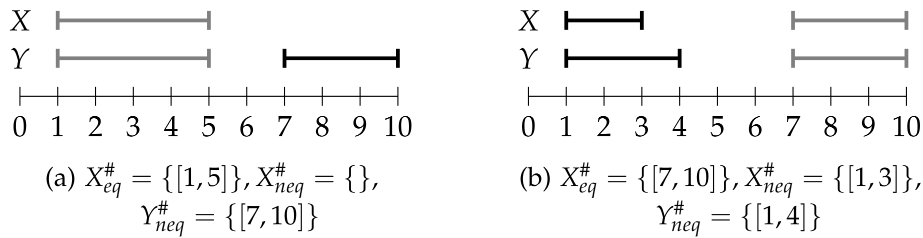
| step | x |
|---|---|
| 1 | $\{\{0\}, \{5\}, \{444\}\}$ |
| 2 | $\{\{0\}, [5, 10], \{444\}\}$ |
| 3 | $\{\{0\}, [5, 15], \{444\}\}$ |
| **4** | $\{\{0\}, [5, 15], \{444\}\} \sqcup \{\{0\}, [5, 20], \{444\}\} = \{\{0\}, [5, 41], \{444\}\}$ |
| 5 | $\{\{0\}, [5, 46], \{444\}\}$ |
| 6 | $\{\{0\}, [5, 51], \{444\}\}$ |
| **7** | $\{\{0\}, [5, 51], \{444\}\} \sqcup \{\{0\}, [5, 56], \{444\}\} = \{0\}, [5, 299], \{444\}\}$ |
| 8 | $\{0\}, [5, 304], \{444\}\}$ |

(a) Widening with delays at steps 1, 2, 3, 5, 6, 8 and widening at steps 4 and 7.

| step | x |
|---|---|
| 1 | $\{\{0\}, \{444\}\} \sqcup \{\{0\}, \{5\}, \{444\}\} = \{\{0\}, [5, 46]\{444\}\}$ |
| 2 | $\{\{0\}, [5, 46]\} \sqcup \{\{0\}, [5, 51], \{444\}\} = \{[0, 299], \{444\}\}$ |
| 3 | $\{[0, 299], \{444\}\} \sqcup \{[0, 304], \{444\}\} = \{[0, \texttt{int.max}]\}$ |

(b) Widening without delays

Figure 4.2: Interval list widening example with and without delays. $R_S = \{-46, 46\}, R_D = \{299\}, R_R = \{\texttt{int.min}, \texttt{int.max}\}$




(a) $X^{\#}_{eq} = \{[1, 5]\}, X^{\#}_{neq} = \{\},$ $Y^{\#}_{neq} = \{[7, 10]\}$

(b) $X^{\#}_{eq} = \{[7, 10]\}, X^{\#}_{neq} = \{[1, 3]\},$ $Y^{\#}_{neq} = \{[1, 4]\}$

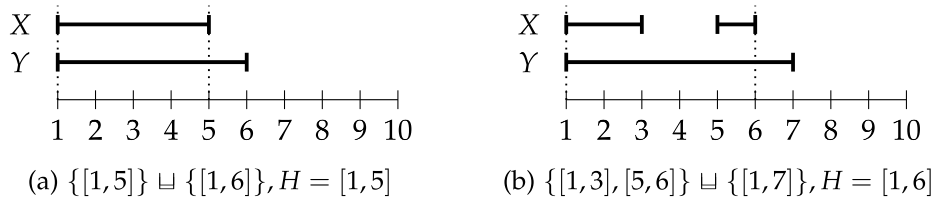
Figure 4.3: Interval list widening partitioning examples. Grey intervals are identical (eq), black intervals are mismatching (neq).

**Non-identical intervals ($X^{\#}_{neq}, Y^{\#}_{neq}$)**

The widening of $X^{\#}_{neq}$ with $Y^{\#}_{neq}$ is performed by widening each interval of $X^{\#}_{neq} \sqcup Y^{\#}_{neq}$ individually and distinguishing between two main cases: (I) existence of *partially matching intervals* in $X^{\#}_{neq}$ and (II) *no matching intervals* $X^{\#}_{neq}$. Additionally, when existent we use the closest interval (including identical intervals) on the left side (`prev`) and on the right side (`next`) as widening ramps to avoid widening over neighboring intervals in one widening step.

$$
\begin{aligned}
(X^{\#}_{neq} \sqcup'' Y^{\#}_{neq})R &\stackrel{\text{def}}{=} \left\{ \left\{ \begin{array}{ll} [\texttt{widen_hull}(c^{\#}, [\underline{H}^{\#}, \overline{H}^{\#}], R, & \text{if } \exists x^{\#} \in X^{\#}_{neq} : \\ \qquad\qquad prev^{\#}, next^{\#}) & overlap(x^{\#}, c^{\#}) \\ \texttt{widen}(c^{\#}, R, prev^{\#}, next^{\#}) & \text{otherwise} \end{array} \right. \;\middle|\; c \in (X^{\#}_{neq} \sqcup^{\#} Y^{\#}_{neq}) \right\} \\
prev^{\#} &= \begin{cases} z^{\#} & \text{if } \exists z^{\#} : \forall d^{\#} \in (X^{\#}_{eq} \sqcup^{\#} (X^{\#}_{neq} \sqcup^{\#} Y^{\#}_{neq})) : z^{\#} < c^{\#} \land \neg(d^{\#} < c^{\#} \land z^{\#} < d^{\#}) \\ \bot & \text{otherwise} \end{cases} \\
next^{\#} &= \begin{cases} z^{\#} & \text{if } \exists z^{\#} : \forall d \in (X^{\#}_{eq} \sqcup^{\#} (X^{\#}_{neq} \sqcup^{\#} Y^{\#}_{neq})) : c^{\#} < z^{\#} \land \neg(c^{\#} < d \land d^{\#} < z^{\#}) \\ \bot & \text{otherwise} \end{cases} \\
H^{\#} &= \{x^{\#} \mid x^{\#} \in X^{\#}_{neq} \land overlap(x^{\#}, c^{\#})\}
\end{aligned}
\tag{4.2}
$$

Case (I) represents intervals which overlap with at least one interval from the previous iteration ($X_{neq}$). In this case the respective interval can be widening against the hull of all respective overlapping intervals (see Figure 4.4 for examples).




(a) $\{[1,5]\} \sqcup \{[1,6]\}, H = [1,5]$ (b) $\{[1,3],[5,6]\} \sqcup \{[1,7]\}, H = [1,6]$

Figure 4.4: Interval list widening of partially matching intervals examples.

However, when the interval has no overlapping intervals (case II), it requires more distinctions (see Figure 4.5 for examples). If there are no intervals in $X_{neq}$, the interval was a newly added value. If the new interval was either added to the right (I) or left (II), widening will be performed only in one direction (see Figure 4.5 (a) and (b)). The motivation for this behavior is to preserve existing interval gaps during the widening. Otherwise, full widening must be performed in both directions, but should use `prev` and `next` (if available) as additional dynamic ramp parameters.

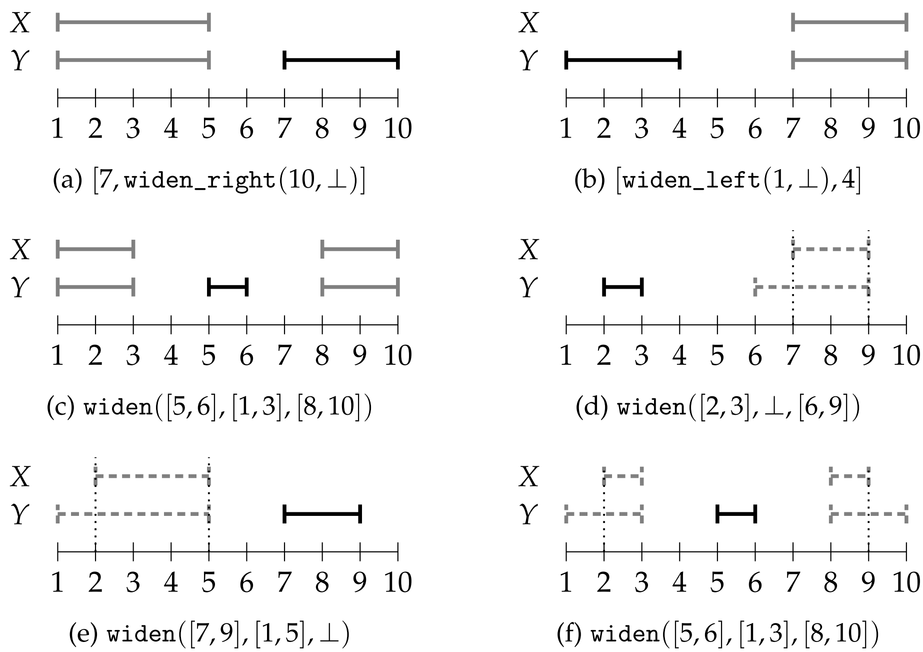



Figure 4.5: Interval list widening addition examples (without ramp parameters). The black interval of $Y$ is widened. Identical intervals are colored in solid gray lines and overlapping intervals are colored in dashed gray lines.

The widening of individual intervals takes into account a widening range $R_r$ (typically $[T_{min}, T_{max}]$), a pre-defined static widening ramp set ($R_s$) and dynamic threshold set ($R_D$).

$$\texttt{widen}(x^\#, R, prev^\#, next^\#) \stackrel{\text{def}}{=} \begin{cases} [\texttt{widen_left}(\underline{x^\#}, R, prev^\#), \overline{x^\#}] & \text{if } prev^\# = \bot \\ [\underline{x^\#}, \texttt{widen_right}(\overline{x^\#}, R, next^\#)] & \text{if } next^\# = \bot \\ [\texttt{widen_left}(\underline{x^\#}, R, prev^\#), & \text{otherwise} \\ \quad \texttt{widen_right}(\overline{x^\#}, R, next^\#)] & \end{cases}$$
$$\texttt{widen_hull}(x^\#, a^\#, R, prev^\#, next^\#) \stackrel{\text{def}}{=} \left[ \begin{cases} \underline{a}^\# & \text{if } \underline{a}^\# \leq \underline{x}^\# \\ \texttt{widen_left}(\underline{x^\#}, R, prev^\#) & \text{otherwise} \end{cases}, \begin{cases} \overline{a}^\# & \text{if } \overline{a}^\# \geq \overline{x}^\# \\ \texttt{widen_right}(\overline{x^\#}, R, next^\#) & \text{otherwise} \end{cases} \right] \tag{4.3}$$

The individual widening operations `widen_left` and `widen_right` are defined as:

$$\begin{aligned} \texttt{widen_left}(x^\#, R, prev^\#) &\stackrel{\text{def}}{=} \begin{cases} prev^\# & \text{if } prev^\# \neq \bot \wedge l' \leq prev^\# \\ l' & \text{otherwise} \end{cases} \\ &\text{with } l' = max(\underline{R_r}, \texttt{prevElement}(x^\#, R_S), \\ &\qquad \texttt{prevElement}(x^\#, R_D)) \\ \texttt{widen_right}(x^\#, R, next^\#) &\stackrel{\text{def}}{=} \begin{cases} next^\# & \text{if } next^\# \neq \bot \wedge next^\# \leq r' \\ r' & \text{otherwise} \end{cases} \\ &\text{with } r' = min(\overline{R_r}, \texttt{nextElement}(x^\#, R_S), \\ &\qquad \texttt{nextElement}(x^\#, R_D)) \end{aligned} \tag{4.4}$$

$$\begin{aligned} \texttt{nextElement}(x^\#, L) &= a^\# \in L : x^\# \leq a^\# \wedge \nexists b^\# \in L : x^\# \leq b^\# \wedge b^\# \leq a^\# \\ \texttt{prevElement}(x^\#, L) &= a^\# \in L : a^\# \leq x^\# \wedge \nexists b^\# \in L : b^\# \leq x^\# \wedge a^\# \leq b^\# \end{aligned} \tag{4.5}$$

With this widening strategy identical intervals are kept and mismatching intervals ($X^\#_{neq} \sqcup Y^\#_{neq}$) are widened. The additional normalization during the union can only increase the interval list to a superset. We can summarize for $Z^\# = X^\# \sqcup'^\# Y^\#$:

1. $X^\#_{eq} \sqsubseteq Z^\# \wedge Y^\#_{eq} \sqsubseteq Z^\#$ (Equation 4.1)

2. $\forall a^\# \in \mathbb{T} : \texttt{prevElement}(a^\#) \leq a^\# \wedge a^\# \leq \texttt{nextElement}(a^\#)$ (Equation 4.5)
3. $\forall a^\# \in \mathbb{T} : \texttt{widen_left}(a^\#) \leq a^\# \wedge a^\# \leq \texttt{widen_right}(a^\#)$ (Equation 4.4)
4. $\forall A^\# \in \mathbb{I} : A^\# \sqsubseteq \texttt{widen}(A^\#)$ (Equation 4.3)
   a) $A^\# \sqsubseteq [\texttt{widen_left}(\underline{A}^\#), \overline{A}^\#]$ if left-most interval
   b) $A^\# \sqsubseteq [\underline{A}^\#, \texttt{widen_right}(\overline{A}^\#)]$ if right-most interval
   c) $A^\# \sqsubseteq [\texttt{widen_left}(\underline{A}^\#), \texttt{widen_right}(\overline{A}^\#)]$
5. $\forall A^\# \in \mathbb{I} : A^\# \sqsubseteq \texttt{widen_hull}(A^\#)$ (Equation 4.3)
   a) $A^\# \sqsubseteq [\underline{H}^\#, \overline{H}^\#]$ when $\underline{H}^\# \leq \underline{H}^\# \wedge \overline{H}^\# \leq \overline{H}^\#$
   b) $A^\# \sqsubseteq [\underline{H}^\#, \texttt{widen_right}(\overline{A}^\#)]$ when $\underline{H}^\# \leq \underline{A}^\#$
   c) $A^\# \sqsubseteq [\texttt{widen_left}(\underline{A}^\#), \overline{H}^\#]$ when $\overline{A}^\# \leq \overline{H}^\#$
   d) $A^\# \sqsubseteq [\texttt{widen_left}(\underline{A}^\#), \texttt{widen_right}(\overline{A}^\#)]$
6. $X^\#_{neq} \sqsubseteq C^\# \wedge Y^\#_{neq} \sqsubseteq C^\#$ with $C^\# = \{op(c^\#) \mid c^\# \in (X^\# \sqcup''^\# Y^\#)\}$ (Equation 4.2)
   a) $c^\# \sqsubseteq op(c^\#) = \texttt{widen_hull}(c^\#)$
   b) $c^\# \sqsubseteq op(c^\#) = \texttt{widen}(c^\#)$
7. $C^\# \sqsubseteq Z$ (Equation 4.1)

Each individual widen step for mismatching intervals ($X^\# \sqcup''^\# Y^\#$) can only use *extensive* operations. Therefore, we conclude for the last step (IV) of $\sqcup^\#$ that $X^\# = \{X^\#_{eq}, X^\#_{neq}\} \sqsubseteq (X^\# \sqcup'^\# Y^\#)$ and $Y^\# = \{Y^\#_{eq}, Y^\#_{neq}\} \sqsubseteq (X^\# \sqcup'^\# Y^\#)$

Additionally, we experimented with a few variants of IntervalList widening. For example, we tested an inserting additional dynamic ramps based on the distance of the current interval to its respective *prev* and *next* intervals with an *inverse binary exponential backoff*. Hence, an additional ramp could be given depending on a defined stop distance $c$ (e.g. 100):

$$\texttt{right_ramp}(x, next) = \begin{cases} |x - next|/2 & \text{if } next \neq \bot \wedge |x - next| > c \\ next & \text{otherwise} \end{cases}$$

$$\texttt{left_ramp}(x, prev) = \begin{cases} |prev - x|/2 & \text{if } prev \neq \bot \wedge |prev - x| > c \\ prev & \text{otherwise} \end{cases}$$

However, we did not find precision improvements with this additional ramp.

# 5 Results

The IntervalList domain has been implemented as an optional domain for the Astrée Static Analyzer. For this evaluation we measured the impact of the IntervalList domain on real-world industrial software from the automotive and avionics domain. The codes have been anonymized and will only be described by parameters like code size, number of functions, or number of tasks. For each of these codes, a configuration for an Astrée analysis was already available prior to our evaluation. Analyzer settings correspond to typical industrial usage scenarios. We used the same subset of abstract domains, semantic and precision options, and checks for coding guidelines. We compared the results and performance of a baseline run (IntervalList abstract domain disabled) against various improvement runs (IntervalList abstract domain enabled with specific options). All evaluations were performed with a development version of Astrée 20.10.

It should be noted that all analysis configurations have already been optimized to obtain high quality results from the analyzer (see [DS07] for an overview of false alarm reduction technique). Moreover, the IntervalList domain was only implemented for integers and has little effect on codes which perform extensive floating-point calculations. Thus, we only considered projects which use integer variables to a significant degree. Finally, analysis configurations for larger codes typically enable only a small subset of domains to allow for an analysis within acceptable time and memory bounds. We therefore expect a slight bias of the effectiveness towards larger code bases. However, since these codes often cannot be analyzed successfully with some of the advanced abstract domains, we believe this to be a fair assessment of the impact of the IntervalList domain as it measures against the status quo.

First, we summarize the impacts on multiple smaller industry codes (Section 5.1). Afterwards, we have a closer look at the impact of the specialized domain to a larger industry code (Section 5.2) and finally we show individual code patterns for which the IntervalList domain can provide better approximations (Section 5.3).

## 5.1 Medium-sized project evaluation

In this section we evaluate the impact of the IntervalList domain on 19 small- and medium-sized industry codes from mostly automotive and avionic software. We provide only summarized overall and per-project overviews. The analyzed statements in these projects are within the range of $5.000 - 100.000$ and on average have approximately 30.000 statements. All analyses were run on a workstation with an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz processor and 64 GB RAM.

In the overview tables below the baseline run ($N = 0$) is stated in absolute values whereas results from individual runs with the IntervalList domain enabled from $N = 2$ to $N = 7$ are given relative to their respective baseline. The three major findings of this evaluation are:

1. $0.3 - 0.4\%$ more unreachable statements (relative to all prior reachable code statements) and up to 1.4% more unreachable code alarms (subsection 5.1.1).
2. 0.3% less false run-time errors over all analyzed code statements (1.3% per project) and up to 8.4% less false run-time alarms in individual projects (subsection 5.1.2).
3. $7 - 11\%$ overall ($5 - 8\%$ per project) memory usage increase (subsection 5.1.3).

### 5.1.1 Unreachable code alarms and newly proven unreachable code statements

Overall the IntervalList domain proves about 0.4% more unreachable statements (see Table 5.1 for details) relative to all prior reachable code statements in the baseline run. Starting with $N = 3$ we observed only minimal improvements for higher $N$. This is an expected finding as most improvements were intended to originate from a *better abstraction of extremal variables* with an uninitialized state which already happens at $N = 2$. Further improvements are often due to better abstraction of state variables, but most projects do not have complex branching logic or state variables with more than two constant holes. Hence, we observe further improvements with higher $N$ of 5 or 7 significantly only in the outlier projects. For example, from $N = 3$ to $N = 5$ we can only observe an overall improvement of 0.01%. Lastly, we note that the impact highly depends on whether the project uses code patterns that the IntervalList domain can detect (range is from no change to 1.73%).

| | Overall | | Per project | | | | |
|---|---|---|---|---|---|---|---|
| N | abs. | rel. | mean | median | min | max | std |
| 0 | 202858 | | 10676.7 | 5632.0 | 0 | 44800 | 12390.6 |
| 2 | +547 | 0.27% | 0.28% | 0.21% | 0.0% | 1.73% | 0.38% |
| 3 | +779 | 0.38% | 0.4% | 0.28% | 0.0% | 1.73% | 0.43% |
| 5 | +796 | 0.39% | 0.4% | 0.28% | 0.0% | 1.73% | 0.43% |
| 7 | +802 | 0.39% | 0.41% | 0.28% | 0.0% | 1.73% | 0.43% |

Table 5.1: Overall newly proven unreachable code statements.

For an alternative overview we consider newly found unreachable code alarms. This is the number of instances for which the analyzer with the help of the IntervalList domain could prove that for all instances a condition can never occur whereas the first metric captures the actual unreachable statements. As we can see in Table 5.2 with this metric the overall improvement is more than double when we only count unreachable code statements and at more than 1.3% and the maximal project improvement was 2.33%.

However, this metric reveals that for some outlier projects with lower $N$ the analyzer is able to prove less code as unreachable. This occurs in projects with concurrency and is a result of a different behavior in parallel widening when local precision improvements increase the number of required widening iterations for finding a fixpoint. In particular, at the moment the Astrée static analyzer uses global widening to find a fixpoint over the individual process states. In its current implementation, the Astrée static analyzer delays parallel widening for a fixed amount of widening steps and otherwise defines the widen operation to always result in the $\top$ element of a domain. In such worst-case projects, local precision improvements lead to a more delayed parallel widening iterations and a worsened approximation ($\top$) and possibly its transitive dependencies.

| N | Overall | | Per project | | | | |
|---|---|---|---|---|---|---|---|
| | abs. | rel. | mean | median | min | max | std |
| 0 | 4089 | | 215.2 | 0.0 | 0 | 2453 | 599.9 |
| 2 | +19 | 0.46% | 0.21% | 0.58% | -0.77% | 0.82% | 0.86% |
| 3 | +48 | 1.16% | 1.02% | 1.35% | -0.39% | 2.1% | 1.27% |
| 5 | +55 | 1.33% | 1.22% | 1.47% | -0.13% | 2.33% | 1.25% |
| 7 | +57 | 1.37% | 1.31% | 1.47% | 0.13% | 2.33% | 1.11% |

Table 5.2: Overall new unreachable alarms.

### 5.1.2 Run-time error findings

In Table 5.3 we provide an overview of the change in run-time error findings. We can observe that there are huge discrepancies between the individual projects as the relative run-time error alarm findings can be reduced by more than 8 % or even be slightly increased by 0.36%. However, these projects have a quite different application, style, and size. The increase in run-time error findings in the worst project observed comes from projects with concurrency and the above mentioned problem of locally improved approximations increasing the number of required global parallel widening iterations and approximating to $\top$ after the parallel widening delay. Overall, we observed an improvement of 0.28% (1.1% per project) less false run-time errors with $N = 2$ and a smaller additional improvement of 0.35% (1.22% per project) when storage for a second hole ($N = 3$) is provided, and another slight improvement by 0.01% (0.03% per project) for $N = 5$. However, for bigger interval list sizes ($N = 7$) almost no improvements were observed.

| N | Overall | | Per project | | | | |
|---|---|---|---|---|---|---|---|
| | abs. | rel. | mean | median | min | max | std |
| 0 | 74866 | | 3940.3 | 990.0 | 0 | 18144 | 6136.1 |
| 2 | -208 | -0.28% | -1.1% | -0.22% | -8.42% | 0.36% | 2.17% |
| 3 | -260 | -0.35% | -1.22% | -0.34% | -8.42% | 0.27% | 2.13% |
| 5 | -266 | -0.36% | -1.25% | -0.37% | -8.42% | 0.27% | 2.14% |
| 7 | -267 | -0.36% | -1.28% | -0.37% | -8.42% | 0.27% | 2.16% |

Table 5.3: Overall change in run-time error findings.

### 5.1.3 Analysis duration and memory usage

We measured the overall analysis duration, but the results were non-deterministic and hard to reproduce. We observed values in $-46\% - 55\%$. Furthermore, the results were impacted by the exact development version of Astrée used and the current load of the workstation. Hence, we have decided to omit these unreliable measurements and direct the reader to either the memory usage measurements below or the moderately stable analysis time comparison with the larger industry code example in subsection 5.2.3.

In Table 5.4 we list the measured memory usage between the baseline version and different version of the IntervalList domain. While these results were reproducible, the individual variations are less obvious. However, we can say that the average overhead cost of the IntervalList domain is between $7 - 11\%$ ($5 - 8\%$ per project). In some projects proving more code statements unreachable can lead to less partitioning or smaller partition spaces and, thus an overall memory usage reduced by up to 18%. However, it is also possible for the opposite to happen and memory usage to drastically increase (up to 40%). We measured this in two outlier projects with many concurrent processes in which local precision improvements lead to more local fixpoint iterations and, thus a different global fixpoint, both leading to higher memory usage.

| | Overall | | Per project | | | | |
|---|---|---|---|---|---|---|---|
| N | abs. | rel. | mean | median | min | max | std |
| 0 | 44224 | | 2327.6 | 1940.0 | 0 | 6501 | 1877.9 |
| 2 | +3540 | 7.41% | 5.35% | 2.13% | -17.69% | 31.79% | 13.85% |
| 3 | +5306 | 10.71% | 7.92% | 11.12% | -14.95% | 39.94% | 13.06% |
| 5 | +5272 | 10.65% | 7.22% | 7.13% | -18.07% | 39.94% | 14.15% |
| 7 | +3712 | 7.74% | 5.58% | 7.12% | -18.07% | 18.49% | 10.81% |

Table 5.4: Overall memory usage in MB.

## 5.2 Large industry example

We evaluated the effects of the IntervalList domain on a bigger industrial code of an automotive software. This code contains approximately 5000 functions and has an overall size of 400.000 physical lines of source code (comment and whitespace lines excluded). The number of recursive paths in its call graph was slightly below 10.000. Furthermore, this code spawns up to 26 different processes. The analysis reaches approximately 71.200 of the overall XXX (redacted) statements.

All analyses were performed on a workstation with an Intel(R) Xeon(R) CPU E5-1620 v4 @ 3.50GHz processor and 256 GB RAM. An overview of the results can be found in Table 5.6. We observed six major findings:

1. Statically proven unreachable code increased by 0.5% and unreachable code alarms increased by 5.6% (subsection 5.2.1).
2. False run-time errors reported by the Astrée static analyzer decreased by 0.6% (subsection 5.2.2).
3. Memory usage reduction by $18 - 38\%$ (subsection 5.2.3).
4. Analysis runtime increase by $7 - 22\%$ (subsection 5.2.3).
5. No further improvements in neither unreachable code lines nor alarms with interval list sizes higher than 4 (subsection 5.2.1 and subsection 5.2.2).
6. Analysis time overhead and memory usage minimum at $N = 5$ (subsection 5.2.3).

We will first discuss these overview findings and afterwards provide real examples for recognized patterns by the IntervalList domain.

### 5.2.1 Unreachable code statements

One of the major benefits of the extremal domain is that it helps to prove code statements as unreachable:

| | Reachable code | | Unreachable alarms | |
|---|---|---|---|---|
| N | abs. | rel. | abs. | rel. |
| 0 | 58383 | | 3928 | |
| 2 | -254 | -0.44 % | +183 | 4.66% |
| 3 | -316 | -0.54 % | +230 | 5.86% |
| 4 | -312 | -0.54 % | +222 | 5.65% |
| 5 | -312 | -0.54 % | +222 | 5.65% |
| 7 | -312 | -0.54 % | +222 | 5.65% |

Table 5.5: Overview of reachable code statements and unreachable code alarms. Rows $N = 2$ to $N = 7$ are relative to the prior results from the baseline run ($N = 0$).

Apart from detecting dead code with one interval hole (i.e. extremal values) which motivated the design of this domain, variables with two holes are very common as well. For example, they could be part of a state machine which can now be proven to have unreachable cases. In subsection 5.3.1 we provide individual examples of code pattern from this project that can now be proven to be unreachable.

### 5.2.2 Alarm findings

We provide an overview of differences in *run-time error findings* in Table 5.6. We listed *flow anomaly alarms* and *data race alarms* separately are they can only be influenced indirectly by the IntervalList domain. An example of a flow anomaly alarm is an alarm about the read of a global variable that has not been proven to be initialized (see subsection 5.3.4 for an example).

Additionally, Astrée has a vast category of rule checks that can be enabled. For example, one such rule check is to warn the user about invariants that are truthy or falsy in all paths (see subsection 5.3.3 for a detailed example). However, as projects use different rules checks and rule checks themselves only warn the user about a potential logic error or code style problem, rule checks are difficult to compare and have not been listed in the overview table. However, we provide a detailed listing of all individual alarm changes in Table 5.7 and included impacted rule check categories in this table.

In summary, we observe up to 30 (0.6%) less run-time errors. The four individual alarm run-time error categories that were improved are: (I) conversion overflow, (II) field overflow upon dereference, (III) array out-of-bounds error, (IV) invalid pointer arithmetic, and (V) overflow in arithmetic. Moreover, no changes in the data race alarm category were observed, but a similar percentage of false flow anomalies was removed. As with unreachable code we see the biggest improvement across all categories with $N = 2$ already and only a small increase for $N = 3$. As before, with more intervals ($N = 4$) there are even additional improvements and no improvements for $N > 4$ were observed.

| N | Runtime errors abs. | Runtime errors rel. | Flow anomalies abs. | Flow anomalies rel. | Alarms overall abs | Alarms overall rel. |
|---|---|---|---|---|---|---|
| 0 | 5005 | | 2815 | | 7856 | |
| 2 | -22 | -0.44 % | -18 | -0.64 % | -40 | -0.51 % |
| 3 | -27 | -0.54 % | -19 | -0.67 % | -47 | -0.60 % |
| 4 | -30 | -0.60 % | -18 | -0.64 % | -48 | -0.61 % |
| 5 | -30 | -0.60 % | -18 | -0.64 % | -48 | -0.61 % |
| 7 | -30 | -0.60 % | -18 | -0.64 % | -48 | -0.61 % |

Table 5.6: Overview of runtime-error alarm changes, flow anomalies alarms, and overall alarm changes. Data races alarms were not explicitly stated as they remained constant (36), but were included in the overall alarm column.

In the more detailed overview of the individual alarm categories Table 5.7 we can see that only two run-time error categories improve for $N > 2$: (I) overflow in conversion errors and (II) out-of-bound array access errors. For flow anomaly alarms and rule checks, however, we observe small improvements up to $N = 5$. The increase by one in the "overflow in arithmetic" error is due to more parallel widening iterations which exceeds the initial widening delay and triggers a parallel widening which at the moment widens to $\top$ and an information loss compared to the baseline.

| Finding category | N=0 | N=2 | N=3 | N=4 | N=5 |
| --- | --- | --- | --- | --- | --- |
| Arithmetics on invalid pointers | 301 | -9 | -9 | -9 | -9 |
| Out-of-bound array access | 145 | -5 | -7 | -7 | -7 |
| Possible overflow upon dereference | 720 | -3 | -3 | -3 | -3 |
| Overflow in conversion | 1435 | -5 | -11 | -11 | -11 |
| Overflow in arithmetic | 689 | **+1** | **+1** | **+1** | **+1** |
| Incorrect field dereference | 15 | -1 | -1 | -1 | -1 |
| Read of global/static variable before init | 2791 | -18 | -19 | -19 | -20 |
| Controlling invariant | 1830 | +55 | +67 | +67 | +67 |
| Array index range | 144 | -5 | -8 | -8 | -18 |
| Unused suppress directives | 1101 | 0 | +1 | +1 | +1 |

Table 5.7: Individual alarm changes in project X. N=7 was omitted as there were no changes over N=5. The first block lists run-time errors, the second flow anomaly alarms (read of a global/static variable), and the last block rule check violations.

### 5.2.3 Analysis runtime and memory usage

In Table 5.8 we provide an overview of the impact of the IntervalList domain on the analysis runtime and memory usage. The two overall patterns are: (1) significant decrease in memory usage by $18-38\%$ whereas (2) analysis runtime increases by $8-22\%$.

In greater detail we can observe that memory usage decrease monotonically improves until $N=5$ and only for a high $N=7$ this trend stops. As we did not observe any additional alarm improvements for $N=7$ the slight increase by 0.1% comes from the overhead of the longer list data structure and more costly arithmetic operations.

For the analysis duration we can observe a similar trend with a minimum of the least overhead at $N=5$. The outlier $N=3$ is harder to explain and might be the consequence of more precision and, thus more iterations from $N=2$ to $N=3$, but a significant pruning improvement through unreachable code with $N=4$.

| N | Memory (MB) absolute | Memory (MB) relative | Duration (mins) absolute | Duration (mins) relative |
|---|---|---|---|---|
| 0 | 55260 | | 642 | |
| 2 | -10006 | -18.1 % | +115 | +17.9 % |
| 3 | -17423 | -31.5 % | +139 | +21.7 % |
| 4 | -17400 | -31.5 % | +86 | +13.4% |
| 5 | -21071 | -38.1 % | +50 | +7.8 % |
| 7 | -20983 | -38.0 % | +62 | +9.7 % |

Table 5.8: Memory (MB) usage and runtime duration (mins) in project X.

## 5.3 Code patterns of improved alarms

We will have a more detailed look into the individual findings and show what patterns the IntervalList domain helps to detect. Code examples were taken from the previously discussed industry project (Section 5.2), but have been obfuscated and simplified, so that they only show an outline of the pattern involved.

### 5.3.1 Unreachable code

One important aspect of the IntervalList domain is that it stores more value information about a variable than the traditional interval, bitfield or finite set abstract domains. In particular, and in contrast to the traditional interval range, it is aware of holes. For example, this allowed for the identification of unreachable if-clauses (see Listing 5.1) and the marking of certain switch-cases (see Listing 5.2) as unreachable. In the following example (Listing 5.1) the prior information about `programVar` was `[0, 11]`, but with IntervalList domain the variable was approximated to `{[0, 8], 11}` and thus the analyzer is aware of the hole:

Listing 5.1: Unreachability via conditions

```
if(programVar == 10u)
{
  ... // now unreachable
}
```

A variation is a switch block in Listing 5.2 for which certain cases can now be proven to be unreachable. This is because state would traditionally only be `[0, 10]`, but now it is `{[0, 5], [7, 10]}` and `STATE_G` can correctly be identified as unreachable:

Listing 5.2: Unreachability via switch blocks

```
switch (state) {
 case STATE_A: // 0
    ...
    break;
 ...
 case STATE_G: // 6
    ... // now unreachable
    break;
 ...
}
```

The most common dead branch identified is the `default` case. The IntervalList

domain can also directly create holes through refinement. In a simplified example (Listing 5.3) taken from this project, `mode` is known to have range of `[1, 3]` at the begin of program block. As the interval domain cannot memorize holes, it cannot use the negation of `2u == mode` for refining its approximation. However, the IntervalList domain is able to memorize holes from ranges and can store a better approximation of `mode` at the first `else if` as `{1, 3}`. After, two more refinement tests operations the IntervalList domain represents `mode` as $\perp$ and can prove the final `else` body to be unreachable:

Listing 5.3: Unreachability via refinement

```
unsigned char mode; // [1, 3]
if(2u == mode) {
  // ...
} else if(1u == mode) { // {1, 3} /\ [1, 3]
  // ...
} else if(3u == mode) { // {3} /\ [2, 3]
    // ...
} else {
   // now unreachable, before: [2, 2]
}
```

Incidentally, when program nodes can be proven to be unreachable this very often leads to a decrease in runtime alarms as all alarms appearing in respective unreachable code can no longer be triggered. For example, in Listing 5.4, removal of the `default` case in `FunA` increased the precision of the return value of `FunA` to be non-zero and thus in this cascade the analyzer can determine `a` to always be 1 after the loop:

Listing 5.4: Transitive unreachability improvements

```
unsigned char a = 1;
for(unsigned long i = 0; i <= arr_length; i++) {
  if(!(0x00u == FunA(i, ...)) {
    a = 0; // now unreachable
    break;
  }
}
if (a == 1) { ... } else {
  // now unreachable
}
```

### 5.3.2 Array Out-of-Bounds

In many projects maximal values of a variable are used to symbolize invalid states. The IntervalList domain allows to store these extremal states and avoids the otherwise common Out-of-Bounds alarms. An example of this pattern taken from the analyzed industry project is provided in Listing 5.5. Here, function `fun1` returned a variable whose state would normally be approximated by `[0, 255]`, but now it can be approximated better with `{0, [254, 255]}` and, subsequently the extremal values can be removed precisely. Afterwards, the approximation of `text` is not `[0, 253]` as in the interval domain, but `0`:

Listing 5.5: Array Out-of-Bounds alarm

```
unsigned char fun1() {
  unsigned char r = 254;
  do {
    if (...) {
      r = 0;
    } else {
      r = 255;
    }
  } while (...);
  return r;
}
unsigned char fun2() {
  unsigned char r = fun1(); // {0, [254, 255]} /\ [0, 255]
  if (r == 254) return 1;
  if (r == 255) return 1;
  unsigned char v = arr[r].member; // {0}, no out-of-bound
  ...
}
```

### 5.3.3 Invariants

An invariant alarm is triggered when expressions in conditionals, if- or iteration-statements are invariant for each evaluated context (e.g. `1 == 1` or `0 == 1`). In the industry example (Listing 5.6) the state of `error` has been `[36, 255]` in the baseline run. With the IntervalList domain the approximation was improved to `{36, [39, 42], 255}` and thus the analyzer can prove that `error` can never be 254:

Listing 5.6: Controlling invariant

```
unsigned char error = 255;
if (...) {
   error = 36;
} else if (...) {
   error = 39;
} else if (...) {
   error = 41;
} else if (...) {
   error = 42;
}
... // error = {36, [39, 42], 255} /\ [36, 255]
if (error == 254) { // INVARIANT ERROR
  ...
}
```

### 5.3.4 Read of a not written global variable

The IntervalList domain can also help to improve the precision of tracking of reads to potentially unwritten variables. In the example in Listing 5.7 the read of global variable `globalVar` in `fun3` triggers an alarm as the uninitialized global might be read before it has been initialized. The IntervalList domain approximates the value of `r` better in `fun` to `{[0, 1], 10}` instead of `[0, 10]`. In the else-body the negation of `r == 10` condition leads to a more precise refinement of `r` instead of before `[0, 9]` to `{[0, 1]}`. Hence, the analyzer can prove that `fun2` writes to `globalVar` in all cases and subsequently the variable must be initialized in `fun3`. Consequently, the false invariant alarm does not get reported with the IntervalList domain in `fun3`.

Listing 5.7: Read of a not written global variable

```
unsigned char globalVar;
void fun1() {
  unsigned char r = 10;
  if (...) {
    r = 0;
  } else {
    r = 1;
  }
  // r = {[0, 1], 10} /\ [0, 10]
  ...
  if (r == 10) { ...} else {
    fun2(r);
    fun3();
  }
}
void fun2(unsigned char r) { // r = {[0, 1]} /\ [0, 9]
  switch (r) {
    case 0:
      globalVar = 2;
      break;

    case 1:
      globalVar = 4;
      break;
    ...
    default:
      break;
  }
}
void fun3() {
  if (globalVar == 2) {
    ... // No read of unwritten global variable error
  }
}
```

# 6 Summary

In this work - motivated by code patterns from industrial software - we discussed Abstract Interpretation for variables with values in disjunctive intervals. For this problem we proposed a dedicated abstract value domain - the IntervalList domain (Chapter 3). This specialized abstract domain was implemented and integrated in the Astrée Static Analyzer (Chapter 4). We evaluated the impact on 19 medium-sized industry codes from mostly automotive and avionic software (Section 5.1) and one large industrial automotive code (Section 5.2). In addition, we presented examples of code patterns for which the IntervalList domain can help to reduce precision loss (Section 5.3).

In summary, we observed that the IntervalList domain is able to prove more unreachable code blocks on average by 1.4% in medium-sized code and by 5.5% in large code. Relative to all prior reachable statements, this resulted in 0.4 (medium-sized) to 0.5% (large) more code statements being proven as unreachable. Furthermore, the domain helped to reduce the number of reported false run-time errors overall by 0.3% in medium-sized code (on average per project by 1.3%) and 0.6% in the large code example for which many advanced domains cannot be enabled.

The impact on memory usage depends heavily on the code as improvements in proven unreachable code can either reduce the for partitioning or decrease its space or through general list allocation overhead and more iterations lead to an increase. For the medium-sized codes increased overall about $7-11\%$ ($5-8\%$ on average per project) and for the large industry code memory usage significantly decreased by $18-38\%$. In medium-sized code the analysis time varied drastically based on the code and external factors (version of the Astrée development build, load of the workstation), but in the large industry code example we measured an analysis run-time increase of $7-22\%$ which is mostly caused by more widening iterations due to more retained local precision.

As the IntervalList domain was implemented as an optional domain, performance improvements are expected with a tighter integration into the traditional interval domain as duplicated computational efforts can be avoided. Additionally, the results show that for $N = 2$ a majority of all improvements can be observed ($50-80\%$) and most gains are reached for $N = 3$ ($> 90\%$). Hence, another improvement would be to use 3 as a fixed list size and optimize the implementation accordingly.

In conclusion, the IntervalList domain trades analysis run-time and memory usage for less precision loss and our experimental results show that this trade-off can be a good choice for certain real-world industrial code.

# List of Figures

1.1 Visualization of different improvements to the interval abstract domain 4

2.1 Hasse diagram of the interval abstract domain $(\mathcal{D}^I, \sqsubseteq)$ . . . . . . . . . . 8

3.1 Examples of non-unique interval representations . . . . . . . . . . . . . . 19
3.2 Hasse diagram of the IntervalList abstract domain $(\mathcal{D}^E, \sqsubseteq)$ for $\{-1, 0, 1\}$ 22
3.3 Hasse diagram of the IntervalList abstract domain $(\mathcal{D}^E, \sqsubseteq)$ for $\{0, 1, 2, 3\}$ 22
3.4 Example of an IntervalList intersection . . . . . . . . . . . . . . . . . . . . 26

4.1 Backward inequality test examples for the IntervalList . . . . . . . . . . . 42
4.2 Example of IntervalList widening with and without delays . . . . . . . . . 47
4.3 Example of partitioning in IntervalList widening . . . . . . . . . . . . . . 47
4.4 Example of IntervalList widening for partially matching intervals . . . . 48
4.5 Example of IntervalList widening with additional intervals . . . . . . . 49

# List of Tables


5.1 Overall newly proven unreachable code statements . . . . . . . . . . . . 54
5.2 Overall new unreachable alarms . . . . . . . . . . . . . . . . . . . . . . . . 55
5.3 Overall change in run-time findings . . . . . . . . . . . . . . . . . . . . . 55
5.4 Overall memory consumption . . . . . . . . . . . . . . . . . . . . . . . . . 56
5.5 Unreachable code changes for industry example X . . . . . . . . . . . . 58
5.6 Overview of changes in run-time error findings for industry example X 59
5.7 Individual alarm changes in project X . . . . . . . . . . . . . . . . . . . . 60
5.8 Memory usage and runtime duration in project X . . . . . . . . . . . . . 61

# Bibliography

[Ber+10] Julien Bertrane et al. "Static analysis and verification of aerospace software by abstract interpretation." In: *AIAA Infotech@ Aerospace 2010*. 2010, p. 3385.

[Bir48] Garrett Birkhoff. "Lattice theory, rev. ed." In: *Amer. Math. Soc. Colloq. Publ.* Vol. 25. 1948.

[CC76] Patrick Cousot and Radhia Cousot. "Static determination of dynamic properties of programs." In: *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252.

[CC79] Patrick Cousot and Radhia Cousot. "Systematic design of program analysis frameworks." In: *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1979, pp. 269–282.

[CC92a] Patrick Cousot and Radhia Cousot. "Abstract interpretation and application to logic programs." In: *The Journal of Logic Programming* 13.2-3 (1992), pp. 103–179.

[CC92b] Patrick Cousot and Radhia Cousot. "Abstract interpretation frameworks." In: *Journal of logic and computation* 2.4 (1992), pp. 511–547.

[CC93] Patrick Cousot and Radhia Cousot. "Galois connection based abstract interpretations for strictness analysis." In: *Formal Methods in Programming and Their Applications*. Springer. 1993, pp. 98–127.

[Cou+05] Patrick Cousot et al. "The ASTRÉE analyzer." In: *European Symposium on Programming*. Springer. 2005, pp. 21–30.

[Cou+09] Patrick Cousot et al. "Why does Astrée scale up?" In: *Formal Methods in System Design* 35.3 (2009), pp. 229–264.

[Cou00] Patrick Cousot. "Abstract interpretation: Achievements and perspectives." In: *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*. 2000.

[DP02] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[DS07] David Delmas and Jean Souyris. “Astrée: from research to industry.” In: *International Static Analysis Symposium*. Springer. 2007, pp. 437–451.

[Min04] Antoine Miné. “Weakly relational numerical abstract domains.” PhD thesis. 2004.

[SWH12] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Compiler Design: Analysis and Transformation*. Springer Science & Business Media, 2012.