HAL
open science

Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation

Antoine Miné

To cite this version:

Antoine Miné. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. Foundations and Trends in Programming Languages, 2017, 4 (3-4), pp.120-372. 10.1561/2500000034 . hal-01657536

HAL Id: hal-01657536

https://hal.sorbonne-universite.fr/hal-01657536

Submitted on 1 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation

Antoine Miné
Sorbonne Universités, UPMC Univ. Paris 06, CNRS, LIP6
antoine.mine@lip6.fr

Contents

1 Introduction **3**
1.1 A First Static Analysis: Informal Presentation . . . 4
1.2 Scope and Applications . . . 10
1.3 Outline . . . 15
1.4 Further Resources . . . 15

2 Elements of Abstract Interpretation **17**
2.1 Order Theory . . . 18
2.2 Fixpoints . . . 27
2.3 Approximations . . . 31
2.4 Summary . . . 43
2.5 Bibliographic Notes . . . 43

3 Language and Semantics **45**
3.1 Syntax . . . 46
3.2 Atomic Statement Semantics . . . 48
3.3 Denotational-Style Semantics . . . 51
3.4 Equation-Based Semantics . . . 55
3.5 Abstract Semantics . . . 58
3.6 Bibliographic Notes . . . 62

4 Non-Relational Abstract Domains **65**
4.1 Value and State Abstractions . . . 65
4.2 The Sign Domain . . . 70
4.3 The Constant Domain . . . 73
4.4 The Constant Set Domain . . . 75
4.5 The Interval Domain . . . 76
4.6 Advanced Abstract Tests . . . 80
4.7 Advanced Iteration Techniques . . . 84
4.8 The Congruence Domain . . . 93
4.9 The Cartesian Abstraction . . . 96
4.10 Summary . . . 97
4.11 Bibliographic Notes . . . 97

5 Relational Abstract Domains **99**
5.1 Motivation 99
5.2 The Affine Equalities Domain (Karr's Domain) 102
5.3 The Affine Inequalities Domain (Polyhedra Domain) 110
5.4 The Zone and Octagon Domains 125
5.5 The Template Domain 141
5.6 Summary 145
5.7 Bibliographic Notes 146

6 Domain Transformers **149**
6.1 The Lattice of Abstractions 149
6.2 Product Domains 152
6.3 Disjunctive Completions 161
6.4 Summary 176
6.5 Bibliographic Notes 177

7 Conclusion **179**
7.1 Summary 179
7.2 Principles 179
7.3 Towards the Analysis of Realistic Programs 182

Acknowledgements **183**

Abstract

Born in the late 70s, Abstract Interpretation has proven an effective method to construct static analyzers. It has led to successful program analysis tools routinely used in avionic, automotive, and space industries to help ensuring the correctness of mission-critical software.

This tutorial presents Abstract Interpretation and its use to create static analyzers that infer numeric invariants on programs. We first present the theoretical bases of Abstract Interpretation: how to assign a well-defined formal semantics to programs, construct computable approximations to derive effective analyzers, and ensure soundness, i.e., any property derived by the analyzer is true of all actual executions — although some properties may be missed due to approximations, a necessary compromise to keep the analysis automatic, sound, and terminating when inferring uncomputable properties. We describe the classic numeric abstractions readily available to an analysis designer: intervals, polyhedra, congruences, octagons, etc., as well as domain combiners: the reduced product and various disjunctive completions. This tutorial focuses not only on the semantic aspect, but also on the algorithmic one, providing a description of the data-structures and algorithms necessary to effectively implement all our abstractions. We will encounter many trade-offs between cost on the one hand, and precision and expressiveness on the other hand. Invariant inference is formalized on an idealized, toy-language, manipulating perfect numbers, but the principles and algorithms we present are effectively used in analyzers for real industrial programs, although this is out of the scope of this tutorial.

This tutorial is intended as an entry course in Abstract Interpretation, after which the reader should be ready to read the research literature on current advances in Abstract Interpretation and on the design of static analyzers for real languages.

Chapter 1

Introduction

While software are naturally meant to be run on computers, they can also be studied, manipulated, analyzed, either by hand or mechanically, that is, by other computer programs. A common example is compilation, which transforms programs in source code form into programs in binary code suitable for direct interpretation by a processor — or by a virtual machine, yet another program. This tutorial concerns *static analysis*, a less common example of computer programs manipulating other programs. A static analyzer is a program that takes as input a program and outputs information about its possible behaviors, without actually executing it.

In a broad sense, static analysis also covers syntactic analyses, that search for predefined patterns, as well as code quality metrics, such as counting the number of comments. However, we focus here on *semantic-based* static analyses. These methods output program properties that are *provably correct* with respect to a clear mathematical formalization of program behaviors. Such a high level of confidence in the analysis results is necessary in many applications, ranging from compiler optimization to program verification. One example property is that two pointers never alias. If proved true, the property can be exploited by a compiler to enable optimizations that would be incorrect in the presence of aliasing. Another example is finding bounds on array index expressions. This can be exploited in program verification to ensure that a program is free from out-of-bound array accesses. For the correctness proof to be valid, it is necessary to ensure that the inferred bounds indeed encompass all possible index values computed in all possible executions of the program.

Formal methods. The idea of reasoning with mathematical rigor about programs dates back from the early days of computers [Turing, 1949] and lead to the rich field of *formal methods* with the pioneering work of Hoare [1969] and Floyd [1967] on program logic. The lack of automation for writing and checking program proofs hindered these early efforts. In fact, Turing famously proved the undecidability of the halting problem, and Rice [1953] generalized this result, stating that all non-trivial properties about programs are undecidable. Hence, program verification cannot be fully automated. This fundamental

limitation can be sidestep in different ways, leading to the various flavors of program verification methods used today. Cousot and Cousot [2010] classify current formal methods into three categories, depending on whether automation, generality, or completeness is abandoned:

- *Deductive Methods*, which inherit directly from the work of Hoare [1969] and Floyd [1967], use interactive logic-based tools, including proof assistants such as Coq [Bertot and Castéran, 2004] and theorem provers such as PVS [Owre et al., 1992]. These tools are largely mechanized, but rely ultimately on the user, to a varying degree, to guide the proof.

- *Model Checking*, pioneered by Clarke et al. [1986], restricts program verification problems to decidable fragments. Initially restricted to finite models, it has since been generalized to infinite-state but regular models by McMillan [1993] in symbolic model checking. In practice, this often means that a model must be extracted, by hand, before the analysis can be performed. Alternatively, software bounded model checkers, such as CBMC [Clarke et al., 2004], analyze programs in actual programming languages such as C, but consider only a finite part of their executions.

- *Static Analysis*, studied in this tutorial, performs a direct analysis of the original source code, considering all possible executions and without user intervention, but resorts to approximations and analyzes the program at some level of abstraction that forgets about details that are, hopefully, irrelevant for the kind of properties checked. The abstraction is incomplete and can miss some properties, resulting in false alarms, i.e., the program is correct but the analyzer cannot prove it.

Abstract Interpretation. The theory of *Abstract Interpretation*, introduced by Cousot and Cousot [1977], is a general theory of the approximation of formal program semantics. It is an invaluable tool to prove the correctness of a static analysis, as it makes it possible to express mathematically the link between the output of a practical, approximate analysis, and the original, uncomputable program semantics. Both are seen as the same object, at different levels of abstraction. Additionally, Abstract Interpretation makes it possible to derive, from the original program semantics and a choice of abstraction, a static analysis that is correct by construction. Finally, the notion of abstraction is a first class citizen in Abstract Interpretation: abstractions can be manipulated and combined, leading to modular designs for static analyses. In this tutorial, we will design static analyses by Abstract Interpretation.

The rest of this chapter presents informally static analyses by Abstract Interpretation in order to derive simple numeric properties on the variables of a program.

1.1 A First Static Analysis: Informal Presentation

Let us consider, as first example, the program in Fig. 1.1. The `mod` function takes two arguments, A and B, then computes in Q and R, respectively, the integer dividend A/B

```
    //@ requires A >= 0 && B >= 0;
    int mod(int A, int B) {
1:    int Q = 0;
2:    int R = A;
3:    while (R >= B) {
4:      R = R - B;
5:      Q = Q + 1;
6:    }
7:    return R;
    }
```

Figure 1.1: A simple C function returning the modulo R of its arguments, with some precondition on the arguments A and B.

and the remainder $A\%B$, and finally returns R. This very naive function is written in a C-like language, and enriched with a `//@requires` annotation, written in the ACSL specification language [Cuoq et al., 2012], stating that it is always called with positive values for A and B.

The most straightforward way to model the function behavior is to consider execution traces: we execute the function step by step (where each step is a simple assignment or test) and record, at each step, the current program location and the value of each variable in scope. In our example, a program state would have the form $\langle l : a, b, q, r\rangle$ where l is the line number from Fig. 1.1 and a, b, q, r are, respectively, the values of variables A, B, Q, R. The execution starting with $A = 10$ and $B = 3$ would give the following trace (where variables not yet in scope are not shown in the state):

$$\begin{aligned}
&\langle 1 : 10, 3\rangle \to \langle 2 : 10, 3, 0\rangle \to \langle 3 : 10, 3, 0, 10\rangle \\
&\to \langle 4 : 10, 3, 0, 10\rangle \to \langle 5 : 10, 3, 0, 7\rangle \to \langle 6 : 10, 3, 1, 7\rangle \\
&\to \langle 4 : 10, 3, 1, 7\rangle \to \langle 5 : 10, 3, 1, 4\rangle \to \langle 6 : 10, 3, 2, 4\rangle \\
&\to \langle 4 : 10, 3, 2, 4\rangle \to \langle 5 : 10, 3, 2, 1\rangle \to \langle 6 : 10, 3, 3, 1\rangle \to \langle 7 : 10, 3, 3, 1\rangle
\end{aligned}$$

i.e., the function returns 1, which is indeed the remainder of 10 by 3.

There are many such executions, one for each initial value of A and B, but we can see intuitively that, in each of them, R and Q remain positive. This information can be useful to a compiler (which can then use unsigned types and arithmetic instead of signed ones) or to a program verifier (e.g., if the result of the function is used in an unsigned context).

1.1.1 Sign Analysis

Our first static analysis attempts to establish rigorously the sign of the variables. A naive method, which ensures that all possible program behaviors are considered, is to effectively simulate every possible execution by running the program, and then collect the signs of variable values along these executions. Naturally, this is not very efficient, and we will construct a far more efficient method.

```
    //@requires A >= 0 && B >= 0;
    int mod(int A, int B) {
1:      ⦇ A = (≥0), B = (≥0) ⦈
        int Q = 0;
2:      ⦇ A = (≥0), B = (≥0), Q = 0 ⦈
        int R = A;
3:      ⦇ A = (≥0), B = (≥0), Q = 0, R = 0 ⦈
        while (R >= B) {
4:          ⦇ A = (≥0), B = (≥0), Q = (≥0), R = (≥0) ⦈
            R = R - B;
5:          ⦇ A = (≥0), B = (≥0), Q = (≥0), R = ⊤ ⦈
            Q = Q + 1;
6:          ⦇ A = (≥0), B = (≥0), Q = (≥0), R = ⊤ ⦈
        }
7:      ⦇ A = (≥0), B = (≥0), Q = (≥0), R = ⊤ ⦈
        return R;
    }
```

Figure 1.2: Modulo function from Fig. 1.1 annotated with the result of a sign analysis in comments.

A key principle of Abstract Interpretation is replacing these actual, so-called *concrete*, executions, with *abstract* ones. For a sign analysis, we replace the concrete states mapping each variable to an integer value with an abstract state mapping each variable to a sign. Program instructions can then be interpreted in the world of signs by employing well-known rules of signs, such as $(\geq 0) + (\geq 0) = (\geq 0)$, i.e., positive plus positive equals positive, etc. Starting from positive values of A and B, one possible execution is:

$$\begin{aligned}
&\langle 1 : (\geq 0), (\geq 0)\rangle \rightarrow \langle 2 : (\geq 0), (\geq 0), 0\rangle \rightarrow \langle 3 : (\geq 0), (\geq 0), 0, (\geq 0)\rangle \\
&\rightarrow \langle 4 : (\geq 0), (\geq 0), 0, (\geq 0)\rangle \rightarrow \langle 5 : (\geq 0), (\geq 0), 0, \top\rangle \\
&\rightarrow \langle 6 : (\geq 0), (\geq 0), (\geq 0), \top\rangle \rightarrow \langle 4 : (\geq 0), (\geq 0), (\geq 0), \top\rangle \\
&\rightarrow \langle 5 : (\geq 0), (\geq 0), (\geq 0), \top\rangle \rightarrow \langle 6 : (\geq 0), (\geq 0), (\geq 0), \top\rangle \\
&\rightarrow \langle 7 : (\geq 0), (\geq 0), (\geq 0), \top\rangle
\end{aligned}$$

where $\top$ indicates that the sign is unknown — the variable may be positive or negative. Note that the value of Q, which is 0 when first going through location 4, becomes (≥ 0) at the second passage, which is expected as Q increases. Collecting the sign of the variables at each program point, we can annotate the program from Fig. 1.1 with sign information; the result is shown in Fig. 1.2. These annotations are *invariants*: the values of the variables in every concrete execution passing through a given control location satisfy the sign property we provided at this location.

Note that, at the end of the function, we have no information on R ($R = \top$) while, in fact, R is always positive. We can trace the introduction of an uncertainty, $\top$, to the computation, at line 4, of `R - B` which, in the sign domain, gives $(\geq 0) - (\geq 0) = \top$. Indeed, `R - B` can only be proven to be positive if we know that $R \geq B$, which is not a sign information. So, while $R = (\geq 0)$ is an invariant and a sign property, it cannot

be found by reasoning purely in the sign domain. Failure to infer the best invariants expressible in the abstract world is common in Abstract Interpretation and motivates the introduction of more expressive domains, as we will do shortly. The reader familiar with deductive methods will have guessed that this is related to the fact that $R = (\geq 0)$ is an invariant but not an *inductive invariant*. We will discuss this connection in depth later.

Note also that the test `R >= B` is interpreted in the abstract as $\top \geq (\geq 0)$, which is inconclusive. This means that, while we chose, in our abstract execution, to iterate the loop twice, longer executions with more loop iterations are also valid. The program, in the abstract, becomes non-deterministic. We argue, informally for now, that further iterations will not bring any new possible sign values: we have reached a fixpoint. Another key part of Abstract Interpretation is to know how to precisely iterate loops in the abstract, and when to stop, to guarantee that all possible program behaviors have been considered. It will be discussed at length in this tutorial.

1.1.2 Affine Inequalities Analysis

The sign analysis we presented is one of the simplest and least expressive static analysis there is. We illustrate the other end of the spectrum with a static analysis able to infer affine inequalities between variables. The invariants it computes on our modulo example are presented in Fig. 1.3. Its principle remains the same: we propagate an abstract representation of variable values through the program. However, it no longer has the simple form of a map from variables to abstract values, but is rather a conjunction of affine inequalities that delimit the set of possible concrete states the program can be in. As a consequence, the abstraction can represent relations, i.e., it is a *relational analysis*. Geometrically, we obtain a polyhedron.

Program instructions can still be applied on polyhedra. For instance, an assignment `Q = Q + 1` is modeled as a translation, while a test `R >= B` is modeled as adding an affine constraint. The exact algorithms will be described in details in Sect. 5.3. They borrow heavily from the classic mathematical theory of convex polyhedra. We can see, in Fig. 1.3, that the analysis is now able to exactly represent $R \geq B$, and can thus deduce that R remains positive, which was not possible in the sign analysis.

1.1.3 Iterations

To illustrate more clearly the need to iterate loops in the abstract, we consider the simple loop in Fig. 1.4.(a) that increments A and B from 0 to 100. The program is annotated with invariants computed in yet another abstraction, *intervals*, which infers variable bounds: a lower bound and an upper bound. This popular abstraction will be discussed at length in Sect. 4.5. We can easily compute the abstract effect of instructions using interval arithmetic. For instance, `A = A + 1` adds 1 to both the lower and the upper bounds of A.

Program location 2 in Fig. 1.4.(a) is the location reached just before testing the condition `A < 100` a first time to determine whether to enter the loop at all, and reached again after each loop iteration before testing the condition to determine whether to reenter the

```
    //@requires A >= 0 && B >= 0;
    int mod(int A, int B) {
1:      ⟅ A ≥ 0, B ≥ 0 ⟆
        int Q = 0;
2:      ⟅ A ≥ 0, B ≥ 0, Q = 0 ⟆
        int R = A;
3:      ⟅ A ≥ 0, B ≥ 0, Q = 0, R = A ⟆
        while (R >= B) {
4:          ⟅ A ≥ 0, B ≥ 0, Q ≥ 0, R ≥ B ⟆
            R = R - B;
5:          ⟅ A ≥ 0, B ≥ 0, Q ≥ 0, R ≥ 0 ⟆
            Q = Q + 1;
6:          ⟅ A ≥ 0, B ≥ 0, Q ≥ 1, R ≥ 0 ⟆
        }
7:      ⟅ A ≥ 0, B ≥ 0, Q ≥ 0, 0 ≤ R < B ⟆
        return R;
    }
```

Figure 1.3: Modulo function from Fig. 1.1 annotated with the result of an affine inequality analysis in comments. In red, we show the invariants that were not found by the sign analysis of Fig. 1.2.

```
    A = 0; B = 0;
1:  ⟅ A ∈ [0,0], B ∈ [0,0] ⟆
    while
2:      ⟅ A ∈ [0,100], B ∈ [0,+∞] ⟆
        (A < 100) {
3:      ⟅ A ∈ [0,99], B ∈ [0,+∞] ⟆
        A = A + 1;
4:      ⟅ A ∈ [1,100], B ∈ [0,+∞] ⟆
        B = B + 1;
5:      ⟅ A ∈ [1,100], B ∈ [1,+∞] ⟆
    }
6:  ⟅ A ∈ [100,100], B ∈ [0,+∞] ⟆
```

(a)

iteration	A	B
1	$[0,0]$	$[0,0]$
2	$[0,1]$	$[0,1]$
3	$[0,2]$	$[0,2]$
...	...	
100	$[0,99]$	$[0,99]$
101	$[0,100]$	$[0,100]$
102	$[0,100]$	$[0,101]$
103	$[0,100]$	$[0,102]$

(b)

Figure 1.4: Interval analysis of a simple loop (a) and the detailed iteration for location 2 (b).

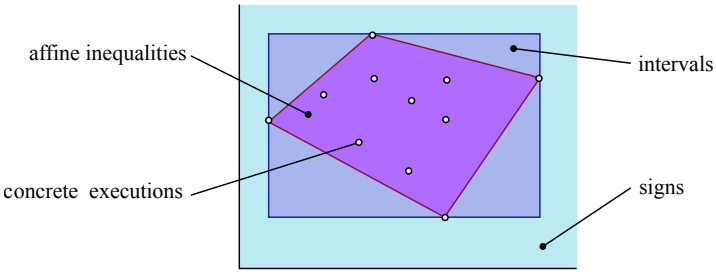


Figure 1.5: A set of points abstracted using affine inequalities (dark polyhedron), intervals (lighter rectangle) and signs (light quarter-plane).

loop body for a new iteration. The corresponding invariant is called a *loop invariant*, and provides a convenient summary of the behavior of the loop. A classic execution would have, for A at location 2, the sequence of values: 0, 1, 2, . . . , 100. The output of the analysis must, however, provide a single interval for location 2 that takes into account all the reachable values. Hence, the abstract semantics accumulates, at each iteration, every new value with that of preceding iterations. This flavor of semantics, useful for verification, is called a *collecting semantics*.

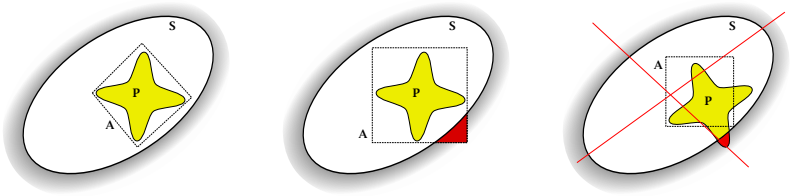
The iteration is shown in Fig. 1.4.(b). We observe that, for A, the iteration stabilizes at $[0, 100]$ after 101 iteration steps, allowing us to deduce that A equals 100 when the program ends, after the loop exit condition `A >= 100`. Such convergence is long. Another important contribution of Abstract Interpretation is a set of *convergence acceleration* methods, to construct more efficient analyses that use less iterations.

In some cases, the plain abstract iteration may not even converge. This is the case for variable B in Fig. 1.4 as there is no test on B to bound it. Convergence acceleration will ensure that, after a finite number of accelerated iterations, this behavior is detected and we output the stable interval $B \in [0, +\infty]$.

1.1.4 Precision

As seen on the modulo example from Figs. 1.2–1.3, the result of a static analysis depends on the abstract domain of interpretation, but it will always represent an over-approximation of the set of possible program states. More expressive abstractions generally lead to tighter over-approximations, and so, more precise results.

Figure 1.5 illustrates this by showing a set of planar points (representing, e.g., a set of concrete program states over two variables) and its best enclosing into a polyhedron (in the affine inequality domain), a box (in the interval domain), and a quarter-plane (in the sign domain). Polyhedra add less spurious states with respect to the concrete world, but, as we will see, polyhedra algorithms are also more complex and more costly, leading to a slower analysis. There is a tradeoff to reach between precision and cost.



precise analysis
$A \subseteq S \implies P \subseteq S$
(a)

false alarm
$A \not\subseteq S$ but $P \subseteq S$
(b)

unsound analysis
$A \subseteq S$ but $P \not\subseteq S$
(c)

Figure 1.6: Proving that a program P satisfies a safety specification S, i.e., that $P \subseteq S$, using an abstraction A of P: (a) succeeds, (b) fails with a false alarm, and (c) is not a possible configuration for a sound analysis.

1.1.5 Soundness

In the general sense, *soundness* states that whatever the properties inferred by an analysis, they can be trusted to hold on actual program executions. It is a very desirable property of formal methods, and one we will *always* ensure in this tutorial.

In our case, we expect the analysis to output invariants. It must thus contain at least all actual program states, but it may safely contain more. Computing over-approximations is thus our soundness guarantee. Considering over-approximations allows us to check rigorously so-called *safety* correctness specifications, that is, specifications stating that the set of reachable program states is included in a set of *safe* states — in practice, this set is either specified by the user, through explicit assertions, or specified implicitly by the language, such as the absence of arithmetic overflow. The need for over-approximations is intuitive: if the abstraction is included in the specification then, *a forciori*, the set of actual executions is included in the specification. This is illustrated in Fig. 1.6.(a).

If the abstract state computed does not satisfy the specification, however, the analysis is inconclusive. Either the program is actually flawed, or the program is correct but the abstraction over-approximates its behavior too coarsely for the analysis to prove it. This last case, called *false alarm*, is depicted in Fig. 1.6.(b). Handling this case requires either some investigation of the alarm, either manually or employing some other formal method, or running the analysis again with more precise abstractions. All the analyses we discus here are sound: the case where the program does not satisfy its safety specification while the analysis reports no specification violation, illustrated in Fig. 1.6.(c), will never occur.

1.2 Scope and Applications

This tutorial focuses on sound static analysis based on Abstract Interpretation in order to infer numeric invariants. For the sake of a pedagogical presentation, we analyze a simple toy-language missing many features from real-life languages, such as: functions, arrays, pointers, dynamic memory allocation, objects, exceptions, etc. We refer the reader to other

(a)

$\langle i \in [0,9] \rangle$
$\langle i \in [0,9] \rangle$
$\langle i+1 \in [-2^{31}, 2^{31}-1] \rangle$

```
int delay[10], i;
i = 0;
while (1) {
    int y = delay[i];
    delay[i] = input();
    i = i + 1;
    if (i >= 10) i = 0;
}
```

(b)

$\wr i = 0 \int$
$\wr i \in [0,9] \int$
$\wr i \in [0,9] \int$
$\wr i \in [0,9] \int$
$\wr i \in [1,10] \int$

```
int delay[10], i;
i = 0;
while (1) {
    int y = delay[i];
    delay[i] = input();
    i = i + 1;
    if (i >= 10) i = 0;
}
```

Figure 1.7: A C-like program manipulating an array annotated with: (a), correctness verification conditions implied by the language; and (b), invariants inferred by an interval static analysis.

publications and tool presentations, such as [Bertrane et al., 2015], to explain how to adapt the ideas presented here to the analysis of real-life languages and software. Nevertheless, in this section, we justify the interest of numeric invariants by showing analysis applications that are based on, or parameterized with, numeric abstractions. As programs manipulate, at their core, numbers, it is natural to think about numeric abstractions as a key component in most value-sensitive program analyses.

1.2.1 Safety Verification

Figure 1.7.(a) gives an example program together with the verification conditions it must satisfy at various program locations in order to be free from arithmetic overflows and out-of-bound array accesses. These conditions can be derived easily and purely mechanically from the syntax of the program, and they have a purely numeric form.

Figure 1.7.(b) shows the invariants inferred at these points by a static analysis based on intervals. The invariants clearly imply the verification conditions. Hence, the program is free from the errors we target. As we have employed an interval analysis, and the verification conditions can be expressed exactly as intervals, checking the conditions can be done without leaving the abstract world of intervals.

1.2.2 Pointer Analysis

Numeric invariants are not only useful to analyze numeric variables, but also any variable with a numeric aspect. Consider the program in Fig. 1.8.(a) employing pointer arithmetic

```
float* p = q;                 unsigned off_p = off_q;
for (i = 0; i < 10; i++)      for (i = 0; i < 10; i++)
    if (...) p++;                 if (...) off_p += 4;
                                  ⦃ off_q ≤ off_p ≤ off_q + 4 i + 4 ⦄
        (a)                                  (b)
```

Figure 1.8: A C-like program manipulating a pointer p (a) and its translation into a numeric program manipulating its offset off_p (b). Program (b) also shows the numeric invariants inferred on off_p.

on a pointer p to traverse data in a loop. We can view a pointer value as a pair composed of a variable, and a numeric offset counting a number of bytes from the first byte of the variable — offset 0. Pointer arithmetic will only operate on the offset part, and in a way similar to integer arithmetic. We can transform this program into a purely numeric program operating on synthetic offset variables, such as off_p, instead of pointers, as shown in Fig. 1.8.(b). We can then apply a standard numeric static analysis to infer numeric invariants on offsets. On the example of Fig. 1.8.(b), an affine inequalities analysis would find a relation between the pointer offset and the loop counter i.

Some information about pointer alignment, namely the fact that the offset is a multiple of 4, is missing, because it cannot be represented using affine inequalities. We will see, in Sect. 4.8, a *congruence abstraction* that solves this issue. In fact, each inference problem, for each required property can be solved by designing some adapted abstraction. Finally, note that, in practice, a numeric analysis is combined with a non-numeric points-to analysis [Balakrishnan and Reps, 2004, Miné, 2006b] that infers the first component of pointer values, i.e., the identity of the variables the pointers may point into.

Another, related class of analyses is that of C strings, for instance the analyses by Dor et al. [2001] or by Simon and King [2002]. In this case, a string buffer and a pointer into such a buffer are translated into purely numeric synthetic variables. In addition to offset variables, we need to insert instrumentation variables tracking the position of the first occurrence of the null character (i.e., the string length) and the number of bytes available until the end of the buffer. We also need to modify the program to update them. Using a relational analysis, such as affine inequalities, allows inferring non-trivial relationships, such as a relation between the lengths of the strings used as arguments and return in a string concatenation function such as `strcat`.

1.2.3 Shape Analysis

Beyond pointer analyses, *shape analyses* are a sophisticated family of analyses targeting programs with dynamic memory allocation and recursive data-structures, such as lists or trees. Such analyses also benefit from instrumenting numeric quantities to discuss about, for instance, list length or tree height. Additionally, a *non-uniform* analysis, as proposed by Venet [2004], is able to express properties that distinguish between different instances of a recursive data-structure. Figure 1.9 presents an application to the allocation of a

```
cell *x, *head = NULL;
for (i = 0; i < n; i++) {
    x = alloc();
    x->next = head; head = x;
}
for (i = 0, x = head; x; x = x->next, i++) {
    ≀ ∀k ∈ [0, i − 1] : a[k] = head(->next)^k ->data ∫
    a[i] = x->data;
}
≀ ∀k ∈ [0, n − 1] : a[k] = head(->next)^k ->data ∫
```

Figure 1.9: A C-like program manipulating a linked list and an array, annotated with non-uniform invariants stating a relation between the contents of the array at position k and the list at the same position k.

```
cost = 0;
for (i = 0; i < n-1; i++) {
    ≀ cost = i × n − i × (i + 1)/2 ∫
for (j = i+1; j < n; j++) {
        ≀ cost = i × (n − i) × (i + 1)/2 + j − i − 1 ∫
        if (tab[i] > tab[j]) swap(tab[i],tab[j]);
        cost = cost+1;
    }
}
≀ cost = (n + 1) × (n − 2)/2 ∫
```

Figure 1.10: A sorting algorithm, with an instrumentation variable, *cost*, added to help compute the time complexity.

linked list followed by a copy from an array into the list. The loop invariant states that, at loop step i, the $k-$th element of the linked list, pointed to by $head(\texttt{->}next)^k\texttt{->}data$, equals $a[k]$. This very symbolic logic predicate is complemented by the numeric invariant $0 \leq k \leq i-1$, which restricts the predicate to elements at indices up to i. This numeric invariant can be inferred using the numeric abstractions presented in this tutorial.

1.2.4 Cost Analysis

Numeric invariants do not necessarily refer to quantitative information on the memory state, but can also refer to quantitative information about execution traces, such as their length. This provides some information about the time complexity of the program. One prime example is the Costa analyzer, introduced by Albert et al. [2007].

Figure 1.10 shows a very simple method for obtaining such a bound: the program is instrumented with a synthetic variable, named *cost*, which is incremented at each step. A numeric invariant analysis can then be used to infer properties on *cost*, including an upper bound which is symbolic in the arguments of the function, thanks to a relational

(a)	(b)	(c)
`x = input(`$[-10, 10]$`)`	$\wr\, x \in [-10, 10] \,\wr$	$\wr\, \bot \,\wr$
`if (x == 0) z = 0;`		
`else {`	$\wr\, x \in [-10, 10] \,\wr$	$\wr\, x = 0 \,\wr$
`  y = x;`	$\wr\, x \in [-10, 10],\ y \in [-10, 10] \,\wr$	$\wr\, y = 0 \,\wr$
`  if (y < 0) y = -y;`	$\wr\, x \in [-10, 10],\ y \in [0, 10]] \,\wr$	$\wr\, y = 0 \,\wr$
`  z = x / y;`	$\wr$ division by zero $\wr$	
`}`		

Figure 1.11: A program (a); the result of a forward analysis (b); and the result of a backward analysis assuming a division by zero (c).

analysis. Note that the invariants here are far more complex than those we encountered before as they are not affine, but polynomial. In this tutorial, we will limit ourselves to affine invariants, which are generally not sufficient for cost analyses, but are much simpler and can be inferred more efficiently.

Another, related application is proving termination. Classic termination proofs require finding a decreasing ranking function that is bounded below, and numeric properties can help with that [Urban and Miné, 2014].

1.2.5 Backward Analysis

We return to purely numeric properties and intervals to show another flavor of analysis, which goes backward. Instead of inferring the value of variables by propagating forward an abstract memory state from the beginning of the program, an analysis can start from a program point of interest and an abstract property on the memory state, and go backward to derive necessary conditions so that the executions reach the given program point satisfying the given abstract state property. In fact, backward analysis is most often used in combination with a preliminary forward analysis, to refine and focus its results. This scheme is developed for instance by Bourdoncle [1993a].

Figure 1.11.(a) shows a simple C program that divides x by its absolute value $y = |x|$. As the division is guarded by the test `x == 0`, there is no division by zero. Figure 1.11.(b) annotates the program with the result of an interval analysis, starting from $x \in [-10, 10]$. As the interval domain cannot represent $[-10, 10] \setminus \{0\}$, it cannot exploit the fact that $x \neq 0$, and so, $y \neq 0$, when the division `x / y` occurs. The analysis outputs an alarm, which is actually a false alarm. To help the user reason about this alarm, a backward analysis is performed starting just before the error, at the division, with the erroneous state $y = 0$. This state is propagated backward, in the interval domain. We deduce, in particular, that $x = 0$ must hold just after the test `x == 0` has returned false. Propagating backward one more step, the analysis infers that there is no possible program state, denoted here as $\bot$. In our case, the backward analysis has proved automatically that the error is spurious. In more complex cases, the analysis would simply find a restriction of the state space that would help the user, or another formal method, decide whether the alarm is false or

justified.

In the rest of the tutorial, all our examples concern forward analyses to infer invariants. Nevertheless, backward analyses are very similar, and require only a few additional operators.

1.3 Outline

This chapter provided an informal introduction to numeric invariant inference and its applications. The rest of the tutorial will present inference methods in a rigorous way, based on the theory of Abstract Interpretation.

Chapter 2 presents the mathematical tools that will be needed in our formal presentation, including a short course on Abstract Interpretation. Chapter 3 presents our target programming language: a toy language tailored to illustrate numeric invariants. It presents not only the language syntax, but also its concrete semantics in a mathematical, unambiguous way. It then presents how abstractions can be applied to derive an effective static analysis that is sound with respect to the concrete world: we state the operators and hypotheses required on the abstraction, and then develop an analysis that is fully parametric in the choice of the abstraction. Chapters 4 and 5 present two families of such abstractions: firstly, non-relational domains, including signs, constants, intervals, and congruences; secondly, relational domains, including affine equalities, affine inequalities, and weakly relational domains (zones, octagons, and templates). Chapter 6 discusses abstract domain combiners that improve the precision of existing domains: firstly, the reduced product, a technique to combine two or more existing abstractions and design a more expressive analyzer in a modular way; secondly, three methods that improve the precision of a given abstraction by allowing it to express symbolic disjunctions (powerset completion, state partitioning, and path partitioning). To close this tutorial, Chap. 7 provides concluding remarks.

Naturally, we devote a large amount of time presenting the data-structures and algorithms necessary to implement effectively these abstractions in a static analyzer, and we discuss their relative merits in terms of precision, cost, and expressiveness. Each chapter ends with bibliographical notes recalling major articles the reader is invited to consult to complete this necessarily superficial survey.

1.4 Further Resources

To end our introduction, we list additional resources available on-line that can be used as a complement to this tutorial.

For an informal introduction to Abstract Interpretation and links to selected technical resources — including articles, slides, and video presentations — we refer the reader to Patrick Cousot's web-page.[1]

[1] `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`

This tutorial is based on several Master-level courses, at École Normale Supérieure, Paris 6, and Paris 7 Universities in France.[2] A programming project focusing on the development, in OCaml, of a simple static analyzer for numeric properties on a toy-language, not unlike the language studied here, is also available.[3] We also refer the reader to Master-level courses by Patrick Cousot at MIT[4] and at Marktoberdorf Summer School.[5]

Implementations of numeric static analyses are also available. The Interproc analyzer[6] is a simple, open-source numeric analyzer on a toy-language, for educational and scientific demonstration purposes. It demonstrates the use of some of the abstract domains we will present in this tutorial: intervals, linear equalities, linear inequalities, and octagons. It additionally features backward and modular inter-procedural analyses, which we will not present formally here. Its most notable feature is that it can be used on-line, through a web interface. The Apron library[7] [Jeannet and Miné, 2009], on which Interproc is based, is an open-source library implementing classic numeric domains; it can be used in static analysis projects. Industrial-strength commercial static analyzers include the Astrée analyzer for C [Bertrane et al., 2010], which was used to analyze the run-time errors in avionics software. Evaluation versions are freely available from AbsInt.[8] Julia[9] is a commercial static analyzer for Java. Frama-C[10] [Cuoq et al., 2012] is an open-source program analyzer for C incorporating Abstract Interpretation.

[2] Course slides in English are available at: `https://www-apr.lip6.fr/~mine/enseignement/mpri/2016-2017/`

[3] English version available at: `https://www-apr.lip6.fr/~mine/enseignement/l3/2015-2016/project`

[4] `http://web.mit.edu/16.399/www/`

[5] `http://www.di.ens.fr/~cousot/Marktoberdorf98.shtml`

[6] `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`

[7] `http://apron.cri.ensmp.fr/library/`

[8] `http://www.absint.com/astree`

[9] `https://www.juliasoft.com/`

[10] `https://frama-c.com/`

Chapter 2

Elements of Abstract Interpretation

The Abstract Interpretation theory helps tremendously in the design of static analyzers: it ensures their soundness by construction, guides design choices, and can even ensure optimality at some level. In this chapter, we review the mathematical foundations of Abstract Interpretation, and we introduce notations, definitions, and key theorems that will be used in the rest of the tutorial. The theorems are well-known, and stated here without proofs, as we are more interested in providing an intuitive understanding — we provide references to the proofs in textbooks and articles for the interested reader.

We assume a basic knowledge of first-order logic, set theory, and linear algebra. However, we review order theory, which is less known.

Basic notations. We use standard notations from first-order logic: disjunction $\vee$, conjunction $\wedge$, logical negation $\neg$, implication $\Longrightarrow$, equivalence $\Longleftrightarrow$, as well as universal $\forall$ and existential $\exists$ quantifiers. To distinguish a definition from an equality or an equivalence, we use $\stackrel{\text{def}}{=}$ and $\stackrel{\text{def}}{\Longleftrightarrow}$ for the former, and $=$ and $\Longleftrightarrow$ for the later.

We also use standard notations from set theory: the empty set $\emptyset$, set union $\cup$ and intersection $\cap$, in binary form ($A \cup B$ and $A \cap B$) or over a family ($\bigcup_{i \in I} A_i$ and $\bigcap_{i \in I} A_i$ for a family $(A_i)_{i \in I}$ of sets indexed by I, which may be finite or infinite), Cartesian product $\times$, set inclusion $\subseteq$, set ownership $\in$, set difference $\setminus$. Set comprehension, in particular, will be used pervasively: $\{\, x \in A \mid P(x) \,\}$ is the subset of elements from A that additionally satisfy some logic predicate P. Given a set A, we denote as $\mathcal{P}(A)$ the set of its parts, also called its *powerset*, i.e., the set of all the sets included in A. We denote as $\mathcal{P}_{\mathit{finite}}(A)$ the set of *finite* subsets of A. We denote as $|A|$ the number of elements in A. Given two sets A and B, we denote as $A \to B$ the set of functions from A to B; A is called the codomain of such a function, and B is its domain. We will sometimes use the lambda notation $\lambda x.\, f(x)$ to denote functions concisely. Alternatively, a function can be described in extension as $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$, meaning that its codomain is $\{x_1, \ldots, x_n\}$ and it maps any x_i to the corresponding v_i. Given a function f, $f[x \mapsto v]$ denotes the function that maps x to v

and any y such that $x \neq y$ to $f(y)$, i.e., it updates the function at point x to equal value v. Finally, $\circ$ denotes function composition, f^i denotes f composed i times, and id denotes the identity function ($f^0 \stackrel{\text{def}}{=} id$ and $f^{i+1} \stackrel{\text{def}}{=} f^i \circ f = f \circ f^i$).

We denote respectively as $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, the sets of natural integers, integers, rationals, and reals. $\mathbb{N}^*$, $\mathbb{Z}^*$, $\mathbb{Q}^*$, and $\mathbb{R}^*$ are the set of non-zero natural integers, integers, rationals, and reals. Finally, $\mathbb{R}^+$ and $\mathbb{Q}^+$ represent positive (possibly null) reals and rationals.

As we work with numeric programs only, memory states map variables to values in some numeric set, such as integers or reals. Hence, memory states can be assimilated to vectors in a vector space. Vectors are denoted as $\vec{v}$, matrices as $\mathbf{M}$, matrix-matrix and matrix-vector multiplications as $\times$, and the dot product of vectors is denoted as $\cdot$ (a dot). The $i-$th component of a vector $\vec{v}$ is denoted as v_i. Moreover, the $i-$th line or $i-$th column of a matrix $\mathbf{M}$, depending on the context, is a vector denoted as $\vec{M}_i$. The element at line i and column j of a matrix $\mathbf{M}$ is denoted as M_{ij}. We order vectors element-wise: $\vec{v} \geq \vec{w}$ means $\forall i\colon v_i \geq w_i$. The zero vector is denoted as $\vec{0}$.

Additional mathematical tools. Each abstract domain employs data-structures and algorithms specific to the shape of invariants it represents and manipulates. Abstract Interpretation thus leverages existing mathematical theories, and adapt them to discuss about program semantics. For instance, we will see that interval abstractions (Sect. 4.5) employ algorithms from interval arithmetic and constraint programming; affine inequalities abstractions (Sect. 5.3) leverage tools from the theory of convex polyhedra; congruence abstractions (Sect. 4.8) are based on basic number theory, etc. We do not assume a prior knowledge of these various fields. We will present relevant results and algorithm only when needed, omitting the proof of well-known theorems as well as irrelevant parts of these theories when not needed. Our view is that of a client, intent on using them as tools in the design of abstract domains, and possibly adapting them to our needs.

2.1 Order Theory

2.1.1 Partial Orders

The main structure we require on mathematical objects describing concrete and abstract computations is a partial order:

Definition 2.1 (Partial order, Poset)**.**
A partial order $\sqsubseteq$ *on a set* X *is a relation* $\sqsubseteq \in X \times X$ *that is:*

1. *reflexive:* $\forall x \in X\colon x \sqsubseteq x$;
2. *anti-symmetric:* $\forall x, y \in X\colon (x \sqsubseteq y) \wedge (y \sqsubseteq x) \implies x = y$;
3. *and transitive:* $\forall x, y, z \in X\colon (x \sqsubseteq y) \wedge (y \sqsubseteq z) \implies x \sqsubseteq z$.

We denote as $(X, \sqsubseteq)$ *the set* X *equipped with the partial order* $\sqsubseteq$*, and call this pair a* partially ordered set*, or a* poset*.* ■

Compared to the usual, so-called total orders, such as number comparison $\leq$, a partial order is not total: there can exist unordered pairs of elements — x, y such that neither $x \sqsubseteq y$ nor $y \sqsubseteq x$ holds.

Example 2.1 (Poset examples)**.**

1. *Any completely ordered set is also a poset; for instance:* $(\mathbb{Z}, \leq)$.

2. *The parts of any set X ordered by inclusion,* $(\mathcal{P}(X), \subseteq)$*, is a poset. The order is not complete; consider for instance that neither* $\{1\} \subseteq \{2\}$ *nor* $\{2\} \subseteq \{1\}$ *holds.*

3. $(\mathbb{Z}^2, \sqsubseteq)$*, the set of pairs of integers ordered as:* $(a, b) \sqsubseteq (a', b') \iff a \geq a' \wedge b \leq b'$ *is a poset. When interpreting the first element of each pair as the lower bound of an interval and the second element as its upper bound, this order is equivalent to interval inclusion.* ♦

Partial orders are extremely important in several areas of Theoretical Computer Science. For instance, they play a crucial role in the field of denotational semantics, introduced by Scott and Strachey [1971], to ensure that the definitions assigning a mathematical meaning to programs are well-founded. In Abstract Interpretation, we will use partial orders to model no less than four different concepts:

1. Partial orders convey the idea of *approximation* (Fig. 1.5). Some analysis results may be coarser than some other results. The order is indeed partial as, sometimes, analysis results are incomparable. Consider, for instance, a variable that actually ranges in $[1, 9]$ and two sound interval analyses that output respectively $[0, 9]$ and $[1, 10]$.

2. Partial orders convey the idea of *validating a specification* (Fig. 1.6). For instance, we can see a program P satisfying a specification S as the set inclusion $P \subseteq S$, meaning that all program behaviors are included in the authorized set of behaviors.

3. Partial orders convey the idea of *soundness* (Fig. 1.6). An analysis is sound when it provably outputs a result that is coarser than the actual behavior.

4. Partial orders convey the idea of *iteration* (Fig. 1.4.(b)). When analyzing loops, a sequence of increasing semantic elements is computed, and the fact that they are ordered will be important to ensure that the computation is advancing towards a well-defined loop invariant.

2.1.2 Lower and Upper Bounds

Given two elements a and b in a poset X, we call *upper bound* of a and b any element $c \in X$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$, i.e., c is greater than both a and b. Likewise, a *lower bound* c of a and b satisfies $c \sqsubseteq a$ and $c \sqsubseteq b$. Moreover, c is the *least upper bound*, also called *lub* or *join*, if it is the smallest element greater than both a and b. Such a least

upper bound does not necessarily exist but, when it does, it is unique. It is written as $a \sqcup b$. Likewise, the unique *greatest lower bound* of a and b, also called *glb* or *meet*, if it exists, is the greatest element smaller than a and b, and it is denoted as $a \sqcap b$.

As $\sqcup$ and $\sqcap$ are associative operators, we will employ also the notations $\sqcup A$ and $\sqcap A$ to compute lubs an glbs on arbitrary (possibly infinite) sets A of elements. Finally, we denote respectively as $\bot$ (called *bottom*) and $\top$ (called *top*) the *least element* and the *greatest element* in the poset, if they exist.

Example 2.2. *In the powerset poset $(\mathcal{P}(X), \subseteq)$ of any (possibly infinite) set X, the lub $\sqcup$ is simply the set union $\cup$, and the glb $\sqcap$ is simply the set intersection $\cap$. Both always exist for all arguments — hence, we actually have a complete lattice structure, as described in Sect. 2.1.5. We also have $\bot = \emptyset$ and $\top = X$.* ♦

Example 2.3. *Not all posets have upper bounds or lubs. Consider, for instance, the poset $(\{a, b\}, =)$, where the partial order is the equality. Then, there is no upper bound for a and b, and so, there is no lub. Likewise, we can construct posets where a pair of elements have several upper bounds but no least upper bound, and posets where finite sets of elements have lubs but infinite sets of elements do not.* ♦

Remark 2.1. *We will also use the notation* $\min A$ *and* $\max A$ *to denote, respectively, the minimum and the maximum element in a set $A \subseteq X$. Note the difference between* $\min A$ *(resp.* $\max A$*) and $\sqcap A$ (resp. $\sqcup A$): the former imposes that the element smaller (resp. larger) than all the elements in A is in A, while the later may denote an element in X outside A. For instance, in the classic order $(\mathbb{R}, \leq)$, $0 \sqcup 1 = \max\{0, 1\} = 1 \in \{0, 1\}$, while $\sqcup\{\, x \in \mathbb{R} \mid x < 1 \,\} = 1 \notin \{\, x \in \mathbb{R} \mid x < 1 \,\}$. Naturally,* $\min A$ *and* $\max A$ *are not always defined for every set A in every poset, even less so than $\sqcap A$ and $\sqcup A$.* ◊

2.1.3 Hasse Diagrams

A poset is often presented graphically by placing greater elements higher, and materializing the order with lines. Such a visual representation is called a *Hasse diagram*. It also makes lubs and glbs easy to see.

Example 2.4 (Hasse diagrams)**.** *Figure 2.1.(a) provides the Hasse diagram for the simple poset where $a \sqsubseteq b$, $b \sqsubseteq c$, $b \sqsubseteq d$, $c \sqsubseteq e$, $c \sqsubseteq f$, $d \sqsubseteq f$, $e \sqsubseteq g$, and $f \sqsubseteq g$. Note that we have omitted lines between elements that are obviously ordered through reflexivity or transitivity (e.g., $c \sqsubseteq g$ but there is no line from c to g as $c \sqsubseteq e \sqsubseteq g$). Also note that, although e and f are not ordered, they do have a lub, g.*

Figures 2.1.(b)–(c) present Hasse diagrams for total orders: the natural integers ordered as usual by $\leq$, and the integers enriched with a infinity element ∞ greater than all integers: $\forall x \in \mathbb{N} \cup \{\infty\} \colon x \sqsubseteq \infty$.

Figure 2.2 gives the Hasse diagram for the powerset of integers $(\mathcal{P}(\mathbb{Z}), \subseteq)$. ♦

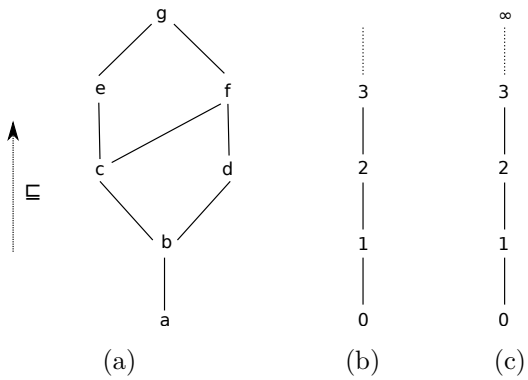


Figure 2.1: Hasse diagrams for: (a) a finite poset, (b) the natural integers $(\mathbb{N}, \leq)$, and (c) the natural integers enriched with infinity $(\mathbb{N} \cup \{\infty\}, \leq)$.

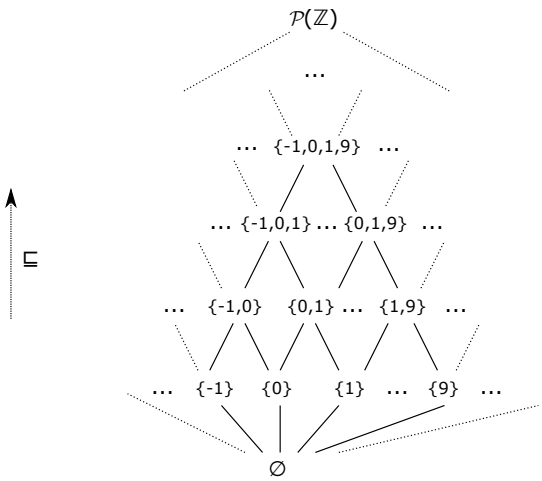


Figure 2.2: Hasse diagram for the powerset poset of integers $(\mathcal{P}(\mathbb{Z}), \subseteq)$.

2.1.4 Chains and CPO

The informal example of Fig. 1.4.(b) from the last chapter features, during loop iteration, increasing sequences of intervals: $[0,0]$, $[0,1]$, $[0,2]$,... To get an invariant, it is necessary to find a limit to such sequences. For variable A, the iteration stops at $[0,100]$, which is naturally the limit. For variable B, the iteration keeps increasing, so that a natural limit is actually $[0,+\infty]$. In both cases, our notion of limit coincides with that of lub. Note, however, that we do not require lubs for arbitrary families of elements, but only for those that come from iteration sequences. This motivates the following definitions:

Definition 2.2 (Chain)**.** *A* chain $C \subseteq X$ *in a poset* $(X, \sqsubseteq)$ *is a subset* C *of* X *that is totally ordered:* $\forall c, d \in C\colon (c \sqsubseteq d) \vee (d \sqsubseteq c)$. ■

Definition 2.3 (CPO)**.** *A* complete partial order*, or* CPO*, is a poset such that every chain has a lub.* ■

Note that $\emptyset$ is a chain, so, our definition requires that $\sqcup\emptyset$ exists. Stretching our definition a little, we state that $\sqcup\emptyset = \bot$. Indeed, the more elements we join, the larger the result is, so, when joining no element at all, we obtain the least element of all. As a consequence, all our CPO are so-called *pointed*, i.e., they feature a least element $\bot$.

Example 2.5 (Chains, CPO)**.**

1. *In Fig. 2.1.(a),* $\{a, c, f, g\}$ *forms a chain.*
2. *Figure 2.1.(a) is a CPO. In fact, every finite poset is a CPO. Indeed every finite chain always has a lub.*
3. *Figure 2.1.(b) is not a CPO because infinite subsets of* $\mathbb{N}$ *do not have a lub in* $\mathbb{N}$.
4. *Figure 2.1.(c) is a CPO as every infinite subset of* $\mathbb{N} \cup \{\infty\}$ *has a lub:* ∞.
5. *For any (even infinite) set* X*, its powerset poset* $(\mathcal{P}(X), \subseteq)$ *is a CPO.* ♦

Remark 2.2. *Our definition of CPO deviates slightly from the usual definition from domain theory [Scott and Strachey, 1971], which uses directed subsets instead of chains. While we believe that the two definitions are equivalent (assuming Zorn's lemma), we chose to use Def. 2.3 as it makes the connection with limits of iterations more explicit.* ◊

2.1.5 Lattices

Posets provide the bare minimum algebraic structure, but many ordered sets have a richer structure, than we can exploit. Such structures include, in particular, *lattices*. Lattices are posets where the lub and glb always exist, either only for pairs of elements (and, by extension, for finite sets of elements), or for arbitrary (possibly infinite) families of elements, in which case the lattice is said to be complete:

Definition 2.4 (Lattice)**.** *A* lattice $(X, \sqsubseteq, \sqcup, \sqcap)$ *is a poset such that* $\forall a, b \in X\colon a \sqcup b$ *and* $a \sqcap b$ *exist.* ■

Definition 2.5 (Complete lattice)**.** *A* complete lattice $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *is a poset such that:*

1. $\forall A \subseteq X\colon \sqcup A$ *exists;*
2. $\forall A \subseteq X\colon \sqcap A$ *exists;*
3. X *has a least element* $\bot$*;*
4. X *has a greatest element* $\top$*.* ■

Naturally, a complete lattice is both a lattice and a CPO. Less obviously, to get a complete lattice, either Def. 2.5.1 or Def. 2.5.2 is sufficient, as each one implies the other. Indeed, assuming that $\forall A \subseteq X\colon \sqcup A$ exists, then $\forall B \subseteq X\colon \sqcap B = \sqcup \{ x \in X \mid \forall a \in B\colon x \sqsubseteq a \}$ also exists: we can reduce a glb to a lub, which always exists (the converse naturally holds). Moreover, both imply Def. 2.5.3–4. Indeed, we have: $\bot = \sqcup \emptyset = \sqcap X$ and $\top = \sqcap \emptyset = \sqcup X$. More information on lattices can be found in [Birkhoff, 1967].

Example 2.6 (Lattices, Complete lattices)**.**

1. *For any set* X*, the powerset:*

$$(\mathcal{P}(X), \subseteq, \cup, \cap, \emptyset, X) \tag{2.1}$$

   is a complete lattice, as we can compute the intersection and union of arbitrary many (including infinitely many) sets. The Hasse diagram for $\mathcal{P}(\mathbb{Z})$ *is presented in Fig. 2.2.*

2. *We construct an integer interval lattice as follows:*

$$(\{ [a, b] \mid a, b \in \mathbb{Z},\ a \leq b \} \cup \{ \bot \}, \subseteq, \sqcup, \cap) \tag{2.2}$$

   To simplify, we assimilate here a pair of bounds (a, b) *with the set* $[a, b]$ *of integers comprised between* a *and* b*, taking care that* $a \leq b$*. We add a special, unique, smallest element* $\bot$ *to represent* $\emptyset$*. We thus use the classic set operators:* $\subseteq$ *as partial order and* $\cap$ *as glb, as intervals are closed under intersection. However, they are not closed under set union, hence, we define the lub as* $[a, b] \sqcup [a', b'] \stackrel{\text{def}}{=} [\min(a, a'), \max(b, b')]$*, while* $\forall x\colon x \sqcup \bot = \bot \sqcup x = x$*. Indeed,* $[a, b] \sqcup [a', b']$ *computes the smallest interval containing intervals* $[a, b]$ *and* $[a', b']$*.*

3. *The integer interval lattice (2.2) from the previous point is not complete as the infinite family of intervals* $\{ [0, i] \mid i \geq 0 \}$ *has no lub. We complete the interval lattice into a complete lattice by allowing positive and negative infinities as bounds:*

$$\begin{aligned}&(\{ [a, b] \mid a \in \mathbb{Z} \cup \{ -\infty \},\ b \in \mathbb{Z} \cup \{ +\infty \},\ a \leq b \} \cup \{ \bot \},\\ &\subseteq, \sqcup, \cap, \bot, [-\infty, +\infty])\end{aligned} \tag{2.3}$$

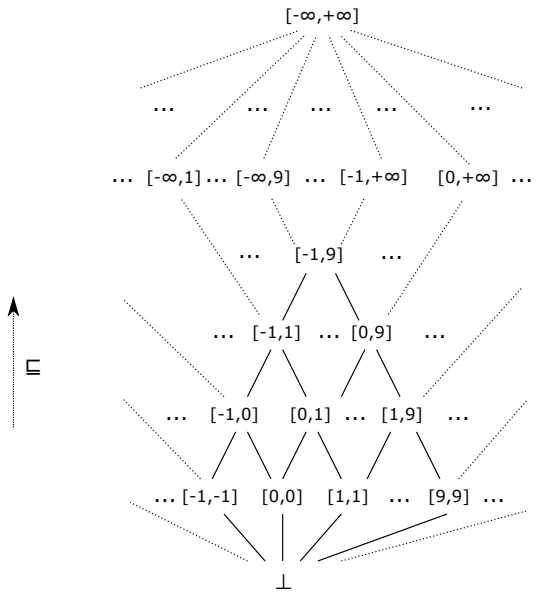


Figure 2.3: Hasse diagram for the interval lattice, with bounds extended to ∞.

Lubs are computed as: $\sqcup_{i\in I}\,[a_i, b_i] \stackrel{\text{def}}{=} [\min_{i\in I} a_i,\ \max_{i\in I} b_i]$. *Moreover, the lattice now has a greatest element:* $[-\infty, +\infty]$. *The complete lattice of intervals is illustrated in Fig. 2.3.*

4. *The* divisibility lattice *is defined as* $(\mathbb{N}^*, |, \mathrm{lcm}, \gcd)$, *where* $x|y$ *means that* x *divides* y, *or equivalently that* y *is a multiple of* x, *i.e.,* $\exists k \in \mathbb{N}^*\colon xk = y$. *Then, the lub and glb are respectively the least common multiple,* lcm, *and the greatest common divisor,* gcd. *This lattice is illustrated in Fig. 2.4. It is the basis of a congruence analysis that we will study in Sect. 4.8 to infer properties such as the fact that a variable is a multiple of some constant. Note that this lattice is not complete as chains such as* $\{2, 4, 8, 16, \ldots\}$ *have no lub. Moreover, although an arbitrary non-empty, possibly infinite integer set* $A \subseteq \mathbb{N}^*$ *has a glb, there is no greatest element, i.e., no glb for the empty set.* ♦

2.1.6 Distributivity

A lattice $(A, \sqsubseteq, \sqcup, \sqcap)$ is said to be *distributive* if $\sqcup$ distributes over $\sqcap$, and $\sqcap$ distributes over $\sqcup$, i.e., $\forall a, b, c \in A\colon a\sqcup(b\sqcap c) = (a\sqcup b)\sqcap(a\sqcup c)$ and $a\sqcap(b\sqcup c) = (a\sqcap b)\sqcup(a\sqcap c)$. Actually, very few lattices we will encounter are distributive, and these generally correspond to concrete worlds rather than abstract worlds:

Example 2.7 (Distributive lattices)**.**

1. *The powerset lattice (2.1) is distributive as* $\cup$ *distributes over* $\cap$, *and* $\cap$ *distributes over* $\cup$.

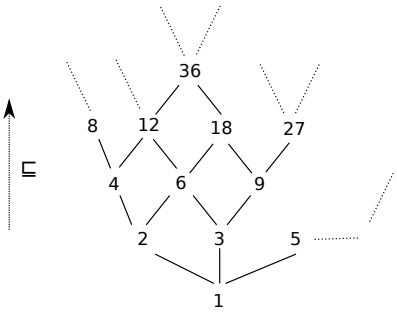


Figure 2.4: Hasse diagram for the divisor lattice $(\mathbb{N}^*, |, \mathrm{lcm}, \mathrm{gcd})$ where $x|y$ if x divides y.

2. *The interval lattice (2.3) is not distributive. Indeed, on the one hand* $([0,0] \sqcup [2,2]) \sqcap [1,1] = [0,2] \sqcap [1,1] = [1,1]$*; on the other hand* $([0,0] \sqcap [1,1]) \sqcup ([2,2] \sqcap [1,1]) = \bot \sqcup \bot = \bot$. ♦

As we will see, non-distributivity is a common cause of precision loss. Indeed, a classic analysis design (Sect. 3.5) tends to join information, such as the result of different control-flow paths merging at some common program location, as early as possible, favoring computations of the form $(a \sqcup b) \sqcap c$. A formulation such as $(a \sqcap c) \sqcup (b \sqcup c)$, which delays the join, requires more operators, and is thus more costly. The later is, however, always at least as precise as the former, and it is strictly more precise if the lattice is not distributive, as shown in Ex. 2.7 — as $\bot \sqsubset [1,1]$. Yet, limiting ourselves to distributive lattices would severely hinder our ability to choose the appropriate domain for each task. When precision matters, we may consider a tradeoff where the later formulation, delaying joins, is used, but parsimoniously (Sect. 6.3.4).

This phenomenon is well-known in the field of data-flow analysis [Kildall, 1973], where the second formulation leads to the *meet-over-all-paths* algorithm, and the former leads to the *least fixpoint* algorithm.

2.1.7 Sublattice

A static analysis replaces costly, expressive representations with simpler, more restricted ones. For instance, it can use intervals to represent integer sets. Note that the set of intervals (2.3) is a subset of the sets of integers (2.1), and that both are (complete) lattices. A natural question is thus whether we should require our abstract elements to actually form a *sublattice* of the concrete lattice, where a sublattice is defined as follows:

Definition 2.6 (Sublattice)**.** $(X', \sqsubseteq, \sqcup, \sqcap)$ *is a sublattice of* $(X, \sqsubseteq, \sqcup, \sqcap)$ *if:* $X' \subseteq X$*; we use the same order* $\sqsubseteq$ *on both* X *and* X'*; and* X' *is closed under* $\sqcup$ *and* $\sqcap$*, i.e., lubs and glbs exist in* X' *and coincide with those in* X*.* ■

Example 2.8 (Sublattice)**.**

1. *If* $X' \subseteq X$*, then* $(\mathcal{P}(X'), \subseteq, \cup, \cap)$ *is a sublattice of* $(\mathcal{P}(X), \subseteq, \cup, \cap)$*.*

2. *The interval lattice (2.3) is* not *a sublattice of the lattice of powerset of integers (2.1). Indeed, the interval join is defined as* $[a, b] \sqcup [a', b'] = [\min(a, a'), \max(b, b')]$*, which may differ from the set union* $[a, b] \cup [a', b']$*. Consider, for instance, that* $[0, 0] \sqcup [2, 2] = [0, 2] = \{0, 1, 2\}$*, while* $[0, 0] \cup [2, 2] = \{0, 2\}$*.* ♦

Not being a sublattice means that the result of some operations, such as the join of two intervals, must be approximated to stay within the abstract world of intervals. The situation is similar to that of distributivity: we lack a strong algebraic property, here being a sublattice, which results in a loss of precision, but limiting ourselves to sublattices would hinder our ability to use extremely useful abstractions, such as intervals.

The required connection between the concrete world and the abstract world is actually more subtle than the sublattice property, as we will discuss shortly in Sect. 2.3. We can already argue that Abstract Interpretation is particularly lax when it comes to algebraic requirements, especially on the abstract world. This may be a key to its success, as it leaves abstractions open to many possibilities.

2.1.8 Derived Ordered Structures

Given one or several ordered structures that can be posets, CPO, or (complete) lattices, we can derive new ordered structures of the same nature by duality (i.e., reversing the order and switching lubs with glbs and $\top$ with $\bot$), by lifting (adding a least element $\bot$), by Cartesian product, by smashed product (or coalescent product, i.e., a product where least elements are fused) or, finally, by point-wise lifting. More precisely, we define:

Definition 2.7 (Derived order structures)**.** *Assume that* $(X_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1)$ *and* $(X_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2)$ *are complete lattices (resp. lattices, CPO, posets), then so are the ordered structures derived by:*

1. *Duality:* $(X_1, \sqsupseteq_1, \sqcap_1, \sqcup_1, \top_1, \bot_1)$*.*

2. *Lifting:* $(X_1 \cup \{\bot\}, \sqsubseteq, \sqcup, \sqcap, \bot, \top_1)$ *where:*

   - $\bot \notin X_1$ *is a new least element;*
   - $a \sqsubseteq b \overset{\text{def}}{\iff} (a = \bot) \vee (a \sqsubseteq_1 b)$*;*
   - $\bot \sqcup a \overset{\text{def}}{=} a \sqcup \bot \overset{\text{def}}{=} a$*, and* $a \sqcup b \overset{\text{def}}{=} a \sqcup_1 b$ *if* $a, b \neq \bot$*;*
   - $\bot \sqcap a \overset{\text{def}}{=} a \sqcap \bot \overset{\text{def}}{=} \bot$*, and* $a \sqcap b \overset{\text{def}}{=} a \sqcap_1 b$ *if* $a, b \neq \bot$*;*
   - $\top_1$ *is unchanged.*

3. *Cartesian product:* $(X_1 \times X_2, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *where:*

   - $(x, y) \sqsubseteq (x', y') \overset{\text{def}}{\iff} (x \sqsubseteq_1 x') \wedge (y \sqsubseteq_2 y')$*;*
   - $(x, y) \sqcup (x', y') \overset{\text{def}}{=} (x \sqcup_1 x',\ y \sqcup_2 y')$*;*
   - $(x, y) \sqcap (x', y') \overset{\text{def}}{=} (x \sqcap_1 x',\ y \sqcap_2 y')$*;*

- $\bot \stackrel{\text{def}}{=} (\bot_1, \bot_2)$;
- $\top \stackrel{\text{def}}{=} (\top_1, \top_2)$.

4. *Smashed product:*

$$(((X_1 \setminus \{\bot_1\}) \times (X_2 \setminus \{\bot_2\})) \cup \{\bot\}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$$

where $\sqsubseteq$, $\sqcup$, $\sqcap$, $\top$ *are defined as in the regular product on* $(X_1 \setminus \{\bot_1\}) \times (X_2 \setminus \{\bot_2\})$, *and then lifted with a new least element* $\bot$.

The smashed product can also be viewed as the Cartesian product $X_1 \times X_2$, *quotiented by the equivalence relation* $(x, y) \equiv (x', y') \stackrel{\text{def}}{\iff} (x = x' \wedge y = y') \vee ((x = \bot_1 \vee y = \bot_2) \wedge (x' = \bot_1 \vee y' = \bot_2))$, *which identifies elements where at least one component is the least element.*

5. *Point-wise lifting:* $(S \to X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *where:*

- S *is an arbitrary (finite or infinite) set;*
- $x \sqsubseteq y \stackrel{\text{def}}{\iff} \forall s \in S\colon x(s) \sqsubseteq_1 y(s)$;
- $\forall s \in S\colon (x \sqcup y)(s) \stackrel{\text{def}}{=} x(s) \sqcup_1 y(s)$;
- $\forall s \in S\colon (x \sqcap y)(s) \stackrel{\text{def}}{=} x(s) \sqcap_1 y(s)$;
- $\forall s \in S\colon \bot(s) \stackrel{\text{def}}{=} \bot_1$;
- $\forall s \in S\colon \top(s) \stackrel{\text{def}}{=} \top_1$.

6. *Smashed point-wise lifting:*

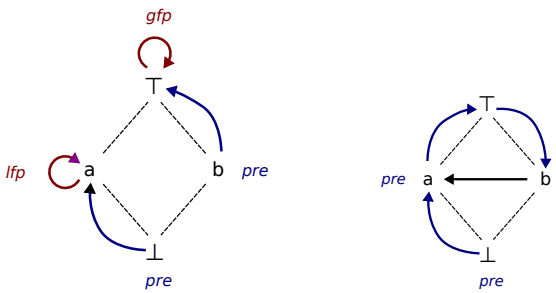
$$((S \to (X_1 \setminus \{\bot_1\})) \cup \{\bot\}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$$

where $\bot$ *is a new least element, and* $\bot_1$ *is no longer in the domain of the elements of the structure.*

Similarly to the smashed product, the smashed point-wise lifting quotients the classic point-wise lifting with the equivalence relation $x \equiv y \stackrel{\text{def}}{\iff} (\forall s \in S\colon x(s) = y(s)) \vee (\exists s, s' \in S\colon x(s) = \bot_1 \wedge y(s') = \bot_1)$, *which identifies elements where at least one component is the least element.* ■

2.2 Fixpoints

We call *operator* a function $f : X \to X$ with the same domain and codomain. Given an operator f, a *fixpoint* is any value x such that $f(x) = x$. Fixpoints are pervasive in Mathematics and Theoretical Computer Science. They provide a uniform way to discuss about the solutions to many kinds of equations. In program semantics, fixpoints closely model invariants, as we will see formally in Sect. 3.3. Informally, if f models the action of a loop body, then $f(x) = x$ means that x is left invariant by a loop iteration. As we will see shortly, fixpoints are also related to computation by iteration, such as presented informally in Fig. 1.4. Ordered structures provide a rich set of theorems ensuring the existence of fixpoints. We present them now and will exploit them in the following sections.



(a) (b)

Figure 2.5: Operators on posets: (a) is monotonic and has two fixpoints, a and $\top$; (b) is non-monotonic and has no fixpoint.

2.2.1 Fixpoints

Let us first complete our definitions:

Definition 2.8 (Fixpoints, Prefixpoints, Postfixpoints)**.**
Given a poset $(X, \sqsubseteq)$ and an operator $f : X \to X$:

1. *x is a* fixpoint *of f if $f(x) = x$.*
   We denote as $\mathrm{fp}(f) \stackrel{\text{def}}{=} \{\, x \in X \mid f(x) = x \,\}$ the set of fixpoints of f.
2. *x is a* prefixpoint *of f if $x \sqsubseteq f(x)$.*
3. *x is a* postfixpoint *of f if $f(x) \sqsubseteq x$.*
4. *$\mathrm{lfp}_x\, f \stackrel{\text{def}}{=} \min \{\, y \in \mathrm{fp}(f) \mid x \sqsubseteq y \,\}$, if it exists, is the* least fixpoint *of f greater than x.*
5. *$\mathrm{lfp}\, f \stackrel{\text{def}}{=} \mathrm{lfp}_\perp\, f$, if it exists, is the least fixpoint of f.*
6. *Dually, $\mathrm{gfp}_x\, f \stackrel{\text{def}}{=} \max \{\, y \in \mathrm{fp}(f) \mid y \sqsubseteq x \,\}$ is the* greatest fixpoint *of f smaller than x.*
7. *$\mathrm{gfp}\, f \stackrel{\text{def}}{=} \mathrm{gfp}_\top\, f$ if the greatest fixpoint of f.* ■

Note that an operator does not necessarily have any fixpoint at all. Figure 2.5 presents a Hasse diagram for the lattice $\{\perp, a, b, \top\}$ as well as two operators. Operator (a) has two fixpoints, $a = \mathrm{lfp}$ and $\top = \mathrm{gfp}$, while operator (b) has no fixpoint at all.

2.2.2 Monotony and Continuity

We need some extra hypotheses on an operator f to guarantee the existence of fixpoints. Let us first present a set of useful properties functions can enjoy:

Definition 2.9 (Monotony, Continuity)**.**

1. *Monotonicity: a function* $f : (A_1, \sqsubseteq_1) \to (A_2, \sqsubseteq_2)$ *between two posets is* monotonic *if* $\forall x, y \in A_1 \colon x \sqsubseteq_1 y \implies f(x) \sqsubseteq_2 f(y)$.

2. *Continuity: a function* $f : (A_1, \sqsubseteq_1, \sqcup_1) \to (A_2, \sqsubseteq_2, \sqcup_2)$ *between two CPO is* continuous *if for every chain* $C \subseteq A_1$, $\{ f(c) \mid c \in C \}$ *is also a chain and the limits coincide:* $f(\sqcup_1 C) = \sqcup_2 \{ f(c) \mid c \in C \}$.

3. *Join morphism: a function* $f : (A_1, \sqsubseteq_1, \sqcup_1) \to (A_2, \sqsubseteq_2, \sqcup_2)$ *between two lattices is a* join morphism*, or* $\sqcup-$morphism*, if* $\forall a, b \in A_1 \colon f(a \sqcup_1 b) = f(a) \sqcup_2 f(b)$*; it is a* complete $\sqcup-$morphism *if* A_1 *and* A_2 *are complete lattices and the property extends to arbitrary lubs, i.e.,* $\forall X \subseteq A_1 \colon f(\sqcup_1 X) = \sqcup_2 \{ f(x) \mid x \in X \}$.

4. *Extensivity: an operator* $f : (A, \sqsubseteq) \to (A, \sqsubseteq)$ *is* extensive *if* $\forall a \in A \colon a \sqsubseteq f(a)$*, and* reductive *if* $\forall a \in A \colon f(a) \sqsubseteq a$. ■

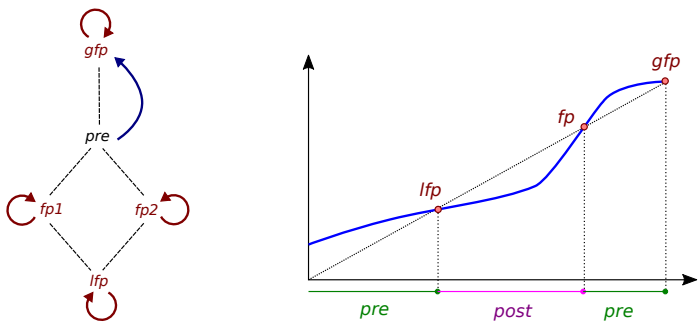
These definitions can be justified informally. In terms of information theory, where the partial order measures a quantity of information, monotonicity means that, given more information as input, a function must output more information. In terms of program semantics, the more input we feed a program, the more behaviors it will exhibit.

Continuity implies monotonicity: when $x \sqsubseteq y$, simply consider the chain $\{x, y\}$ to get that $f(x) \sqcup f(y) = f(x \sqcup y) = f(y)$, i.e., $f(x) \sqsubseteq f(y)$. Continuity goes one step further and requires that functions preserve limits, i.e., there is "no surprise" at the limit, and the result of $f(\sqcup A)$ can be intuited by observing the sequence $\{ f(a) \mid a \in A \}$. For instance, an operator f over $(\mathbb{N} \cup \{\infty\}, \leq)$ such that $f(x) = 0$ if $x \neq \infty$ but $f(\infty) = 1$ is not continuous, as $f(\infty)$ cannot be intuited from the set of all finite images $\{ f(x) \mid x \in \mathbb{N} \} = \{0\}$. Scott and Strachey [1971] postulate that computable functions are continuous, which is understandable given the inherent finite capabilities of computers.

Join morphisms and complete join morphisms distribute f over joins in lattices and complete lattices. Naturally, they imply monotonicity, and a complete $\sqcup-$morphism is moreover continuous. In terms of program semantics, a $\sqcup-$morphism indicates that the set of possible behaviors of a program on a set of inputs can be derived by observing its behaviors on each input separately, and joining them. This is also a natural property we expect program semantics to have and, although we will not need this property in our fixpoint theorems here, it will play a role when defining the semantics of our language in the next chapter.

Extensivity is essentially useful when combined with monotonicity to bootstrap iteration: starting from an arbitrary x, we have $x \sqsubseteq f(x)$ by extensivity; then, applying monotonicity, $f(x) \sqsubseteq f^2(x)$, etc., so that the sequence $\{ f^i(x) \mid i \in \mathbb{N} \}$ is increasing.

In Abstract Interpretation, these properties often hold for concrete operators, but are often lost in the abstract world for the sake of generality — the same way abstract worlds feature less algebraic properties than concrete ones, abstract operators feature less algebraic properties than concrete ones.



(a) (b)

Figure 2.6: Tarski's fixpoint theorem: (a) the lattice $\{\mathsf{lfp}, \mathsf{fp1}, \mathsf{fp2}, \mathsf{gfp}\}$ (in red) of fixpoints of a monotonic operator in a complete lattice $\{\mathsf{lfp}, \mathsf{fp1}, \mathsf{fp2}, \mathsf{pre}, \mathsf{gfp}\}$; and (b) illustration of the least fixpoint lfp bracketed between prefixpoints and postfixpoints.

2.2.3 Fixpoint Theorems

Monotonicity is related to the existence of fixpoints. For instance, the operator enjoying fixpoints in Fig. 2.5.(a) is monotonic, while the operator in Fig. 2.5.(b) is not monotonic, and has no fixpoint.

Tarski fixpoint. This connection is formalized in Tarski [1955]'s Theorem:

Theorem 2.1 (Tarski's Theorem)**.** *If $f \in X \to X$ is a monotonic operator in a complete lattice $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, then the set of fixpoints $\mathrm{fp}(f)$ is a non-empty complete lattice. In particular, $\mathrm{lfp}\, f$ exists.*
Furthermore, $\mathrm{lfp}\, f = \sqcap \{\, x \in X \mid f(x) \sqsubseteq x \,\}$. ■

The complete lattice of fixpoints of a monotonic operator is illustrated in Fig. 2.6.(a). Note that it is not a sublattice as the join, gfp, of fixpoints $\mathsf{fp1}$ and $\mathsf{fp2}$ does not coincide with the join, pre, in the original lattice.

Beside the existence of fixpoints, and in particular a most precise, least fixpoint, a key result of Tarski's Theorem is its characterization of the least fixpoint as the meet of all postfixpoints. This is illustrated in Fig. 2.6.(b): when exploring $f(x)$ from the least element $\bot$, one encounters only prefixpoints until reaching the least fixpoint, which is also the first postfixpoint. An important consequence for static analysis is that any postfixpoint of f is a sound over-approximation of $\mathrm{lfp}\, f$.

Kleene fixpoint. Another useful fixpoint theorem, found, e.g., in [Cousot and Cousot, 1977], is inspired by Kleene [1964]'s star construction:

Theorem 2.2 (Kleene's Theorem)**.** *If $f \in X \to X$ is a continuous operator in a CPO $(X, \sqsubseteq, \sqcup, \top)$, then $\mathrm{lfp}\, f$ exists. Moreover, $\mathrm{lfp}\, f = \sqcup \{\, f^i(\bot) \mid i \in \mathbb{N} \,\}$.* ■

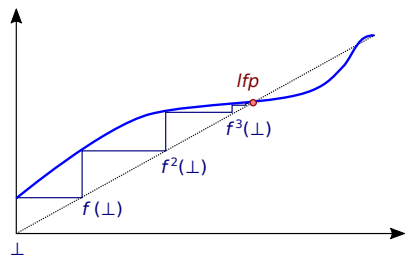


Figure 2.7: Kleene fixpoint iterations, from $\perp$ to $\mathrm{lfp}\, f$.

This theorem requires stronger hypotheses on the operator f than Tarski's Theorem, i.e., continuity instead of monotonicity, but it does not require a complete lattice, only a CPO. More interestingly, it expresses $\mathrm{lfp}\, f$ as a limit of an iteration, by computing the chain $\{ f^i(\perp) \mid i \in \mathbb{N} \}$. This iteration is illustrated in Fig. 2.7. This iteration scheme makes the theorem *constructive*: one can imagine the process of computing iterations one by one from $\perp$, which is what we actually did in Fig. 1.4. Naturally, the iteration often requires passing to the limit after an infinite (yet countable) number of iterations. One particular case is when the CPO X has no infinite strictly increasing chain. In this case, the iteration is guaranteed to converge in finite time, bounded by the maximal height of chains. Dually, it is possible to relax the continuity condition and obtain a theorem, due to Cousot and Cousot [1979b], with even weaker hypotheses than Tarski's theorem, and which is still constructive. However, the iteration may not even converge in a countable number of steps and requires, instead, transfinite iterations, up to larger ordinals.

These theorems are useful to provide a rigorous definition of program semantics, as we will see in the next chapter, but they do not often lead to effective algorithms. To compute effectively and efficiently, we will introduce, in the abstract, convergence acceleration methods.

2.3 Approximations

We now study the relationships between concrete worlds and abstract worlds, and state key results ensuring the soundness of abstract computations with respect to concrete ones. We will first discuss the abstraction of elements, then of operators, and finally of fixpoints.

2.3.1 Concretization

The minimum structure we require in the concrete and the abstract worlds is a partial order that models an amount of information — larger elements expose more program behaviors. Thus, the concrete world is a poset $(C, \leq)$, for instance integer powersets (2.1), and the abstract world is another poset $(A, \sqsubseteq)$, for instance intervals (2.3). The minimum connection we request between these worlds is a *concretization* function, traditionally denoted as γ:

Definition 2.10 (Concretization)**.** *A concretization function* $\gamma \in (A, \sqsubseteq) \to (C, \leq)$ *is a* monotonic *function assigning a concrete meaning, in* C*, to each abstract element in* A*.* ■

The monotonicity simply states that coarser abstract elements represent coarser concrete elements.

Example 2.9 (Concretization)**.**

1. *The interval abstraction (2.3) already considers that an interval is a set of integers. The concretization from intervals to sets (2.1) is thus the identity.*

2. *Consider an alternate abstract domain of intervals where an interval is represented by either a* pair of bounds*, or* $\bot$*:*

$$\{\, (a, b) \mid a \in \mathbb{Z} \cup \{\, -\infty \,\},\, b \in \mathbb{Z} \cup \{\, +\infty \,\},\, a \leq b \,\} \cup \{\, \bot \,\} \tag{2.4}$$

Then, the concretization is given by:

$$\begin{array}{lcl} \gamma((a, b)) & \stackrel{\text{def}}{=} & \{\, x \in \mathbb{Z} \mid a \leq x \leq b \,\} \\ \gamma(\bot) & \stackrel{\text{def}}{=} & \emptyset \end{array}$$

♦

It is convenient, on paper, to confuse an interval as a set of integers (Ex. 2.9.1) and an interval as a pair of bounds (Ex. 2.9.2). However, the later is far more effective when it comes to machine representation and manipulation. Thus, it is important not to see the abstract world only as a subset of the concrete world: the abstract world also adds a notion of representation, which is key to effectiveness. Hence, we separate the concrete world from the abstract world, always mediate through a concretization function γ to compare concrete and abstract elements, and avoid confusing $a \in A$ with $\gamma(a) \in C$.

Note (Interval notation)**.** *Now that the distinction between the abstract world and the concrete world is clear, in the rest of the tutorial, when discussing intervals, we will write* $[a, b]$ *to actually denote the pair of bounds* (a, b) *that internally represents an interval, and write explicitly* $\gamma([a, b])$ *when discussing about the set of integers in this interval.* ◊

Note that γ does not need to be onto — i.e., injective. It is possible to imagine an abstract world where several abstract elements represent the same concrete element. Consider, for instance, a variant of (2.4), $\{\, (a, b) \mid a \in \mathbb{Z} \cup \{\, -\infty \,\},\, b \in \mathbb{Z} \cup \{\, +\infty \,\} \,\}$, where we do not enforce $a \leq b$, and thus, any pair such that $a > b$ is a representation for the empty set. This is mainly useful, as we will see later, in the case of domains where computing unique, normal forms for an abstract element is not possible or would be too time-consuming.

2.3.2 Soundness

Given a concrete and an abstract domain, a sound abstraction is simply an abstract element that over-approximates a concrete one, up to the interpretation given by the concretization:

Definition 2.11 (Soundness, Exactness)**.**
$a \in A$ is a sound *abstraction of $c \in C$ if and only if $c \leq \gamma(a)$.*
It is moreover exact *if $c = \gamma(a)$.* ■

The case where the equality holds corresponds to a rare case where the concrete element can be exactly be represented in the abstract (such as, for instance, a contiguous set of integers, which can be represented exactly as an interval).

We can now formalize the intuition given in Sect. 1.2 for safety verification problems. Given a specification S, then if:

- the specification S can be exactly represented in the abstract as $S^\sharp$, i.e., $S = \gamma(S^\sharp)$, and
- the result $P^\sharp$ of a static analysis is a sound abstraction of the concrete semantics P, i.e., $P \subseteq \gamma(P^\sharp)$, and
- $P^\sharp \sqsubseteq S^\sharp$, i.e., the analyzer proves in the abstract that the program satisfies its specification,

then, the program indeed satisfies its specification in the concrete, i.e., $P \subseteq S$. Note that, not only the computation of $P^\sharp$, but also the check $P^\sharp \sqsubseteq S^\sharp$, are done entirely in the abstract world. Hence, effective abstract algorithms lead to sound and effective static analyses.

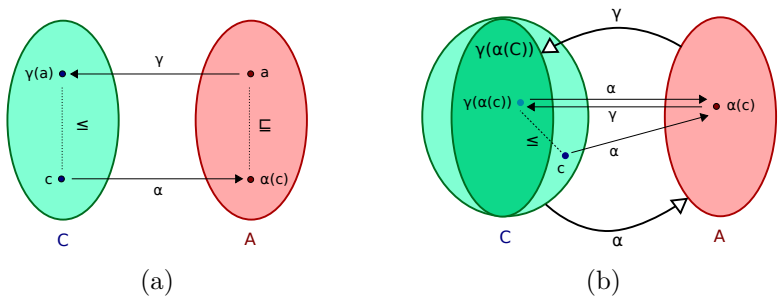
2.3.3 Galois connection

While a monotonic concretization γ is sufficient to reason about soundness, more structure, if available, can help us design sound and accurate analyses. In the standard Abstract Interpretation framework [Cousot and Cousot, 1977], we assume additionally the existence of a monotonic *abstraction function* $\alpha : C \to A$ that associates an abstract element to a concrete one such that (α, γ) forms a Galois connection:

Definition 2.12 (Galois connection)**.** *Given two posets $(C, \leq)$ and $(A, \sqsubseteq)$, the pair $(\alpha : C \to A,\ \gamma : A \to C)$ is a* Galois connection *if:*

$$\forall a \in A,\ c \in C,\ c \leq \gamma(a) \iff \alpha(c) \sqsubseteq a \tag{2.5}$$

which is denoted as $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$.
α and γ are said to be adjoint functions*, where the abstraction α is the upper adjoint and the concretization γ is the lower adjoint.* ■



(a) (b)

Figure 2.8: A Galois connection (a) and a Galois embedding (b).

The fundamental property of Galois connections (2.5) provides, in a very compact form, a strong connection between the concrete and the abstract world. The following theorem provides a less compact, but equivalent view:

Theorem 2.3 (Alternate characterization of Galois connections)**.** *We have a Galois connection* $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *if and only if the function pair* (α, γ) *satisfies all the following properties:*

1. γ *is monotonic;*
2. α *is monotonic;*
3. $\gamma \circ \alpha$ *is extensive, i.e.,* $\forall c \in C\colon c \sqsubseteq \gamma(\alpha(c))$*;*
4. $\alpha \circ \gamma$ *is reductive, i.e.,* $\forall a \in A\colon \alpha(\gamma(a)) \leq a$*.* ■

Apart from the monotonicity of γ, which we already stated, and that of α, which is also intuitive as we want more precise concrete elements to be abstracted more precisely, the extensivity of $\gamma \circ \alpha$ is particularly interesting. It states that, going through the abstract world A and back gives a result that is either equal or less precise than staying in the concrete. The result is strictly less precise when the original concrete element has no exact representation in the abstract, so that we lose information during the conversion between the concrete and the abstract world. This is depicted in Fig. 2.8.(a).

The reductivity of $\alpha \circ \gamma$ shows that, coming from the abstract and going back to the abstract through the concrete, we may end up with a smaller abstract element. In fact, this happens only when a concrete element has several different abstract representations, in which case α will naturally choose the smallest one for the abstract order $\sqsubseteq$. This has generally no consequence on the precision of an abstract computation — these elements all have the same concretization — provided that the operators are carefully designed to be independent from the choice of an abstract representation among several equivalent ones. In many, but not all, cases, there is only a single abstract representation at most for each concrete property anyway. This case will be discussed in more details in Sect. 2.3.4.

Example 2.10 (Galois connection)**.** *Following up on Ex. 2.9.2, we define the Galois connection between the concrete domain $\mathcal{P}(\mathbb{Z})$ and the abstract domain of intervals represented as pairs of bounds (2.4):*

$$\{\, (a,b) \mid a \in \mathbb{Z} \cup \{\, -\infty \,\},\, b \in \mathbb{Z} \cup \{\, +\infty \,\},\, a \leq b \,\} \cup \{\, \bot \,\}$$

as follows:

$$\begin{array}{lcl} \gamma((a,b)) & \stackrel{\text{def}}{=} & \{\, x \in \mathbb{Z} \mid a \leq x \leq b \,\} \\ \gamma(\bot) & \stackrel{\text{def}}{=} & \emptyset \end{array}$$

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

We note, for instance, that $\alpha(\{0,2\}) = [0,2]$ and that $\gamma([0,2]) = \{0,1,2\}$. Hence, $\gamma \circ \alpha$ is extensive and not the identity, i.e., our abstraction generally looses precision. ♦

In addition to Thm. 2.3, Galois connections enjoy many useful properties:

Theorem 2.4 (Galois connection properties)**.** *Given a Galois connection $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, we have:*

1. *$\gamma \circ \alpha \circ \gamma = \gamma$ and $\alpha \circ \gamma \circ \alpha = \alpha$;*
2. *$\alpha \circ \gamma$ and $\gamma \circ \alpha$ are idempotent;*
3. *$\forall c \in C\colon \alpha(c) = \sqcap \{\, a \mid c \leq \gamma(a) \,\}$;*
4. *$\forall a \in A\colon \gamma(a) = \vee \{\, c \mid \alpha(c) \sqsubseteq a \,\}$;*
5. *α maps concrete lubs to abstract lubs:*
   $\forall X \subseteq C$: if $\vee X$ exists, then $\alpha(\vee X) = \sqcup \{\, \alpha(x) \mid x \in X \,\}$;
6. *γ maps abstract glbs to concrete glbs:*
   $\forall X \subseteq A$: if $\sqcap X$ exists, then $\gamma(\sqcap X) = \wedge \{\, \gamma(x) \mid x \in X \,\}$. ■

Theorem 2.4.2 states that, although passing through the abstract once may lose precision, as $\gamma \circ \alpha$ is extensive (Thm. 2.3.3), further round-trips through the abstract world do not lose any more precision, as $(\gamma \circ \alpha)^2 = \gamma \circ \alpha$. Theorem 2.4.3–4 states that, in a Galois connection, one adjoint can be derived from the other one.

Additionally, Thm. 2.4.3 states a very important property relating Galois connections, soundness, and optimality. Recall that $c \leq \gamma(a)$ means, by Def. 2.11, that a is a sound abstraction of c. Then, given $c \in C$, first recall that $c \leq \gamma(\alpha(c))$, which means that $\alpha(c)$ is a sound abstraction of c. Moreover, $\alpha(c) = \sqcap \{\, a \mid c \leq \gamma(a) \,\}$ means literally that $\alpha(c)$ is the best (i.e., smallest) sound abstraction of c:

Corollary 2.5 (Best abstraction)**.** *If we have a Galois connection $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, then $\forall c \in C\colon \alpha(c)$ is the* best abstraction *of c, i.e., the smallest abstract element which is a sound abstraction of c.* ■

For instance, in the interval domain, the abstraction $\alpha(X)$ of a set of integers X computes $[\min X, \max X]$, which is indeed the tightest interval that contains all the values from X.

Example 2.11 (Absence of a Galois connection)**.** *Not all abstract domains enjoy a Galois connection. Consider the affine inequalities domain, that we mentioned in Sect. 1.1.2 and will present in details in Sect. 5.3. Intuitively, it abstracts a set of points as a convex polyhedron, and a best abstraction would be the smallest polyhedron enclosing these points. Some shapes, such as discs, do not have a smallest enclosing polyhedron, hence, no α function, and so, no Galois connection can exist. We can still reason about soundness through γ, though, and choose a, possibly arbitrary, way to soundly abstract a disc.* ♦

Finally, an important consequence of Thm. 2.4.6 is that the set of concrete properties that can be exactly represented in the abstract must be closed by concrete meet.

Example 2.12 (Closure by conjunction and Galois connections)**.** *We consider, as concrete world, the poset $(\mathcal{P}(\mathbb{Z}), \subseteq)$ of sets of integers, and the sign abstraction already discussed informally in Fig. 1.2.*

Figure 2.9 proposes formally two variants. A naive version, (a), has only four elements: positive (≥ 0), negative (≤ 0), $\bot$, and $\top$, with the natural concretization: $\gamma((\geq 0)) = \mathbb{N}$, $\gamma((\leq 0)) = -\mathbb{N}$, $\gamma(\bot) = \emptyset$, and $\gamma(\top) = \mathbb{Z}$. We note that the set of representable properties is not closed under intersection; indeed, $\gamma((\geq 0)) \cap \gamma((\leq 0)) = \{0\}$, but $\{0\}$ is not exactly representable in the abstract. In fact, $\{0\}$ has two incomparable abstractions: (≥ 0) and (≤ 0), but no best abstraction. Hence, no Galois connection can exist for this lattice. Observe also that, in the abstract, we have $(\geq 0) \sqcap (\leq 0) = \bot$, which is not a sound abstraction of $\gamma((\geq 0)) \cap \gamma((\leq 0)) = \{0\}$.

On the other hand, if we complete the lattice with a dedicated 0 abstract element with the natural interpretation $\gamma(0) = \{0\}$, then we obtain the lattice in Fig. 2.9.(b), which enjoys a Galois connection. We simply define:

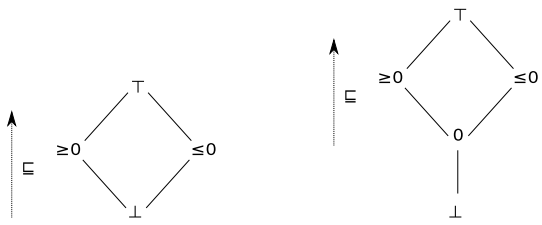
$$\alpha(X) = \begin{cases} \bot & \textit{if } X = \emptyset \\ 0 & \textit{if } X = \{0\} \\ (\geq 0) & \textit{otherwise if } X \subseteq \mathbb{N} \\ (\leq 0) & \textit{otherwise if } X \subseteq -\mathbb{N} \\ \top & \textit{otherwise} \end{cases}$$

♦

2.3.4 Galois Embeddings

We saw, for instance in Ex. 2.10, that $\gamma \circ \alpha$ is extensive and generally not the identity, as abstracting looses precision. Concretizing, however, does not loose precision, so we would expect $\alpha \circ \gamma$ to be the identity. When this is the case, we have a Galois embedding:

Definition 2.13. *A Galois connection $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ is a* Galois embedding *if any of the following, equivalent properties hold:*



(a) (b)

Figure 2.9: Sign abstraction lattices: (a) does not enjoy a Galois connection, while (b) does.

1. *α is surjective:* $\forall a \in A\colon \exists c \in C\colon \alpha(c) = a$;

2. *γ is injective:* $\forall a, a' \in A\colon \gamma(a) = \gamma(a') \implies a = a'$;

3. $\alpha \circ \gamma = id$.

We denote a Galois embedding as $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$. ■

Properties 1 and 2 state that every abstract element is useful: there is no redundancy as every abstract element abstracts a different concrete element. Property 3 allows us to view the abstract domain A as isomorphic to a subset of the concrete domain C. This is depicted in Fig. 2.8.(b).

Example 2.13 (Galois embedding)**.**

1. *The interval Galois connection from Ex. 2.10 is actually an embedding.*

2. *Consider the following, alternate interval abstraction:*

$$\{\, [a, b] \mid a \in \mathbb{Z} \cup \{\, +\infty, -\infty \,\},\, b \in \mathbb{Z} \cup \{\, +\infty, -\infty \,\} \,\}$$

with order $[a, b] \sqsubseteq [c, d] \overset{\text{def}}{\iff} (a \geq c) \wedge (b \leq d)$ *and natural concretization:* $\gamma([a, b]) \overset{\text{def}}{=} \{\, x \in \mathbb{Z} \mid a \leq x \leq b \,\}$. *Then, we have* $\gamma([a, b]) = \emptyset$ *for any pair* $[a, b]$ *such that* $a > b$*: the concrete element* $\emptyset$ *has several representations in the abstract.*

We can construct an abstraction function α *as follows:* $\alpha(S) \overset{\text{def}}{=} [\min S, \max S]$*, so that* (α, γ) *is a Galois connection. However, it is not a Galois embedding. The abstraction* α *differs from that of the interval abstraction from Ex. 2.10 only for the empty set, where* $\alpha(\emptyset) = [+\infty, -\infty]$*. Note that, in order for the set of intervals* $\{\, [a, b] \mid a > b \,\}$ *representing the empty set to have a least element, we had to allow* $+\infty$ *as lower bound and* $-\infty$ *as upper bound; with only a slight extension of notations, we state that* $\min \emptyset = +\infty$ *and* $\max \emptyset = -\infty$. ♦

2.3.5 Derived Galois Connections

Similarly to the way we can derive new ordered structures from basic ones (Def. 2.7), we can derive new Galois connections by applying generic constructions to existing ones:

Definition 2.14 (Deriving Galois connections)**.** *Assume we have a Galois connection* $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$*, we can derive new Galois connections as follows:*

1. *Duality:* $(A, \sqsupseteq) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} (C, \geq)$.

2. *Point-wise lifting:* $(S \to C, \dot{\leq}) \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} (S \to A, \dot{\sqsubseteq})$ *where*

$$\begin{array}{lcl} f \dot{\leq} f' & \overset{\text{def}}{\iff} & \forall s \in S\colon f(s) \leq f'(s) \\ f \dot{\sqsubseteq} f' & \overset{\text{def}}{\iff} & \forall s \in S\colon f(s) \sqsubseteq f'(s) \\ \dot{\alpha}(f) & \overset{\text{def}}{=} & \lambda s \in S.\, \alpha(f(s)) \\ \dot{\gamma}(f) & \overset{\text{def}}{=} & \lambda s \in S.\, \gamma(f(s)) \end{array}$$

   and S *is an arbitrary set.*

3. *Composition:* $(X_1, \sqsubseteq_1) \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} (X_3, \sqsubseteq_3)$,
   given $(X_1, \sqsubseteq_1) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (X_2, \sqsubseteq_2) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (X_3, \sqsubseteq_3)$. ■

The second point is useful to lift Galois connections on single variables to Galois connections on multi-variable program states. We will use it extensively in Chap. 4 to define non-relational domains.

The third point is useful to build complex abstractions by composing several small abstraction steps, encouraging a modular view of abstractions.

2.3.6 Operator Approximation

In order to construct a static analysis by abstraction, any operator that operates in the concrete world must be replaced with a similar operator but operating within the abstract world. Even with only a concretization function and no Galois connection, the notion of sound and exact abstraction (Def. 2.11) carries naturally from domain elements to domain operators:

Definition 2.15 (Sound and exact operator abstraction)**.** *Given a concretization* γ *from an abstract domain* $(A, \sqsubseteq)$ *to a concrete domain* $(C, \leq)$*, a concrete operator* $f : C \to C$*, and an abstract operator* $g : A \to A$*:*

1. g *is a* sound abstraction *of* f *if* $\forall a \in A\colon f(\gamma(a)) \leq \gamma(g(a))$*;*

2. g *is an* exact abstraction *of* f *if* $f \circ \gamma = \gamma \circ g$. ■

An exact abstraction is always sound. For a sound abstraction to be exact, the concrete image of any abstract-representable element must also be abstract-representable (e.g., the image of an interval is an interval), which is quite rare.

When we have a Galois connection, we can additionally transport the notion of best abstraction (Thm. 2.5) to operators:

Definition 2.16 (Best operator abstraction)**.** *Given a Galois connection* $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *and a concrete operator* $f : C \to C$*, the* best abstraction *of* f *is given by* $\alpha \circ f \circ \gamma$*.* ■

Indeed, note that the soundness definition, $f(\gamma(a)) \leq \gamma(g(a))$ from Def. 2.15.1, can be written equivalently, given the definition of Galois connections (Def. 2.5), as $\forall a \in A\colon \alpha(f(\gamma(a))) \sqsubseteq g(a)$. Hence, a sound abstraction g of f is any operator that is greater than $\alpha \circ f \circ \gamma$. This means that $\alpha \circ f \circ \gamma$ is thus the best abstraction, i.e., the smallest sound abstraction, of f.

Example 2.14 (Operator abstraction)**.** *Consider the following concrete operator on integer sets:* $f \stackrel{\text{def}}{=} \lambda X.\, \{\, x + 1 \mid x \in X \,\}$*. Then* $g_1 \stackrel{\text{def}}{=} \lambda[a, b].\, [-\infty, +\infty]$ *is a sound abstraction of* f *in the interval domain, while* $g_2 \stackrel{\text{def}}{=} \lambda[a, b].\, [a+1, b+1]$ *is a sound and exact abstraction of* f*.*

Consider now $f \stackrel{\text{def}}{=} \lambda X.\, \{\, 2x \mid x \in X \,\}$*. Then* $g \stackrel{\text{def}}{=} \lambda[a, b].\, [2a, 2b]$ *is the best abstraction of* f *in the interval domain, but it is not exact. Indeed, the concrete image of an interval is not necessarily an interval. Consider, for instance, that* $f(\{0, 1\}) = \{0, 2\}$*, while* $g([0, 1]) = [0, 2]$*, which contains the spurious value* 1*.* ♦

Definition 2.16 is a powerful tool as it allows deriving the abstract semantics systematically from the concrete one and a Galois connection. Note, however, that this is a constructive mathematical definition that cannot generally be implemented as is, as neither its components α, f, γ are likely to be computable. Rather, it is the designer's responsibility to turn this mathematical definition into an algorithm. Sometimes, it is not easy to derive such an algorithm, or the algorithm might not be sufficiently efficient. Finally, there is always the case of abstract domains without a Galois connection (Ex. 2.11). In those cases, the designer has to rely on intuition to invent a suitable abstract operator, several choices being possibles, and then prove its soundness through Def. 2.15, without relying on Def. 2.16.

2.3.7 Operator Composition

It is best to keep our abstractions as modular and composable as possible. As we will see in the next chapter, the semantics of a program is generally obtained by composing atomic semantic functions from a limited library, corresponding to basic language operations. This lends itself well to a modular abstraction scheme, where we design abstract operators only for this alphabet of basic operations, and compose the abstract operators following the same rules as in the concrete semantics. This is made possible because sound and exact abstractions compose:

Theorem 2.6 (Operator composition)**.** *Assume that $f, f' : C \to C$ are concrete operators, and $g, g' : A \to A$ are abstract operators:*

1. *if g and g' are sound abstractions of respectively f and f', and f is monotonic, then $g \circ g'$ is a sound abstraction of $f \circ f'$;*

2. *if g and g' are exact abstractions of respectively f and f', then $g \circ g'$ is an exact abstraction of $f \circ f'$.* ■

The technical condition requiring that f is monotonic for sound abstractions to compose is not very severe: it concerns the concrete world, which, unlike the abstract world, is monotonic, as we will see in the next chapter.

Naturally, as best abstractions are sound, then, if g and g' are the best abstractions respectively of f and f', then $g \circ g'$ is a sound abstraction of $f \circ f'$. However, it is not necessarily the best abstraction of $f \circ f'$, as shown below:

Example 2.15 (Non-composability of optimality)**.** *Consider, in the concrete domain of integer sets, the operators $f \stackrel{\text{def}}{=} \lambda X.\, \{\, x \in X \mid x \leq 1 \,\}$ and $f' \stackrel{\text{def}}{=} \lambda X.\, \{\, 2x \mid x \in X \,\}$. Consider now their best abstractions g and g' in the interval domain: $g \stackrel{\text{def}}{=} \lambda[a, b].\, [a, \min(b, 1)]$ and $g' \stackrel{\text{def}}{=} \lambda[a, b].\, [2a, 2b]$. Then, $g \circ g'$ is not the best abstraction of $f \circ f'$ as $(g \circ g')([0, 1]) = [0, 1]$ while $(\alpha \circ f \circ f' \circ \gamma)([0, 1]) = [0, 0]$.*

More, generally, composing best abstractions gives $\alpha \circ f \circ \gamma \circ \alpha \circ f' \circ \gamma$, while the best abstraction of the composition would be $\alpha \circ f \circ f' \circ \gamma$. The former performs an additional trip trough the abstract world, $\gamma \circ \alpha$, between f and f', which is a source of imprecision and the reason the composition is no longer optimal. ♦

As a side-note, although composing two optimal abstractions does not necessarily result in an optimal abstraction, applying an exact abstraction and then an optimal abstraction results in an optimal abstraction.

The lack of general composability for the notion of best abstraction has rather important practical ramifications. The precision of an analysis depends on the granularity of the decomposition of the program semantics into atomic operations abstracted independently. The finer the decomposition, the larger the risk of some imprecision appearing. A coarser decomposition allows, on the other hand, optimal abstractions for larger code blocks. It may not be practicable, however, as the number of possible blocks grows in a combinatorial fashion with block size. There is generally a trade-off to achieve. It may also be worth keeping the semantics small and modular, and turn instead to more expressive abstract domains to alleviate the loss of precision (i.e., increasing the probability of exact abstractions for atomic instructions).

To conclude, it is important to keep in mind that, despite our best efforts, the analysis results will seldom be the most precise information representable in the abstract, even if it employs solely best abstractions. For instance, an interval analysis will seldom output the tightest variable bounds possible.

2.3.8 Fixpoint Approximation

Critical parts of the semantics of a program are defined as least fixpoints $\mathrm{lfp}\, f$ of some monotonic or continuous operator $f : C \to C$ in the concrete domain $(C, \leq)$. In order to abstract $\mathrm{lfp}\, f$ in an abstract domain $(A, \sqsubseteq)$, a natural idea is to start with a sound abstraction $g : A \to A$ of f. Then, there exists many variants of fixpoint approximation theorems that provide guidelines on how to use g to soundly approximate $\mathrm{lfp}\, f$. We mention here two of them that will be useful in the rest of the tutorial.

Kleene fixpoint transfer. A first, natural idea is to mimic the fixpoint computation, with g instead of f. For instance, relying on the constructive definition of $\mathrm{lfp}\, f$ as the limit of an iteration sequence from Kleene's theorem (Thm. 2.2), we get:

Theorem 2.7 (Kleenian fixpoint approximation)**.** *If $f : C \to C$ is continuous in a CPO $(C, \leq, \vee, \bot)$, and $g : A \to A$ is a sound — not necessarily monotonic — abstraction of f in a poset abstract domain $(A, \sqsubseteq, \bot')$, and the sequence $\{\, g^i(\bot') \mid i \in \mathbb{N} \,\}$ has a limit x in A, then it is a sound approximation of $\mathrm{lfp}\, f$, i.e., $\mathrm{lfp}\, f \leq \gamma(x)$.* ■

The theorem relies on the fact that, at each step i, $g^i(\bot')$ is a sound approximation of $f^i(\bot)$, and passes to the limit. In particular, if $g^i(\bot')$ stabilizes after some finite step N (e.g., when there are no strictly increasing infinite chains in A) then $g^N(\bot')$ over-approximates our concrete fixpoint. Note that we only guarantee soundness, not optimality, even if g is the best abstraction of f. Indeed, we compose g many times, and we know that the composition of best abstractions is not necessarily the best abstraction of the composition.

Tarski fixpoint transfer. A second theorem we will use is based on Tarski's characterization of $\mathrm{lfp}\, f$ as the smallest postfixpoint (Thm. 2.1) and the observation that any abstract postfixpoint of a sound abstraction represents, through γ, a concrete postfixpoint, thus, an overapproximation of $\mathrm{lfp}\, f$.

Theorem 2.8 (Tarskian fixpoint approximation)**.** *Given a complete lattice concrete domain $(C, \leq, \vee, \wedge, \bot, \top)$, a monotonic concrete function $f : C \to C$, and a sound — not necessarily monotonic — abstraction $g : A \to A$ of f in a poset abstract domain $(A, \sqsubseteq)$, then any postfixpoint a of g, i.e., such that $g(a) \sqsubseteq a$, is a sound abstraction of $\mathrm{lfp}\, f$, i.e., $\mathrm{lfp}\, f \leq \gamma(a)$.* ■

The theorem does not state how a postfixpoint of g can be computed in the abstract. While it is conceivable to compute a least fixpoint in the abstract world, as for Thm. 2.7, the theorem can be applied in the useful case where abstract fixpoints are hard to compute, or do not even exist at all. Indeed, we will see important examples where, while fixpoints exist in the concrete, they are not guaranteed to exist in the abstract, either because the abstract domain is not a complete lattice nor a CPO (such as the affine inequalities domain) or the abstract functions are not monotonic. It is important to keep in mind that we are only trying to compute an abstraction of a concrete fixpoint, and do not require computing an abstract fixpoint: this would be too limiting. In these cases, we may use

fixpoint acceleration techniques, as described below, to get a postfixpoint, even in the absence of fixpoints.

2.3.9 Fixpoint Acceleration

In order to solve the convergence problem in abstract domains, Cousot and Cousot [1977] introduced a specific binary operator, the widening $\triangledown$:

Definition 2.17. *A binary operator* $\triangledown : A \times A \to A$ *is a* widening operator *in an abstract domain* $(A, \sqsubseteq)$ *if:*

1. *it computes upper bounds:* $\forall x, y \in A\colon x \sqsubseteq x \triangledown y$ *and* $y \sqsubseteq x \triangledown y$*;*
2. *and it enforces convergence: for any sequence* $(y^i)_{i\in\mathbb{N}}$ *in* A*, the sequence* $(x^i)_{i\in\mathbb{N}}$ *computed as* $x^0 \stackrel{\text{def}}{=} y^0$*,* $x^{i+1} \stackrel{\text{def}}{=} x^i \triangledown y^{i+1}$ *stabilizes in finite time:* $\exists k \geq 0\colon x^{k+1} = x^k$*.*

■

The first point allows us to construct increasing sequences by iterating abstract operators that are not necessarily monotonic. Another consequence of the first point is that the result $x \triangledown y$ is a sound approximation of the concrete join $\gamma(a) \vee \gamma(b)$.

The second point ensures that all increasing sequences with widening are finite, even if the abstract domain A has infinite strictly increasing chains.

To illustrate this definition, we propose a simple and naive widening that can be applied to an arbitrary abstract domain with a greatest element $\top$. It consists simply in putting an unstable iterate to $\top$:

Example 2.16 (Naive widening)**.**

$$x \triangledown y \stackrel{\text{def}}{=} \begin{cases} x & \textit{if } y \sqsubseteq x \\ \top & \textit{otherwise} \end{cases}$$

♦

We will see many more sophisticated widenings in Chaps. 4–5, when presenting numeric abstract domains in details.

The following theorem states that widenings can indeed be used to approximate least fixpoints in the abstract:

Theorem 2.9. *If* f *is a monotonic operator in a complete concrete lattice and* g *is a sound abstraction of* f*, then the following iteration:*

$$\begin{array}{lcl} x^0 & \stackrel{\text{def}}{=} & \bot \\ x^{i+1} & \stackrel{\text{def}}{=} & x^i \triangledown g(x^i) \end{array} \tag{2.6}$$

converges in finite time, and its limit x *is a sound abstraction of the least fixpoint* $\operatorname{lfp} f$*:* $\operatorname{lfp} f \leq \gamma(x)$*.* ■

The convergence property comes from Def. 2.17.2. Note also that, because of Def. 2.17.1, the iterates x^i, $i \in \mathbb{N}$ actually form a chain, even if g is not monotonic. The soundness comes from the fixpoint approximation Thm. 2.8 given that, when encountering a stable value $x^{i+1} = x^i$, then $f(x^i) \sqsubseteq x^i \triangledown f(x^i) = x^{i+1} = x^i$, i.e., x^i is an abstract postfixpoint. As we will see, such an iteration is rather naive and can sometimes loose much precision. More advanced iterations techniques, and a better use of widenings will be presented in Sect. 4.7, in the context of the interval domain.

2.4 Summary

This chapter presented the mathematical bases of Abstract Interpretation. To sum up, semantic definitions are stated as operators in ordered structures, such as partial orders, CPO, lattices, or complete lattices. Additional hypotheses, such as monotonic operators on lattices or continuous operators on CPO, are necessary to ensure the existence of least fixpoints, which appear in the semantics of loops in the concrete semantics. We discussed the connections between a concrete and an abstract semantics, using two methods: a concretization-only framework, which is very general but can only be used to check the soundness of hand-crafted operators, and a stronger framework, based on Galois connections, which adds the notion of best abstractions and a closed mathematical formula to construct them from the concrete semantics. Finally, we discussed the problem of abstracting concrete least fixpoints, even in the general case where the corresponding abstract operator does not have any fixpoint. The next chapter will apply these results to define the concrete and sound abstract semantics of a concrete, if simple, programming language.

2.5 Bibliographic Notes

Partial orders are pervasive in Computer Science and, particularly, in program semantics, through domain theory introduced by Scott and Strachey [1971]. An influential and reference work on the theory of partial orders and lattices, from an algebraic point of view, is the book by Birkhoff [1967]. The key fixpoint theorem in lattices was introduced by Tarski [1955]. Additional fixpoint theorems, based on the notion of (finite, countable, or transfinite) iteration were introduced by Cousot and Cousot [1979b]. Galois connections stem from Galois theory, that studies the solvability of polynomial equations; they have applications in several branches of Mathematics, but their use in Computer Science seems restricted to Abstract Interpretation.

All the main elements of Abstract Interpretation, including the use of Galois connections and the invention of widenings, are introduced early, by Cousot and Cousot [1976], although [Cousot and Cousot, 1977] gives a more complete presentation. There exist closely related presentations of Abstract Interpretation that replace Galois connections with upper closure operators, complete join congruences relations, principal ideals, or Moore families; those are described by Cousot and Cousot [1979a]. Such presentations identify an abstract domain with the set of concrete elements is represents, which we avoided here, but makes

it possible to manipulate more easily abstract domains as first class objects and derive domain constructions. We will see a classic application in Sect. 6.1, while Cortesi et al. [1997] provide additional constructions. The distributivity of abstract domains was discussed by Cousot and Cousot [1979a]. Finally, Schmidt [2009] provided an interesting connection between Abstract Interpretation and topology.

In another direction, Cousot and Cousot [1992b] introduce the revised Abstract Interpretation framework, which can dispense from Galois connections and allows a concretization-only presentation, with revised notions of soundness and widenings. We based on presentation on the revised framework as some of our domains do not feature a Galois connection, although Cousot and Cousot [1992b] is even more general and more relaxed.

Cousot and Cousot [1992a] discuss the widening operator, one of the less understood aspect of Abstract Interpretation, and justify its usefulness by showing that infinite height domains with widenings are more powerful than finite height domains. The theory of widenings is further investigated by Cortesi and Zanioli [2011].

Chapter 3

Language and Semantics

In this chapter, we introduce the target language of our static analysis. It is a very simple language, with limited constructions and idealized semantics, so that we can fully present the design of various static analyses and compare them. It is also a purely numeric language, as we focus on numeric invariants, with idealized, mathematical numbers. Nevertheless, the method we develop in this chapter and the followings can be applied to realistic languages, as seen for example in the Astrée static analyzer for C [Bertrane et al., 2015]. Note also that, despite its simplicity, the language allows unbounded program executions and unbounded numeric values for variables, so that the problem of inferring invariants is actually undecidable, following Rice [1953]'s result.

We first present the syntax of our language. We then present its concrete semantics, that is, the most precise mathematical expression of the program behaviors. Actually, the term "precise" is relative: as we focus on numeric invariants, we choose a concrete semantics that expresses numeric invariants, and numeric invariants only. It is thus less expressive than other kinds of semantics that could, for instance, discuss about program termination, efficiency, security, etc. It is the "most precise" semantics in that, if we could compute it, it would answer perfectly our original question and infer the tightest program invariants. But we cannot, and so, we will require further abstractions, losing precision and completeness to gain decidability. This concrete semantics, which is tailored to the problem at hand and not (yet) adapted to the finitary computational power of computers, is called the *collecting concrete semantics*. Finally, this chapter will present a generic static analysis parameterized by an abstract domain, which is a first step towards automatic but approximate invariant inference. The analysis will be completed in the next chapters by providing abstract domain instances and iterations strategies.

The static analysis we develop is formally sound by construction, as we leverage the Abstract Interpretation theory presented in the previous chapter. However, the proof of soundness is only relative to the concrete semantics: if the concrete semantics does not match the actual behaviors of the programs, then the analysis may well be incorrect! It is thus critical to state the concrete semantics in the simplest, clearest, unambiguous way, so that all parties can agree on its validity, before moving on to designing static analyses.

$$
\begin{array}{rcll}
\mathit{stat} & ::= & V \leftarrow \mathit{expr} & \textit{(assignment, } V \in \mathbb{V}\textit{)} \\
 & | & \mathit{stat}; \mathit{stat} & \textit{(sequence)} \\
 & | & \textbf{if } \mathit{cond} \textbf{ then } \mathit{stat} \textbf{ else } \mathit{stat} \textbf{ endif} & \textit{(conditional)} \\
 & | & \textbf{while } \mathit{cond} \textbf{ do } \mathit{stat} \textbf{ done} & \textit{(loop)} \\
 & | & \textbf{skip} & \textit{(no-op)} \\
 & | & \textbf{assert } \mathit{cond} & \textit{(assertion)}
\end{array}
$$

Figure 3.1: Syntax of programs.

$$
\begin{array}{rcll}
\mathit{expr} & ::= & V & \textit{(variable, } V \in \mathbb{V}\textit{)} \\
 & | & c & \textit{(numeric constant, } c \in \mathbb{I}\textit{)} \\
 & | & -\mathit{expr} & \textit{(negation)} \\
 & | & \mathit{expr} \diamond \mathit{expr} & \textit{(binary operator, } \diamond \in \{+, -, \times, /\}\textit{)} \\
 & | & [c_1, c_2] & \textit{(input, } c_1, c_2 \in \mathbb{I} \cup \{-\infty, +\infty\}\textit{)}
\end{array}
$$

Figure 3.2: Syntax of (numeric) expressions.

In practice, this is not an easy task, as actual programming languages are complex and often underspecified, and the normative documents, such as the C language specification [ISO/IEC JTC1/SC22/WG14 working group, 2007], are written in a natural, informal language: English.

Once the concrete semantics is agreed upon, various flavors of sound static analyses can be safely developed using the Abstract Interpretation theory, and the analysis user does not need to know the intricate details of the abstractions we currently use to trust the validity of the analysis results — such information is useful, however, for the user to understand the precision and limits of the analysis.

3.1 Syntax

Figures 3.1–3.3 present the syntax of our target language as a grammar in BNF form.

We assume a fixed, finite set $\mathbb{V}$ of program variables, with numeric values. We denote by $\mathbb{I}$ the domain of variables, and assume that $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. Some of our abstract domains will only work for integer-valued variables, others only for real-valued ones, some for both but, in all cases, we assume here, for simplicity, that we use some perfect mathematical version of numeric sets and not machine-integers nor floating-point numbers used actually in most computer languages.

$$
\begin{array}{rcll}
\mathit{cond} & ::= & \mathit{expr} \bowtie \mathit{expr} & \textit{(comparison, } \bowtie \in \{\leq, \geq, <, >, =, \neq\}\textit{)} \\
 & | & b & \textit{(boolean constant, } b \in \mathbb{B}\textit{)} \\
 & | & \neg \mathit{cond} & \textit{(logic negation)} \\
 & | & \mathit{cond} \wedge \mathit{cond} & \textit{(logic and)} \\
 & | & \mathit{cond} \vee \mathit{cond} & \textit{(logic or)}
\end{array}
$$

Figure 3.3: Syntax of (boolean) conditions.

The program statements *stat* are presented in Fig. 3.1. They include assignments "$V \leftarrow$ *expr*"; instruction sequencing with a semicolon; conditionals "**if** *cond* **then** *stat* **else** *stat* **endif**"; while loops "**while** *cond* **do** *stat* **done**"; no-ops "**skip**" — used, e.g., to fill-in unused branches of conditionals — and assertions "**assert** *cond*", which stop the program when a given condition does not hold.

Numeric expressions *expr* are presented in Fig. 3.2: they include variable use $V \in \mathbb{V}$, usual unary ($-$) and binary ($+$, $-$, $\times$, $/$) arithmetic operators, and constants $c \in \mathbb{I}$. Additionally, expressions feature a non-standard construction: $[c_1, c_2]$, where c_1 and c_2 are constants in $\mathbb{I}$ or an infinity. Each evaluation of $[c_1, c_2]$ returns a value within the bounds, independent from the previous evaluations. It is intended to denote non-deterministic inputs, out of the control of the program. We assume that $[c_1, c_2]$ is not empty, that is, $c_1 \in \mathbb{I} \cup \{-\infty\}$, $c_2 \in \mathbb{I} \cup \{+\infty\}$, and $c_1 \leq c_2$.

Finally, Fig. 3.3 presents the syntax of boolean conditions *cond*, that appear in conditionals, assertions, and loops. These include the boolean constants in $\mathbb{B} \stackrel{\text{def}}{=} \{\text{true}, \text{false}\}$, unary ($\neg$) and binary ($\wedge$, $\vee$) logic operators, as well as comparisons of two numeric expressions ($=$, $\neq$, $\leq$, $\geq$, $<$, $>$). Note that variables, which are numeric, can only appear in numeric expressions *expr*, and not in boolean conditions *cond*.

Non-determinism. Due to the $[c_1, c_2]$ construction, the semantics of a program may be non-deterministic: different executions of the program may exhibit different behaviors. What it means for verification is that, for the program to be considered correct, the whole space of its possible executions must be proved correct. Hence, to be sound, our semantics will consider, at each $[c_1, c_2]$ expression, the set of all outcomes, and return a set of program states, even when starting in a well-defined state. Non-determinism has many applications:

- We can analyze programs that interact with an outside environment. One example is a program accessing a file or a network: by modelling write accesses as no-ops and read-accesses as non-deterministic values in the range of the data type, we can prove that a program is correct whatever the behavior of the environment. Another example is the analysis of embedded control-command systems, such as found in planes or cars, which fetch physical values through sensors. Additional hypotheses about sensor ranges, if trusted, can be used to refine the bounds of $[c_1, c_2]$ expressions.

- We can analyze a family of programs parameterized by the value of some variable N. Instead of performing one analysis for each value of N, we perform a single analysis after initializing N with a non-deterministic expression, hence factoring the analyses. It is thus possible to analyze efficiently, in a single step, a program family with a large, or even unbounded range of values of N.

- We can model floating-point rounding errors using small non-deterministic intervals. For instance, a floating-point operation $e_1 + e_2$ is modelled as $[1-\epsilon, 1+\epsilon] \times (e_1 + e_2) + [-\varepsilon, \varepsilon]$, where ϵ and ε account respectively for a relative and an absolute rounding error. The latter expression can then be fed to an analyzer that assumes a real

semantics (such as the one we present here) to obtain a sound over-approximation of the floating-point results.

- We can use $[c_1, c_2]$ expressions to model parts of the program that are omitted — e.g., because they are in libraries, their source code is not available, or they are too complex. For instance, a program using the sin function can be verified assuming only that sin returns a value in $[-1, 1]$, without the need to specify which value is computed, nor how.
- Finally, note that non-deterministic control flow can also be modeled, using $[c_1, c_2]$, by writing **if** $[0, 1] = 0$ **then** *stat* **else** *stat* **endif**. This can provide a way, for instance, to model non-deterministic interrupts.

3.2 Atomic Statement Semantics

There are traditionally two different ways of presenting program semantics in Abstract Interpretation, which differ in the way compound constructions, such as conditionals and loops, are handled: either by induction on the syntax or through an equation system reflecting the control-flow graph. They lead to slightly different static analysis algorithms, with their own strengths and weaknesses, so, we will present both in the next sections. Nevertheless, they coincide on their treatment of expressions, conditions, and atomic statements (such as assignments and assertions). We present here these common parts.

3.2.1 Concrete Domain

We are interested in inferring program invariants, i.e., properties of the memory state a program can be in at each program location. Hence, our concrete collecting semantics operates on a domain of *sets of memory states*. A memory state, denoted as $\rho \in \mathcal{E}$, is a function mapping each variable to its value: $\mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{I}$. The concrete domain is thus the powerset complete lattice:

$$(\mathcal{D}, \subseteq, \cup, \cap, \emptyset, \mathcal{E}) \text{ where } \mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E}) \tag{3.1}$$

3.2.2 Expression Semantics

Given a single memory state $\rho \in \mathcal{E}$, an expression $e \in \mathit{expr}$ can evaluate to one or more (due to non-determinism) values in $\mathbb{I}$. The evaluation function, denoted as $\mathbb{E}[\![e]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{I})$, is defined in Fig. 3.4 by induction on the syntax. For instance, a binary operator such as $+$ evaluates its sub-expressions to get sets of values in $\mathcal{P}(\mathbb{I})$, and returns all the possible sums of these values. For divisions, we take care to avoid dividing by zero. As a consequence, it is possible for $\mathbb{E}[\![e]\!]\rho$ to return an empty set — denoting a definite division by zero. When it appears in the evaluation of a sub-expression, the empty set is naturally propagated, and the whole expression evaluates to $\emptyset$.

Example 3.1 (Expression evaluation)**.**

$$
\begin{array}{lcl}
\mathbb{E}[\![\, expr \,]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{I}) & & \\
\hline
\mathbb{E}[\![\, V \,]\!]\rho & \stackrel{\text{def}}{=} & \{\rho(V)\} \\
\mathbb{E}[\![\, c \,]\!]\rho & \stackrel{\text{def}}{=} & \{c\} \\
\mathbb{E}[\![\, [c_1, c_2] \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x \in \mathbb{I} \mid c_1 \leq x \leq c_2 \,\} \\
\mathbb{E}[\![\, -e \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, -v \mid v \in \mathbb{E}[\![\, e \,]\!]\rho \,\} \\
\mathbb{E}[\![\, e_1 + e_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, v_1 + v_2 \mid v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \,\} \\
\mathbb{E}[\![\, e_1 - e_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, v_1 - v_2 \mid v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \,\} \\
\mathbb{E}[\![\, e_1 \times e_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, v_1 \times v_2 \mid v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \,\} \\
\mathbb{E}[\![\, e_1 / e_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, v_1 / v_2 \mid v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho,\ v_2 \neq 0 \,\}
\end{array}
$$

Figure 3.4: Semantic of expressions.

- $\mathbb{E}[\![\, 1 + (2/0) \,]\!]\rho = \emptyset$ *as* $\mathbb{E}[\![\, 2/0 \,]\!]\rho = \emptyset$.
- $\mathbb{E}[\![\, [-1, 1]/0 \,]\!]\rho = \emptyset$ *as all non-deterministic choices result in a division by zero.*
- $\mathbb{E}[\![\, 1/[-1, 1] \,]\!]\rho = \{-1, 1\}$ *as, although the expression evaluation performs a division by zero when* $[-1, 1]$ *evaluates to 0, for other values, it returns a result: either* -1 *or* 1. ♦

3.2.3 Condition Semantics

Conditions are used to filter out memory states, keeping only those that satisfy the condition. As our semantics ultimately outputs sets of memory states, a natural semantics for conditions is an operator $\mathbb{C}[\![\, cond \,]\!] : \mathcal{D} \to \mathcal{D}$ that takes as input a set of memory states, and outputs the subset of states that pass the test. This semantics is presented in Fig. 3.5. To simplify, we assume that all negations $\neg$ have been removed using DeMorgan's laws and usual arithmetic laws: $\neg(a \vee b) \to (\neg a) \wedge (\neg b)$, $\neg(e_1 > e_2) \to (e_1 \leq e_2)$, etc.

Note that, for an arithmetic comparison such as $e_1 = e_2$ to hold in a given memory state ρ, given that e_1 and e_2 can evaluate to several values in ρ, we only require that at least one possible value of $\mathbb{E}[\![\, e_1 \,]\!]\rho$ equals one possible value of $\mathbb{E}[\![\, e_2 \,]\!]\rho$. As a consequence, it is possible for one state ρ to satisfy both a condition and its negation: $\mathbb{C}[\![\, [0, 1] = 0 \,]\!]\{\rho\} = \mathbb{C}[\![\, \neg([0, 1] = 0) \,]\!]\{\rho\} = \{\rho\}$.

In case an expression in a condition evaluates to $\emptyset$ due to a division by zero, the condition also evaluates to $\emptyset$ — unless the condition is combined with a logic or to a condition that does not return $\emptyset$.

Note that $\mathbb{C}[\![\, c \,]\!]$ has the following useful algebraic property: $\mathbb{C}[\![\, c \,]\!](\cup_{i \in I} R_i) = \cup_{i \in I} (\mathbb{C}[\![\, c \,]\!]R_i)$ for arbitrary families $(R_i)_{i \in I}$ of state sets, i.e., it is a *join morphism*. In fact, for such a function, its result on an arbitrary set can be derived by joining the result on each independent value: $\mathbb{C}[\![\, c \,]\!]R = \cup \{\, \mathbb{C}[\![\, c \,]\!]\{\rho\} \mid \rho \in R \,\}$. Moreover, a join morphism is necessarily monotonic and continuous.

Finally, the function is also reductive: $\mathbb{C}[\![\, c \,]\!]R \subseteq R$, and idempotent: $\mathbb{C}[\![\, c \,]\!] \circ \mathbb{C}[\![\, c \,]\!] = \mathbb{C}[\![\, c \,]\!]$.

$$
\begin{array}{lcl}
\underline{\mathbb{C}[\![\, cond \,]\!] : \mathcal{D} \to \mathcal{D}} & & \\
\mathbb{C}[\![\, \text{true} \,]\!]R & \stackrel{\text{def}}{=} & R \\
\mathbb{C}[\![\, \text{false} \,]\!]R & \stackrel{\text{def}}{=} & \emptyset \\
\mathbb{C}[\![\, c_1 \wedge c_2 \,]\!]R & \stackrel{\text{def}}{=} & \mathbb{C}[\![\, c_1 \,]\!]R \cap \mathbb{C}[\![\, c_2 \,]\!]R \\
\mathbb{C}[\![\, c_1 \vee c_2 \,]\!]R & \stackrel{\text{def}}{=} & \mathbb{C}[\![\, c_1 \,]\!]R \cup \mathbb{C}[\![\, c_2 \,]\!]R \\
\mathbb{C}[\![\, e_1 = e_2 \,]\!]R & \stackrel{\text{def}}{=} & \{\, \rho \in R \mid \exists v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \colon v_1 = v_2 \,\} \\
\mathbb{C}[\![\, e_1 \neq e_2 \,]\!]R & \stackrel{\text{def}}{=} & \{\, \rho \in R \mid \exists v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \colon v_1 \neq v_2 \,\} \\
\mathbb{C}[\![\, e_1 < e_2 \,]\!]R & \stackrel{\text{def}}{=} & \{\, \rho \in R \mid \exists v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \colon v_1 < v_2 \,\} \\
\mathbb{C}[\![\, e_1 > e_2 \,]\!]R & \stackrel{\text{def}}{=} & \{\, \rho \in R \mid \exists v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \colon v_1 > v_2 \,\} \\
\mathbb{C}[\![\, e_1 \leq e_2 \,]\!]R & \stackrel{\text{def}}{=} & \{\, \rho \in R \mid \exists v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \colon v_1 \leq v_2 \,\} \\
\mathbb{C}[\![\, e_1 \geq e_2 \,]\!]R & \stackrel{\text{def}}{=} & \{\, \rho \in R \mid \exists v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \colon v_1 \geq v_2 \,\}
\end{array}
$$

Figure 3.5: Semantic of conditions.

Example 3.2 (Condition semantics)**.**

- $\mathbb{C}[\![\, 1 = (1/0) \,]\!]R = \emptyset$ *due to the division by zero.*
- $\mathbb{C}[\![\, (1 = 1/0) \vee \mathit{true} \,]\!]R = R$ *despite the division by zero, as all the states pass the condition true. Note that, unlike the "shortcut semantics" of a language such as C, both boolean expressions of a* $\vee$ *operator get evaluated in our language, even if the first one evaluates to true.* ♦

3.2.4 Atomic Statements

Simple atomic statements, which operate in one step and do not contain any control nor sub-statement, include assignments, assertions, and no-ops (skip). The effect of a statement is to change a memory state into another one or, in case of non-determinism, one of several possible ones. A natural domain for the semantic function $\mathbb{S}[\![\, s \,]\!]$ of a statement s is thus $\mathcal{E} \to \mathcal{D}$. However, statements are meant to be composed (e.g., by sequence) and it is easier to compose functions that have the same domain as codomain — we can then use the regular function composition $\circ$. So, we extend the semantic function naturally to live in $\mathcal{D} \to \mathcal{D}$, as a join morphism: $\mathbb{S}[\![\, s \,]\!]R = \cup\{\, \mathbb{S}[\![\, s \,]\!]\{\rho\} \mid \rho \in R \,\}$. Given a set of memory states before the statement, the semantic function associates the set of possible memory states after the statement. Such a semantic function is often called *transfer function*, and is, at its core, an input/output relation.

The semantic function, denoted as $\mathbb{S}[\![\, \mathit{stat} \,]\!] : \mathcal{D} \to \mathcal{D}$, is presented in Fig. 3.6. The effect of an assignment $V \leftarrow e$ on a set R of memory states is to update, in every state $\rho \in R$, the value of V with one of the possible values for e in ρ. We note that, if e evaluates to $\emptyset$ for a state $\rho \in R$, i.e., there is a division by zero, then there is no possible output corresponding to this state: the program blocks for ρ — although it can continue its execution for other states in R.

$$\underline{\mathbb{S}[\![\, stat \,]\!] : \mathcal{D} \to \mathcal{D}}$$

$$\begin{array}{lcl}
\mathbb{S}[\![\, V \leftarrow e \,]\!]R & \stackrel{\text{def}}{=} & \{\, \rho[V \mapsto v] \mid \rho \in R,\ v \in \mathbb{E}[\![\, e \,]\!]\rho \,\} \\
\mathbb{S}[\![\, \textbf{assert}\ c \,]\!]R & \stackrel{\text{def}}{=} & \mathbb{C}[\![\, c \,]\!]R \\
\mathbb{S}[\![\, \textbf{skip} \,]\!]R & \stackrel{\text{def}}{=} & R
\end{array}$$

Figure 3.6: Semantic of atomic statements.

$$\underline{\mathbb{S}[\![\, stat \,]\!] : \mathcal{D} \to \mathcal{D}}$$

$$\begin{array}{lcl}
\mathbb{S}[\![\, V \leftarrow e \,]\!]R & \stackrel{\text{def}}{=} & \{\, \rho[V \mapsto v] \mid \rho \in R,\ v \in \mathbb{E}[\![\, e \,]\!]\rho \,\} \\
\mathbb{S}[\![\, \textbf{assert}\ c \,]\!]R & \stackrel{\text{def}}{=} & \mathbb{C}[\![\, c \,]\!]R \\
\mathbb{S}[\![\, \textbf{skip} \,]\!]R & \stackrel{\text{def}}{=} & R \\
\mathbb{S}[\![\, s_1;\ s_2 \,]\!]R & \stackrel{\text{def}}{=} & (\mathbb{S}[\![\, s_2 \,]\!] \circ \mathbb{S}[\![\, s_1 \,]\!])(R)
\end{array}$$

$$\begin{array}{l}
\mathbb{S}[\![\, \textbf{if}\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{endif} \,]\!]R \stackrel{\text{def}}{=} \\
\qquad \mathbb{S}[\![\, s_1 \,]\!](\mathbb{C}[\![\, c \,]\!]R) \cup \mathbb{S}[\![\, s_2 \,]\!](\mathbb{C}[\![\, \neg c \,]\!]R)
\end{array}$$

$$\begin{array}{lcl}
\mathbb{S}[\![\, \textbf{while}\ c\ \textbf{do}\ s\ \textbf{done} \,]\!]R & \stackrel{\text{def}}{=} & \mathbb{C}[\![\, \neg c \,]\!](\mathrm{lfp}\, F) \\
\quad \text{where } F(X) \stackrel{\text{def}}{=} R \cup \mathbb{S}[\![\, s \,]\!](\mathbb{C}[\![\, c \,]\!]X) & & \
\end{array}$$

Figure 3.7: Denotational concrete semantics of programs.

The semantics of assertions blocks the program whenever the expression encounters a division by zero, but also if the condition is not satisfied, as $\mathbb{C}[\![\, c \,]\!]R$ filters out states that cannot satisfy c.

The skip statement simply returns its set of states unchanged.

3.3 Denotational-Style Semantics

One simple way to present the concrete semantics of a program is to generalize the formula we presented in Fig. 3.6 for atomic statements to compound statements. We keep a concrete function $\mathbb{S}[\![\, stat \,]\!] : \mathcal{D} \to \mathcal{D}$ and handle compound statements by induction on the syntax. This extended definition is presented in Fig. 3.7. The sequence of statements "$s_1;\ s_2$" is simply function composition while the, more complex, semantics of conditionals and loops is detailed below.

3.3.1 Conditionals

A conditional "**if** c **then** s_1 **else** s_2 **endif**" first filters the input states to keep only those that can pass the test $\mathbb{C}[\![\, c \,]\!]$, and then applies the effect $\mathbb{S}[\![\, s_1 \,]\!]$ of the **then** statement s_1 to these states. In parallel, it filters input states to keep only those that can fail the test $\mathbb{C}[\![\, \neg c \,]\!]$, and then applies the effect $\mathbb{S}[\![\, s_2 \,]\!]$ of the **else** branch s_2 to them. Finally, the function returns the join of all these states, stating that the program can continue after the conditional with the memory states from both the **then** and the **else** branch.

3.3.2 Loops

The semantics of loops "**while** c **do** s **done**" is the most complex one. It operates in two steps. We study first the first, more involved step: computing the least fixpoint $\mathrm{lfp}\, F$ of the operator $F(X) \stackrel{\text{def}}{=} R \cup \mathbb{S}[\![s]\!](\mathbb{C}[\![c]\!]X)$. This fixpoint corresponds to the notion of *tightest loop invariant*. A loop invariant, in Hoare-Floyd logic [Hoare, 1969, Floyd, 1967], is a property that is true every time the program reaches the test of the loop, either for the first time or after a loop iteration, before determining whether to exit the loop or perform another iteration.

Let us denote the set of states before the loop starts as R, and the states composing the loop invariant as I. Then, $\rho \in I$ can be interpreted by induction through the above definition as:

- either we have not performed any loop iteration yet, and $\rho \in R$;
- or we have performed one or more loop iterations, that is, there is some state $\rho' \in I$, the result of the previous iteration, that additionally satisfies the loop condition c and lead to ρ after executing the loop body s, i.e., $\rho \in \mathbb{S}[\![s]\!](\mathbb{C}[\![c]\!]\{\rho'\})$.

We have thus established the following equation: $I = R \cup \mathbb{S}[\![s]\!](\mathbb{C}[\![c]\!]I)$, which can be rewritten as $I = F(I)$ given our definition of F. An invariant set I is thus any fixpoint of F. By computing the least fixpoint $\mathrm{lfp}\, F$, we indeed retrieve the smallest, tightest loop invariant.

The following theorem states that this least fixpoint indeed exists:

Theorem 3.1. *For any statement $s \in$ stat, the semantic function $\mathbb{S}[\![s]\!] : \mathcal{D} \to \mathcal{D}$ is a join morphism; hence, it is monotonic and continuous. As a consequence, the least-fixpoints used in the semantics of loops are also well-defined, through both Tarski's and Kleene's fixpoint theorems (Thms. 2.1, 2.2).* ■

Applying Kleene's theorem to $\mathrm{lfp}\, F$ provides additional insights on this invariant and a connection with the iterative nature of loops. Indeed, Thm. 2.2 states that $\mathrm{lfp}\, F = \cup_{i \in \mathbb{N}} F^i(\emptyset)$. We observe that:

- $F^0(\emptyset) = \emptyset$;
- $F^1(\emptyset) = R$ is the set of states before entering the loop;
- $F^2(\emptyset) = R \cup \mathbb{S}[\![s]\!](\mathbb{C}[\![c]\!]I)$ is the set of states after zero or one loop iteration;
- more generally, $F^n(\emptyset)$ is the set of states at the loop head (before testing the condition c) after at most $n-1$ loop iterations;
- the limit $\cup_{i \in \mathbb{N}} F^i(\emptyset)$ is thus the set of states after an arbitrary number of loop iterations.

We can view $F(X) \stackrel{\text{def}}{=} R \cup \mathbb{S}[\![s]\!](\mathbb{C}[\![c]\!]X)$ as applying one more loop iteration to the memory states X: it first selects which states in X will continue the loop, using $\mathbb{C}[\![c]\!]$, and then apply the loop body $\mathbb{S}[\![s]\!]$ to them, shifting X by one loop iteration, and then add back R to get back the 0-iteration case. Hence, we accumulate in the least fixpoint the effect of all possible loop iterations. Thus, the tightest invariant also corresponds exactly to the set of reachable states at the loop head.

The second step, after computing $\mathrm{lfp}\, F$, is to filter out the values by $\mathbb{C}[\![\neg c]\!]$, keeping only those states that can exit the loop by failing the loop condition c.

Example 3.3 (Loop semantics)**.** *We come back to the modulo example from Fig. 1.1 — see also Fig. 3.8.(a) — starting with $A = 10$, $B = 3$. When reaching the loop at line 3, the unique possible memory state is $X = \{(10, 3, 0, 10\}$, denoting in that order the values of variables A, B, Q, R. The loop invariant is computed as follows:*

- *we always have $F^0(\emptyset) = \emptyset$ and $F^1(\emptyset) = X$;*
- *we have $\mathbb{C}[\![R \geq B]\!]X = X$ and $\mathbb{S}[\![R \leftarrow R - B; Q \leftarrow Q + 1]\!] = \{(10, 3, 1, 7)\}$, so that $F^2(\emptyset) = \{(10, 3, 0, 10), (10, 3, 1, 7)\}$;*
- *then $F^3(\emptyset) = \{(10, 3, 0, 10), (10, 3, 1, 7), (10, 3, 2, 4)\}$;*
- *and $F^4(\emptyset) = \{(10, 3, 0, 10), (10, 3, 1, 7), (10, 3, 2, 4), (10, 3, 3, 1)\}$;*
- *we then note that $\mathbb{C}[\![R \geq B]\!]F^4(\emptyset) = F^3(\emptyset)$, so that we repeat the previous computation, and $F^5(\emptyset) = F^4(\emptyset)$: we have reached the fixpoint.*

This computation leads to the the following memory state after the loop: $\mathbb{C}[\![R < B]\!]F^4(\emptyset) = \{(10, 3, 3, 1)\}$.

Although this example demonstrates a convergence after a finite number of iterations, this is not necessarily the case. Consider, instead of a singleton X, all the states satisfying the specification $A \geq 0$, $B \geq 0$, i.e., $X = \{\, (a, b, 0, a) \mid a, b \in \mathbb{N} \,\}$. We could then show, by induction on i, that $F^i(\emptyset) = \{\, (a, b, k, a - kb) \mid a \geq 0,\ b \geq 0,\ 0 \leq k < i,\ a - kb \geq 0 \,\}$. At iteration $k < i$ of the loop, provided that $a \geq kb$, the current remainder is $a - kb$. The sets $F^i(\emptyset)$ keep increasing indefinitely and, at the limit, we get: $\cup_i F^i(\emptyset) = \{\, (a, b, k, a - kb) \mid a \geq 0,\ b \geq 0,\ k \geq 0,\ a - kb \geq 0 \,\}$. When exiting the loop, we have the set of states $\{\, (a, b, k, a - kb) \mid a \geq 0,\ b \geq 0,\ 0 \leq a - kb < b \,\}$. This is sufficient information to state that the function indeed computes the Euclidian division and remainder of its arguments. This result has been obtained systematically, almost mechanically, through computation, aided with some induction to go to the limit. ♦

Nested loops. In the case of nested loops, the semantics features nested fixpoint computations. When interpreted using Kleene iterations, nested fixpoints lead to nested iterations, so that, for each iteration of the outer loop, we must compute again the whole sequence of iterations of the inner loop up to its limit.

In many ways, the denotational semantics follows the evaluation of an interpreter, except that it manipulates sets of memory states instead of single state, and it follows both branches of conditionals and accumulates along all (possibly unbounded) loop iterations.

Infinite loops. In the case of possibly infinite loops, the fixpoint semantics $\mathrm{lfp}\, F$ computes a loop invariant that takes into account all executions: those that exit the loop and those that stay within the loop; it enriches the reachable states with new states at each iteration. However, as the semantics of the loop $\mathbb{C}[\![\neg c]\!](\mathrm{lfp}\, F)$ only keeps the states that additionally satisfy the exit condition $\neg c$, the semantics continues after the loop only with the states from executions that do exit the loop, discarding information about infinite loops. As a consequence, when the semantics of the loop is $\emptyset$, we know that the loop never terminates but, when it is not $\emptyset$, although some executions terminate, this does not prevent other executions from looping forever.

Example 3.4 (Infinite loops)**.**

1. *Consider the loop "$Y \leftarrow X$;* **while** $Y \neq 0$ **do** $Y \leftarrow Y + 1$ **done***" starting in a singleton state set $R = \{(x, 0)\}$ where X has value $x > 0$ and Y has value 0. Then, the loop invariant computed is $\mathrm{lfp}\ F = \{\, (x, y) \mid y \geq x \,\}$. Moreover, $\mathbb{C}[\![Y = 0]\!](\mathrm{lfp}\ F) = \emptyset$, which indicates an unreachable point, as there are no possible states where the loop terminates.

2. *When starting the same loop in the state set $R = \{\, (x, 0) \mid x \in \mathbb{Z} \,\}$ where X has an unknown value, we get $\mathrm{lfp}\ F = \{\, (x, y) \mid x, y \in \mathbb{Z},\ x \leq y \wedge (x \leq 0 \implies y \leq 0) \,\}$ and $\mathbb{C}[\![Y = 0]\!](\mathrm{lfp}\ F) = \{\, (x, 0) \mid x \leq 0 \,\}$. This indicates that the program terminates only if the value of X (which is also the initial value of Y) is negative, and it states that the value of Y is zero upon termination. However, it says nothing about non-terminating executions, i.e., when $X > 0$.*

3. *Finally, consider a loop with non-deterministic body "$Y \leftarrow X$;* **while** $Y > 0$ **do** $Y \leftarrow Y - [0, 1]$ **done***." Then, given an initial state such that $X \geq 0$, some executions terminate when Y reaches 0, while some others loop forever with $Y > 0$, depending on the sequence of non-deterministic choices. We get as loop invariant $\mathrm{lfp}\, F = \{\, (x, y) \mid 0 \leq y \leq x \,\}$ and, as exit states, $\{\, (x, 0) \mid x \geq 0 \,\}$. This states that, for all executions that terminate and such that $X \geq 0$, they terminate in a state where $Y = 0$, and it says nothing about non-terminating executions such that $X \geq 0$.* ♦

Multiple fixpoints. We have justified by Kleene's theorem that the states reachable after an arbitrary number of loop iterations correspond to the least fixpoint of F. However, F may have other fixpoints, which correspond to invariants containing non-reachable points.

Example 3.5 (Multiple fixpoints)**.** *Consider, for instance, the loop:*

```
while true do
    if [0,1] = 0 then
        if x ≤ 5 then x ← x + 1 endif
    endif
done
```

starting in the state $R \stackrel{\text{def}}{=} \{0\}$*. Then, the loop body semantic function is* $F(X) \stackrel{\text{def}}{=} R \cup (\{\, x+1 \mid x \in X,\, x \leq 5 \,\} \cup X)$*. The least fixpoint is* $\mathrm{lfp}\, F = \{0,1,2,3,4,5\}$*. However,* $\{0,1,2,3,4,5,6\}$ *is also a fixpoint, and there are many others, up to the greatest fixpoint,* $\mathbb{Z}$*. This is due to the presence of* X *in* $F(X)$*, caused by the implicit* **else** *branch of the non-deterministic conditional* **if** $[0,1] = 0$*. This presence allows self-justification, in the fixpoint, of values, such as* 6*, which are never computed by Kleene iterations: they appear in* $F(X)$ *because we put them in* X*.* ♦

3.3.3 Program Semantics

Given a program $p \in \mathit{stat}$ and a set of initial states $I \subseteq \mathcal{E}$, our semantics $\mathbb{S}[\![\, p \,]\!] I$ computes the set of memory states at the end of the program execution. The semantics does not exactly output the program invariants at all locations, then, only at the end, although it does compute all of them during the evaluation. It would be a simple matter to instrument the semantics to store, in a table, these invariants as they are computed, which we omitted as it does not fit well the functional-style definition of Fig. 3.7.

Additionally, we stated that, in the semantics of atomic statements, errors, such as assertion failures or divisions by zero, discard memory states, as does the semantics of loops for non-terminating executions. An empty set $\emptyset$ is propagated by all semantic functions as they are *strict*, i.e., $\mathbb{S}[\![\, s \,]\!] \emptyset = \emptyset$. Hence, the semantics of a program outputs the set of memory states at the end of the executions that terminate without any error, and provides no information about other executions. In practice, an analyzer would actually report any division by zero or assertion failure, as well as definite non-termination (i.e., a loop with non-empty input but an empty output), either through side-effects (e.g., print an error message) or through a set of error locations that is threaded through the interpretation of the program and ultimately returned with the final invariant. For the sake of simplicity, we do not present these refinements in our semantics, focusing on invariants instead.

3.4 Equation-Based Semantics

Another popular view of a program semantics is through an equation system.

We start with an example, in Fig. 3.8.(a), showing the modulo program from Fig. 1.1, rewritten in our syntax and annotated with the program locations: $\ell 1, \ldots, \ell 7$. Figure 3.8.(b) gives its corresponding equation system. The variables of the system correspond to invariants to be computed at each program location. We denote by $\mathcal{X}_i$ the variable for program point ℓi. Its value is a set of states, in $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E})$. The equations then have the form $\mathcal{X}_i = F_i(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ and express the invariant at each program point ℓi as a function of the states at previous program points or, in the case of loops, following program points.

The equation system is closely related to the control-flow graph of the program, as shown in Fig. 3.8.(c): each node in the graph is a program location, and each arc corresponds to an assignment or a condition. Note that, in the presence of loops, there are dependency cycles between the variables of the system.

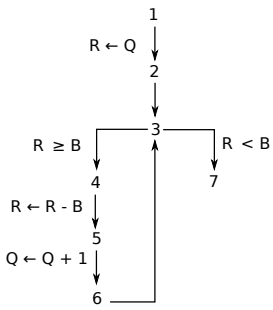
We now present the systematic construction of this equation system, and its solving.

$$
\begin{array}{l}
{}^{\ell 1}\, R \leftarrow Q; \\
{}^{\ell 2}\, \textbf{while}\ {}^{\ell 3}\, R \geq B\ \textbf{do} \\
\quad {}^{\ell 4}\, R \leftarrow R - B; \\
\quad {}^{\ell 5}\, Q \leftarrow Q + 1 \\
{}^{\ell 6}\, \textbf{done}\, {}^{\ell 7}
\end{array}
$$

(a)

$$
\begin{array}{l}
\mathcal{X}_2 = \mathbb{S}[\![\, R \leftarrow Q \,]\!]\mathcal{X}_1 \\
\mathcal{X}_3 = \mathcal{X}_2 \cup \mathcal{X}_6 \\
\mathcal{X}_4 = \mathbb{C}[\![\, R \geq B \,]\!]\mathcal{X}_3 \\
\mathcal{X}_5 = \mathbb{S}[\![\, R \leftarrow R - B \,]\!]\mathcal{X}_4 \\
\mathcal{X}_6 = \mathbb{S}[\![\, Q \leftarrow Q + 1 \,]\!]\mathcal{X}_5 \\
\mathcal{X}_7 = \mathbb{C}[\![\, R < B \,]\!]\mathcal{X}_3
\end{array}
$$

(b)



(c)

Figure 3.8: Modulo program from Fig. 1.1 but in our syntax (a), the corresponding semantic equation system (b), and its control-flow graph (c).

$$
\underline{\mathit{cfg}\,[\mathit{stat}] \in \mathcal{P}(\mathcal{L} \times (\mathcal{D} \to \mathcal{D}) \times \mathcal{L})}
$$

$$
\begin{array}{lcl}
\mathit{cfg}\,[\,{}^{\ell 1}\, V \leftarrow e\, {}^{\ell 2}\,] & \stackrel{\text{def}}{=} & \{(\ell 1, \mathbb{S}[\![\, V \leftarrow e \,]\!], \ell 2)\} \\
\mathit{cfg}\,[\,{}^{\ell 1}\, \textbf{assert}\ c\, {}^{\ell 2}\,] & \stackrel{\text{def}}{=} & \{(\ell 1, \mathbb{C}[\![\, c \,]\!], \ell 2)\} \\
\mathit{cfg}\,[\,{}^{\ell 1}\, \textbf{skip}\, {}^{\ell 2}\,] & \stackrel{\text{def}}{=} & \{(\ell 1, \mathit{id}, \ell 2)\} \\
\mathit{cfg}\,[\,{}^{\ell 1}\, s_1;\ {}^{\ell 2}\, s_2\, {}^{\ell 3}\,] & \stackrel{\text{def}}{=} & \mathit{cfg}\,[\,{}^{\ell 1}\, s_1\, {}^{\ell 2}\,] \cup \mathit{cfg}\,[\,{}^{\ell 2}\, s_2\, {}^{\ell 3}\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{cfg}\,[\,{}^{\ell 1}\, \textbf{if}\ c\ \textbf{then}\ {}^{\ell 2}\, s_1\, {}^{\ell 3}\ \textbf{else}\ {}^{\ell 4}\, s_2\, {}^{\ell 5}\, \textbf{endif}\, {}^{\ell 6}\,] \stackrel{\text{def}}{=} \\
\quad \{(\ell 1, \mathbb{C}[\![\, c \,]\!], \ell 2),\ (\ell 1, \mathbb{C}[\![\, \neg c \,]\!], \ell 4),\ (\ell 3, \mathit{id}, \ell 6),\ (\ell 5,\ \mathit{id}, \ell 6)\} \cup \\
\quad \mathit{cfg}\,[\,{}^{\ell 2}\, s_1\, {}^{\ell 3}\,] \cup \mathit{cfg}\,[\,{}^{\ell 4}\, s_2\, {}^{\ell 5}\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{cfg}\,[\,{}^{\ell 1}\, \textbf{while}\ {}^{\ell 2}\, c\ \textbf{do}\ {}^{\ell 3}\, s\, {}^{\ell 4}\, \textbf{done}\, {}^{\ell 5}\,] \stackrel{\text{def}}{=} \\
\quad \{(\ell 1, \mathit{id}, \ell 2),\ (\ell 2, \mathbb{C}[\![\, c \,]\!], \ell 3),\ (\ell 2, \mathbb{C}[\![\, \neg c \,]\!], \ell 5),\ (\ell 4, \mathit{id}, \ell 2)\} \cup \\
\quad \mathit{cfg}\,[\,{}^{\ell 3}\, s\, {}^{\ell 4}\,]
\end{array}
$$

Figure 3.9: Control-flow graph generated from a program.

3.4.1 Equation System Construction

Figure 3.9 presents the systematic construction of a control-flow graph $\mathit{cfg}\,[p]$ given a program $p \in \mathit{stat}$ suitably annotated with control locations — as red superscripts. We denote as $\mathcal{L}$ the set of all control locations in p, which also correspond to the nodes of the control-flow graph. Note that, for a loop "${}^{\ell 1}\, \textbf{while}\ {}^{\ell 2}\, c\ \textbf{do}\ {}^{\ell 3}\, s\, {}^{\ell 4}\, \textbf{done}\, {}^{\ell 5}$," location $\ell 2$ corresponds to the loop invariant. Then, $\mathit{cfg}\,[p] \in \mathcal{P}(\mathcal{L} \times (\mathcal{D} \to \mathcal{D}) \times \mathcal{L})$ gives the arcs of the graph as a set of triples $(\ell 1, F, \ell 2)$, where $\ell 1$ and $\ell 2$ are location nodes and $F : \mathcal{D} \to \mathcal{D}$ is an atomic semantic function to apply to the states at $\ell 1$ to get the states at $\ell 2$. Naturally, some nodes have several incoming arcs, such as when the two branches of a conditional merge, and for loops.

The construction operates by induction on the syntax of the program, so that a graph for a compound statement is composed from the graphs of its sub-statements, linked through additional arcs. The semantics functions F are limited to assignments $\mathbb{S}[\![\, V \leftarrow e \,]\!]$,

conditions $\mathbb{C}[\![\, c \,]\!]$, and the identity id. The equation system is then defined as follows:

$$\forall j \in \mathcal{L}\colon \mathcal{X}_j = \begin{cases} I & \text{if } j = 1 \\ \cup_{(i,F,j)\in cfg[p]} F(\mathcal{X}_i) & \text{otherwise} \end{cases} \tag{3.2}$$

i.e., we join, at each program location ℓj, the effect of each arc (i, F, j) with destination ℓj applied to the variable at the origin location ℓi. Additionally, we set the variable at the first program point $\mathcal{X}_1$ to a fixed, user-defined value $I \in \mathcal{D}$ corresponding to the initial memory states when the program starts.

3.4.2 Equation Solving

We now justify that the equation system (3.2) always has solutions, and even a smallest solution, i.e., a solution such that every $\mathcal{X}_j$ is minimal for $\subseteq$.

We can indeed view (3.2) as a fixpoint equation. We denote the set of all equation variables in vector form: $\vec{\mathcal{X}} \stackrel{\text{def}}{=} (\mathcal{X}_1, \ldots, \mathcal{X}_{|\mathcal{L}|}) \in \overline{\mathcal{D}}$, where $\overline{\mathcal{D}} \stackrel{\text{def}}{=} \mathcal{L} \to \mathcal{D}$ is a complete powerset lattice obtained by extending $\mathcal{D}$ pointwise to functions mapping each program location to a set of states. Then (3.2) can be written as $\vec{\mathcal{X}} = \vec{F}(\vec{\mathcal{X}})$ where $\vec{F}$ is a combination of semantic operators and joins. As the atomic semantic functions are continuous in $\mathcal{D}$, so is $\vec{F}$ in $\overline{\mathcal{D}}$, hence, $\vec{F}$ has a smallest fixpoint, by both Tarski's and Kleene's fixpoint theorems (Thms. 2.1, 2.2).

Additionally, Kleene's theorem provides a solving method by iteration. Starting from the least element, we iterate:

$$\begin{aligned} \forall j \in \mathcal{L}\colon \quad \mathcal{X}_j^0 &\stackrel{\text{def}}{=} \emptyset \\ \forall j \in \mathcal{L}\colon \quad \mathcal{X}_j^{k+1} &\stackrel{\text{def}}{=} \begin{cases} I & \text{if } j = 1 \\ \cup_{(i,F,j)\in cfg[p]} F(\mathcal{X}_i^k) & \text{otherwise} \end{cases} \end{aligned} \tag{3.3}$$

and take the limit. The tightest invariant at program point i is: $\cup_{k\geq 0} \mathcal{X}_i^k$.

Notwithstanding the fact that the iteration may not converge in finite time, as we are still in the concrete world, note that this iteration scheme is rather naive, as it recomputes every equation at each iteration, even if no variable $\mathcal{X}_i$ used in the equation has changed since the last iteration. Smarter iteration techniques exist, such as *chaotic iterations* introduced by Cousot [1977] and subsequently refined by Bourdoncle [1993b]. In these techniques, only one equation is recomputed at each iteration, and it is carefully chosen to improve state propagation and avoid useless computations.

3.4.3 Comparison with the Denotational Semantics

Both the denotational semantics from Fig. 3.7 and the equational semantics from (3.3) compute the same invariants. The denotational semantics can be seen as an equation solving restricted to a specific iteration scheme that must follow the control-flow of the program, while general chaotic iterations on equation systems are much more flexible. It

is also easier to generalize the equational semantics to control-flow constructions, such as *goto* and *break*. Finally, it is also straightforward to extend the equational semantics to parallel programs, by computing the product of the control-flow graphs of the parallel processes.

On the other hand, the equation-based semantics requires one variable in $\mathcal{D}$ per program location, which is not practicable for programs with a realistic size. The denotation-style semantics is much more parsimonious. For instance, in a sequence $s_1; s_2$, the input state R before s_1 can be discarded as soon as $R' \stackrel{\text{def}}{=} \mathbb{S}[\![s_1]\!] R$ has been computed, and R' as soon as the result $\mathbb{S}[\![s_2]\!] R'$ has been computed: in general, we do not need to keep the intermediate results around. More precisely, we only need to keep additional state sets when computing the effect of both branches of a conditional, or when accumulating a loop invariant at a loop head. Hence, the maximum number of state sets during the computation is linear in the maximum depth of nested loops and conditionals, which is generally much lower than the total size of the program. This is the reason a static analyzer targeting large programs, such as Astrée [Bertrane et al., 2015], uses a denotational-style semantics.

Given that each method has its benefits and drawbacks, we will not settle for one in particular. This is possible, as we will see, because a large part of the design of a static analyzer is actually independent from this choice.

3.5 Abstract Semantics

Sections 3.3–3.4 presented two concrete collecting semantics, able to express exactly the most precise program invariants. Unfortunately, they are not computable, due to three reasons:

1. the concrete elements live in $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E})$, and so, are not all representable in a computer — even when restricting the domain of variables $\mathbb{I}$ to be finite machine-integers or floats, although $\mathcal{D}$ becomes finite, it remains extremely large, so that a naive representation of sets in extension is not practicable;

2. the semantic operators for atomic statements $\mathbb{S}[\![V \leftarrow e]\!]$, $\mathbb{C}[\![c]\!]$ and join $\cup$ are not computable — or, given a finite $\mathcal{D}$, would be too costly to evaluate individually on each memory state;

3. the iterations required in the semantics of loops, for the denotational semantics, or for general solving in the equational semantics, may not converge in finite time — or, for finite $\mathcal{D}$, may require iterating on finite, yet extremely long chains.

We solve points 1 and 2 through abstraction, and point 3 through convergence acceleration, leveraging the Abstract Interpretation framework of Chap. 2.

3.5.1 Abstract Domains

We will replace the computation in the concrete domain $\mathcal{D}$ with a computation in some abstract domain $\mathcal{D}^\sharp$. We do not set the domain yet: the following two chapters will be devoted entirely to presenting abstract domains. Our goal here is to present a generic static analyzer parameterized by an abstract domain, and to list the operators and properties required on this domain. We thus assume that we have the following components:

Definition 3.1 (Abstract domain)**.** *An* abstract domain *is given by:*

- *a set $\mathcal{D}^\sharp$ of computer-representable abstract values;*
- *an effective partial order $\sqsubseteq^\sharp$ on $\mathcal{D}^\sharp$;*
- *a monotonic concretization function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$;*
- *a smallest element $\bot^\sharp \in \mathcal{D}^\sharp$ and a largest element $\top^\sharp \in \mathcal{D}^\sharp$ that represent respectively $\emptyset$ and $\mathcal{E}$;*
- *(optionally) a Galois connection $(\mathcal{D}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$ (Def. 2.12);*
- *sound and effective abstractions of assignments and atomic arithmetic conditions:*

$$\begin{array}{l}
\mathbb{S}^\sharp[\![\, V \leftarrow e \,]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp \\
\mathbb{C}^\sharp[\![\, e_1 \bowtie e_2 \,]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp \\
\textit{such that:} \\
\forall X^\sharp \in \mathcal{D}^\sharp\colon (\mathbb{S}[\![\, V \leftarrow e \,]\!] \circ \gamma) X^\sharp \subseteq (\gamma \circ \mathbb{S}^\sharp[\![\, V \leftarrow e \,]\!]) X^\sharp \\
\forall X^\sharp \in \mathcal{D}^\sharp\colon (\mathbb{C}[\![\, e_1 \bowtie e_2 \,]\!] \circ \gamma) X^\sharp \subseteq (\gamma \circ \mathbb{C}^\sharp[\![\, e_1 \bowtie e_2 \,]\!]) X^\sharp
\end{array}$$

- *sound and effective abstractions of set union $\cup$ and set intersection $\cap$:*

$$\begin{array}{l}
\cup^\sharp : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp \\
\cap^\sharp : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp \\
\textit{such that:} \\
\forall X^\sharp, Y^\sharp \in \mathcal{D}^\sharp\colon \gamma(X^\sharp) \cup \gamma(Y^\sharp) \subseteq \gamma(X^\sharp \cup^\sharp Y^\sharp) \\
\forall X^\sharp, Y^\sharp \in \mathcal{D}^\sharp\colon \gamma(X^\sharp) \cap \gamma(Y^\sharp) \subseteq \gamma(X^\sharp \cap^\sharp Y^\sharp)
\end{array}$$

- *a widening operator $\triangledown$ (Def. 2.17).* ■

We use the traditional convention of distinguishing the abstract version $X^\sharp$ of an operator or an element from its concrete version X by adding a $\sharp$ superscript. Note that the abstract elements must come with a computer representation, and operators must be given as effective algorithms. Hence, the choice of an abstract domain is not only a choice of semantics and expressiveness, but also an algorithmic one.

$$\mathbb{S}^\sharp[\![\, stat \,]\!], \mathbb{C}^\sharp[\![\, cond \,]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$$

$$\begin{array}{lll}
\mathbb{S}^\sharp[\![\, V \leftarrow e \,]\!]R^\sharp & & \text{given} \\
\mathbb{C}^\sharp[\![\, e_1 \bowtie e_2 \,]\!]R^\sharp & & \text{given} \\
\mathbb{S}^\sharp[\![\, \textbf{assert}\ c \,]\!]R^\sharp & \stackrel{\text{def}}{=} & \mathbb{C}^\sharp[\![\, c \,]\!]R^\sharp \\
\mathbb{S}^\sharp[\![\, \textbf{skip} \,]\!]R^\sharp & \stackrel{\text{def}}{=} & R^\sharp \\
\mathbb{S}^\sharp[\![\, s_1;\, s_2 \,]\!]R^\sharp & \stackrel{\text{def}}{=} & (\mathbb{S}^\sharp[\![\, s_2 \,]\!] \circ \mathbb{S}^\sharp[\![\, s_1 \,]\!])R^\sharp \\
\mathbb{S}^\sharp[\![\, \textbf{if}\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{endif} \,]\!]R^\sharp & \stackrel{\text{def}}{=} & \\
\quad \mathbb{S}^\sharp[\![\, s_1 \,]\!](\mathbb{C}^\sharp[\![\, c \,]\!]R^\sharp) \cup^\sharp \mathbb{S}^\sharp[\![\, s_2 \,]\!](\mathbb{C}^\sharp[\![\, \neg c \,]\!]R^\sharp) & & \\
\mathbb{S}^\sharp[\![\, \textbf{while}\ c\ \textbf{do}\ s\ \textbf{done} \,]\!]R^\sharp & \stackrel{\text{def}}{=} & \mathbb{C}^\sharp[\![\, \neg c \,]\!](\lim F^\sharp) \\
\quad \text{where } F^\sharp(X^\sharp) \stackrel{\text{def}}{=} X^\sharp \triangledown (R^\sharp \cup^\sharp \mathbb{S}^\sharp[\![\, s \,]\!](\mathbb{C}^\sharp[\![\, c \,]\!]X^\sharp)) & & \\
\mathbb{C}^\sharp[\![\, \text{true} \,]\!]R^\sharp & \stackrel{\text{def}}{=} & R^\sharp \\
\mathbb{C}^\sharp[\![\, \text{false} \,]\!]R^\sharp & \stackrel{\text{def}}{=} & \bot^\sharp \\
\mathbb{C}^\sharp[\![\, c_1 \wedge c_2 \,]\!]R^\sharp & \stackrel{\text{def}}{=} & \mathbb{C}^\sharp[\![\, c_1 \,]\!]R^\sharp \cap^\sharp \mathbb{C}^\sharp[\![\, c_2 \,]\!]R^\sharp \\
\mathbb{C}^\sharp[\![\, c_1 \vee c_2 \,]\!]R^\sharp & \stackrel{\text{def}}{=} & \mathbb{C}^\sharp[\![\, c_1 \,]\!]R^\sharp \cup^\sharp \mathbb{C}^\sharp[\![\, c_2 \,]\!]R^\sharp
\end{array}$$

Figure 3.10: Denotational abstract semantics of programs.

Additionally, note that we only require soundness through a concretization function (Def. 2.15). A Galois connection, if it exists, can be used to derive optimal abstract operators as $\alpha \circ F \circ \gamma$ (Def. 2.16), but is not mandatory. If it exists, optimal abstractions can also be constructed for binary operations, such as $X^\sharp \cup^\sharp Y^\sharp \stackrel{\text{def}}{=} \alpha(\gamma(X^\sharp) \cup \gamma(Y^\sharp))$.

Finally, note that the abstract domain needs only have few structure: it is required to be a poset, but not necessarily a CPO nor a lattice. Even if $\mathcal{D}^\sharp$ is a lattice, we do not require that $\cup^\sharp$ and $\cap^\sharp$ correspond to the lub $\sqcup^\sharp$ and glb $\sqcap^\sharp$.

3.5.2 Abstract Denotational Semantics

Figure 3.10 presents the abstract version of the denotational semantics of Fig. 3.7. The semantics now operates only in the abstract domain $\mathcal{D}^\sharp$. It uses the abstract version of assignments, tests, join, and meet operators we assume given with the domain. It composes them to construct the semantics of more complex statements and tests by induction on the syntax, and we can see that it follows very closely the concrete definition, up to the use of $\sharp$ symbols.

Another key difference is that the concrete least fixpoint $\text{lfp}\, F$ used in the semantics of loops has been replaced with $\lim F^\sharp$, which computes the limit of the iterates of $F^\sharp$ from $\bot^\sharp$, as in (2.6):

$$\begin{array}{l}
\lim F^\sharp \stackrel{\text{def}}{=} F^{\sharp\delta}(\bot^\sharp) \\
\text{where } \delta \text{ is the minimal value such that } F^{\sharp\delta+1}(\bot^\sharp) = F^{\sharp\delta}(\bot^\sharp)
\end{array}$$

The definition of the widening, Def. 2.17.2, ensures that this limit is always reached

after a finite number of iterations. The result of the analysis is sound: it is the composition of sound abstractions (Thm. 2.6) and sound fixpoint abstractions with widening (Thm. 2.9), so:

Theorem 3.2 (Termination and soundness)**.** *$\mathbb{S}^\sharp[\![p]\!]$ always terminates and is sound: $\forall p \in$ stat, $I^\sharp \in \mathcal{D}^\sharp$: $\mathbb{S}[\![p]\!]\,(\gamma(I^\sharp)) \subseteq \gamma(\mathbb{S}^\sharp[\![p]\!] I^\sharp)$.* ■

3.5.3 Abstract Equational Semantics

Similarly, we can construct an effective, sound, abstract version of the equational semantics from Sect. 3.4. Instead of a variable $\mathcal{X}_i$ with value in $\mathcal{D}$ at each program point i, we have a variable $\mathcal{X}_i^\sharp$ with value in our abstract domain $\mathcal{D}^\sharp$. The control-flow graph $cfg\,[p]$ of a program $p \in stat$ remains the same as described in Fig. 3.9 but, in the equation system, for each arc $(i, F, j) \in cfg\,[p]$, we replace the concrete function $F : \mathcal{D} \to \mathcal{D}$ with a sound abstraction $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$, and $\cup$ with a sound abstraction $\cup^\sharp$. As all the F functions are assignments or conditions, they are assumed to be directly provided by the abstract domain or, for complex boolean conditions, derivable from them using the definition of $\mathbb{C}^\sharp[\![c]\!]$ from Fig. 3.10. Finally, we assume $I^\sharp$ to be a sound abstraction of the initial states I. We can then change the iteration described in (3.3) into an abstract version:

$$\begin{array}{lcl}
\forall j \in \mathcal{L}\colon \mathcal{X}_j^{\sharp 0} & \stackrel{\text{def}}{=} & \bot^\sharp \\
\forall j \in \mathcal{L}\colon \mathcal{X}_j^{\sharp k+1} & \stackrel{\text{def}}{=} & \begin{cases} I^\sharp & \text{if } j = 1 \\ \cup^\sharp_{(i,F,j)\in cfg[p]} F^\sharp(\mathcal{X}_i^{\sharp k}) & \text{otherwise} \end{cases}
\end{array}$$

Unfortunately, this iteration is not guaranteed to terminate, as the right-hand sides are not guaranteed to be monotonic, and the abstract domain may have infinite increasing chains. We can, as for the denotational semantics, ensure the convergence using the widening operator $\triangledown$. It is, however, unnecessary to apply the widening at each equation, and we will see on an example in Sect. 4.7.1 that a careless use of widening can significantly reduce the precision. In order to ensure the convergence, it is sufficient to apply a widening only for a set of control points such that every cycle of the control flow graph passes through one such point.

We assume, now, that we are given such a set $\mathcal{W} \subseteq \mathcal{L}$ of control points, that we call *widening points*. On our control-flow graphs, for instance, it is sufficient to select all loop heads, where the loop invariants are computed, as widening points, but more general algorithms have been developed by Bourdoncle [1993b] to find suitable widening points on arbitrary graphs. Then, we can use the following iteration scheme:

$$\begin{array}{lcl}
\forall j \in \mathcal{L}\colon \mathcal{X}_j^{\sharp 0} & \stackrel{\text{def}}{=} & \bot^\sharp \\
\forall j \in \mathcal{L}\colon \mathcal{X}_j^{\sharp k+1} & \stackrel{\text{def}}{=} & \begin{cases} I^\sharp & \text{if } j = 1 \\ \mathcal{X}_j^{\sharp k} \triangledown \left(\cup^\sharp_{(i,F,j)\in cfg[p]} F^\sharp(\mathcal{X}_i^{\sharp k})\right) & \text{if } j \in \mathcal{W} \\ \cup^\sharp_{(i,F,j)\in cfg[p]} F^\sharp(\mathcal{X}_i^{\sharp k}) & \text{otherwise} \end{cases}
\end{array} \tag{3.4}$$

The gives an effective, sound, and terminating static analyzer, as stated by the following theorem:

Theorem 3.3 (Termination and soundness)**.**
$\forall p \in stat$, $I^\sharp \in \mathcal{D}^\sharp$: if $I \subseteq \gamma(I^\sharp)$, then the limit $(\mathcal{X}_i^\sharp)_{i\in\mathcal{L}}$ of iteration (3.4) is reached in finite time and is a sound abstraction of the limit $(\mathcal{X}_i)_{i\in\mathcal{L}}$ of iteration (3.3), i.e., $\forall i \in \mathcal{L}$: $\mathcal{X}_i \subseteq \gamma(\mathcal{X}_i^\sharp)$. ■

Similarly to clever choices of widening points, Bourdoncle [1993b] also presents advanced iteration techniques, based on chaotic iterations by Cousot [1977], to avoid useless abstract computations when some variables do not change, and in order to improve both speed and precision. Unlike the concrete case, different iteration techniques, and different choices of widening points, may change not only the number of iterations, but also the actual result of the analysis. We will discuss this point further and present advanced abstract iteration techniques in Sect. 4.7, illustrated on interval analyses.

3.6 Bibliographic Notes

Three broad classes of semantics have been proposed. Firstly, the work of Floyd [1967] and Hoare [1969] founded *axiomatic semantics* and presented applications to proving programs through logic. Secondly, *denotational semantics*, founded by Scott and Strachey [1971], aims at assigning to programs mathematical functions that abstract away the internal computation to focus on the input/output relationship. Finally, *operational semantics*, founded by Kahn [1987] and Plotkin [1981], describes instead these computation steps in minute details.

Generally, Abstract Interpretation employs an operational-style approach to describe the collecting concrete semantics precisely enough to expose the correctness properties of interest. The first analyses, by Cousot and Cousot [1977], are presented on so-called flowchart programs, not unlike the control-flow graphs of our programs, and lead to semantics in equational-style that are still widely used in static analysis. Later presentations of Abstract Interpretation, such as [Cousot, 1981], start from transition systems, a very general and minimalist flavor of operational semantics, to state results independently from the specific choice of programming language. The connection between the collecting semantics and the aximoatic semantics of Hoare [1969] and Floyd [1967] is discussed by Cousot [1980]. State-based concrete semantics, which we use in this tutorial, are generalized to trace-based semantics in later presentations of Abstract Interpretation. This allows recasting the alternate proof method by Burstall [1974], which is not state-based, as an Abstract Interpretation in [Cousot and Cousot, 1993]. Cousot [2002] presents a hierarchy of semantics based on (finite and infinite) traces and shows how natural semantics [Kahn, 1987] and denotational semantics [Scott and Strachey, 1971] can be seen as abstractions of trace semantics. We only focus here on Abstract Interpretation as a tool to design effective and sound static analyses, but the works cited above show that it is also a powerful tool to cast seemingly incomparable semantics into a common framework and highlight novel connections.

Analyses based on solving abstract equation systems are present since the beginning of Abstract Interpretation in [Cousot and Cousot, 1977], and are further developed by Bourdoncle [1993b]. Such equations appeared previously in data-flow analyses, pioneered by Kildall [1973], but restricted to finite-height lattices and without the soundness and optimality guarantees that come from a connection with the concrete collecting semantics — both points are addressed by Abstract Interpretation, which thus goes beyond data-flow analyses. On the other hand, Bertrane et al. [2010] advocate for the use of a denotational-style semantics, as an interpretation by induction on the syntax. We present both views in this tutorial.

We only briefly discussed in Chap. 1 the notion of backward analysis. This notion is already present in the early work of Cousot and Cousot [1979a] and further developed by Bourdoncle [1993a], on equational-style semantics.

Chapter 4

Non-Relational Abstract Domains

The previous chapter presented a simple language and a generic static analyzer parameterized by the choice of an abstract domain equipped with a well-defined set of sound abstract operators. This chapter and the following present several such abstract domains.

In Chap. 2, we illustrated the notion of abstract domain by constructing abstractions for sets of integers, such as intervals. A natural idea is to lift such abstractions to memory states so that we can assign to each variable an abstraction of its set of possible values. We already demonstrated this idea informally in Sect. 1.1 with interval and sign analyses. This leads to a family of analyses that abstract each program variable independently, but do not take variable relationships into account. This chapter presents in a fully formal way these so-called *non-relational analyses* on the numeric programs of Chap. 3.

We start by formalizing value abstractions, providing necessary operators, and state the relationship with the abstract domains of the previous chapter. We then provide well-known instances, starting with simple domains, the sign domain and the constant domain, and moving up to domains that are widely used in practice: the interval domain and the congruence domain. This chapter will also discuss advanced iteration techniques with widening, that are necessary to obtain a reasonable precision in practice on domains with infinite height. We will motivate and illustrate these techniques on the interval domain.

4.1 Value and State Abstractions

Recall that, to construct a static analysis, it is sufficient to provide an abstract domain of memory states obeying the requirements of Def. 3.1. All non-relational analyses share this idea of lifting value abstractions to state abstractions. We present here this generic lifting procedure, so that the rest of the chapter can focus on various value abstractions. This presentation also allows factoring out, for non-relational analyses, a large part which is independent from the choice of the value abstraction.

4.1.1 Value Abstract Domain

Recall that our language manipulates numeric values in a set $\mathbb{I}$, which can be $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$. Thus, our concrete domain of interest is the complete powerset lattice $(\mathcal{P}(\mathbb{I}), \subseteq, \cup, \cap, \emptyset, \mathbb{I})$. Given sets of possible values, the arithmetic operators, $+$, $-$, $\times$, $/$, return a set of possible values in the concrete. Similarly to Def. 3.1, an abstract domain will be defined as an abstraction of this concrete world, together with sound abstractions of the concrete operators. We denote it as $\mathcal{B}^\sharp$, and use a b subscript, in addition to the $\sharp$ superscript, in order to distinguish the operations on abstract sets of values from the operations on abstract sets of states (in $\mathcal{D}^\sharp$):

Definition 4.1 (Abstract value domain)**.** *An abstract domain of values is given by:*

- *a set $\mathcal{B}^\sharp$ of computer-representable abstract values;*
- *an effective partial order $\sqsubseteq_b^\sharp$ on $\mathcal{B}^\sharp$;*
- *a monotonic concretization function $\gamma_b : \mathcal{B}^\sharp \to \mathcal{P}(\mathbb{I})$;*
- *a smallest $\bot_b^\sharp$ and a largest element $\top_b^\sharp \in \mathcal{B}^\sharp$ that represent respectively $\emptyset$ and $\mathbb{I}$;*
- *(optionally) a Galois connection $(\mathcal{P}(\mathbb{I}), \subseteq) \underset{\alpha_b}{\overset{\gamma_b}{\leftrightarrows}} (\mathcal{B}^\sharp, \sqsubseteq_b^\sharp)$ (Def. 2.12);*
- *a sound and effective abstraction of constants and non-deterministic intervals:*

$$\begin{array}{rll} \forall c \in \mathbb{I}\colon c_b^\sharp & \textit{such that:} & c \in \gamma_b(c_b^\sharp) \\ \forall c_1, c_2 \in \mathbb{I} \cup \{\pm\infty\}\colon [c_1, c_2]_b^\sharp & \textit{such that:} & [c_1, c_2] \subseteq \gamma_b([c_1, c_2]_b^\sharp) \end{array}$$

  of unary operators:

$$\begin{array}{l} -_b^\sharp \in \mathcal{B}^\sharp \to \mathcal{B}^\sharp \\ \textit{such that:} \\ \forall X^\sharp \in \mathcal{B}^\sharp\colon \{\, -x \mid x \in \gamma_b(X^\sharp) \,\} \subseteq \gamma_b(-_b^\sharp X^\sharp) \end{array}$$

  and of binary operators:

$$\begin{array}{l} +_b^\sharp, -_b^\sharp, \times_b^\sharp, /_b^\sharp \in \mathcal{B}^\sharp \times \mathcal{B}^\sharp \to \mathcal{B}^\sharp \\ \textit{such that:} \\ \forall X^\sharp, Y^\sharp \in \mathcal{B}^\sharp\colon \{\, x + y \mid x \in \gamma_b(X^\sharp), y \in \gamma_b(Y^\sharp) \,\} \subseteq \gamma_b(X^\sharp +_b^\sharp Y^\sharp) \\ \forall X^\sharp, Y^\sharp \in \mathcal{B}^\sharp\colon \{\, x - y \mid x \in \gamma_b(X^\sharp), y \in \gamma_b(Y^\sharp) \,\} \subseteq \gamma_b(X^\sharp -_b^\sharp Y^\sharp) \\ \forall X^\sharp, Y^\sharp \in \mathcal{B}^\sharp\colon \{\, x \times y \mid x \in \gamma_b(X^\sharp), y \in \gamma_b(Y^\sharp) \,\} \subseteq \gamma_b(X^\sharp \times_b^\sharp Y^\sharp) \\ \forall X^\sharp, Y^\sharp \in \mathcal{B}^\sharp\colon \{\, x/y \mid x \in \gamma_b(X^\sharp), y \in \gamma_b(Y^\sharp), y \neq 0 \,\} \subseteq \gamma_b(X^\sharp /_b^\sharp Y^\sharp) \end{array}$$

- *a sound and effective abstraction of set union $\cup$ and set intersection $\cap$:*

$$\begin{array}{l} \cup_b^\sharp, \cap_b^\sharp : \mathcal{B}^\sharp \times \mathcal{B}^\sharp \to \mathcal{B}^\sharp \\ \textit{such that:} \\ \forall X^\sharp, Y^\sharp \in \mathcal{B}^\sharp\colon \gamma_b(X^\sharp) \cup \gamma_b(Y^\sharp) \subseteq \gamma_b(X^\sharp \cup_b^\sharp Y^\sharp) \\ \forall X^\sharp, Y^\sharp \in \mathcal{B}^\sharp\colon \gamma_b(X^\sharp) \cap \gamma_b(Y^\sharp) \subseteq \gamma_b(X^\sharp \cap_b^\sharp Y^\sharp) \end{array}$$

- *a widening operator* $\triangledown_b$ *(Def. 2.17).* ■

As for memory state domains, the Galois connection, if it exists, can be used to derive the best abstraction of each operator; for instance:

$$X^\sharp +_b^\sharp Y^\sharp \stackrel{\text{def}}{=} \alpha_b(\{\, x + y \mid x \in \gamma_b(X^\sharp),\, y \in \gamma_b(Y^\sharp)\,\})$$

but it is otherwise optional.

4.1.2 State Abstract Domain

To derive the abstraction $\mathcal{D}^\sharp$ of $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{V} \to \mathbb{I})$ from the abstraction $\mathcal{B}^\sharp$ of $\mathcal{P}(\mathbb{I})$, we apply the coalescent version of point-wise lifting (Def. 2.7), which maps each variable in $\mathbb{V}$ to an abstract element in $\mathcal{B}^\sharp$, coalescing all elements where $\bot_b^\sharp$ appears into a unique $\bot^\sharp$ element:

$$\mathcal{D}^\sharp \stackrel{\text{def}}{=} (\mathbb{V} \to (\mathcal{B}^\sharp \setminus \{\bot_b^\sharp\})) \cup \{\bot^\sharp\} \tag{4.1}$$

Many of the operators needed on $\mathcal{D}^\sharp$ can be derived from those available on $\mathcal{B}^\sharp$ by point-wise lifting:

Definition 4.2 (Non-relational state abstraction)**.**
Given operators on $\mathcal{B}^\sharp$ *following Def. 4.1, we define the following operators on* $\mathcal{D}^\sharp$*:*

- $X^\sharp \sqsubseteq Y^\sharp \stackrel{\text{def}}{\iff} (X^\sharp = \bot^\sharp) \lor (X^\sharp, Y^\sharp \neq \bot^\sharp \land \forall V \in \mathbb{V}\colon X^\sharp(V) \sqsubseteq_b^\sharp Y^\sharp(V))$
- $\gamma(X^\sharp) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \textit{if } X^\sharp = \bot^\sharp \\ \{\, \rho \in \mathcal{E} \mid \forall V \in \mathbb{V}\colon \rho(V) \in \gamma_b(X^\sharp(V))\,\} & \textit{otherwise} \end{cases}$
- $\alpha(R) \stackrel{\text{def}}{=} \begin{cases} \bot^\sharp & \textit{if } R = \emptyset \\ \lambda V \in \mathbb{V}.\, \alpha_b(\{\, \rho(V) \mid \rho \in R\,\} & \textit{otherwise} \end{cases}$
  when α_b *exists.*
- $\top^\sharp \stackrel{\text{def}}{=} \lambda V \in \mathbb{V}.\, \top_b^\sharp$
- $X^\sharp \cup^\sharp Y^\sharp \stackrel{\text{def}}{=} \begin{cases} Y^\sharp & \textit{if } X^\sharp = \bot^\sharp \\ X^\sharp & \textit{if } Y^\sharp = \bot^\sharp \\ \lambda V \in \mathbb{V}.\, X^\sharp(V) \cup_b^\sharp Y^\sharp(V) & \textit{otherwise} \end{cases}$
- $X^\sharp \triangledown Y^\sharp \stackrel{\text{def}}{=} \begin{cases} Y^\sharp & \textit{if } X^\sharp = \bot^\sharp \\ X^\sharp & \textit{if } Y^\sharp = \bot^\sharp \\ \lambda V \in \mathbb{V}.\, X^\sharp(V) \triangledown_b Y^\sharp(V) & \textit{otherwise} \end{cases}$
- $X^\sharp \cap^\sharp Y^\sharp \stackrel{\text{def}}{=} \begin{cases} \bot^\sharp & \textit{if } X^\sharp = \bot^\sharp \textit{ or } Y^\sharp = \bot^\sharp \\ \bot^\sharp & \textit{if } \exists V \in \mathbb{V}\colon X^\sharp(V) \cap_b^\sharp Y^\sharp(V) = \bot_b^\sharp \\ \lambda V \in \mathbb{V}.\, X^\sharp(V) \cap_b^\sharp Y^\sharp(V) & \textit{otherwise} \end{cases}$ ■

Note in particular that, if $\mathcal{B}^\sharp$ enjoys a Galois connection $(\mathcal{P}(\mathbb{I}), \subseteq) \underset{\alpha_b}{\overset{\gamma_b}{\leftrightarrows}} (\mathcal{B}^\sharp, \sqsubseteq_b^\sharp)$, then this is also the case for the domain $\mathcal{D}^\sharp$ we derive: $(\mathcal{D}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$. All the operators derived above are also sound, and optimal if the underlying value operator in $\mathcal{B}^\sharp$ is optimal. Finally, the widening $\triangledown$ satisfies Def. 2.17: termination is ensured by stabilizing independently the abstract value associated to each variable.

We are now only missing the abstraction of assignments and conditions. We first remark that it is always possible to resort to the following sound, but very coarse, fallback definitions, whatever the abstract value domain $\mathcal{B}^\sharp$:

Definition 4.3 (Fallback abstract operators)**.**
The following abstract operators are always sound:

$$\begin{array}{lcl} \mathbb{S}^\sharp[\![\, V \leftarrow e \,]\!] X^\sharp & \stackrel{\text{def}}{=} & X^\sharp[V \mapsto \top_b^\sharp] \\ \mathbb{C}^\sharp[\![\, e_1 \bowtie e_2 \,]\!] X^\sharp & \stackrel{\text{def}}{=} & X^\sharp \end{array} \tag{4.2}$$

■

The fallback assignment is obviously sound as it forgets any information on the value of the assigned variable. For the condition, we simply remark that, as the concrete semantics removes some possible states, it is sound to use the identity instead, as it refrains from removing any state.

In the following, we will provide more accurate definitions, but it is important to know that we can always count on the fallback operators to perform a sound analysis, in case more accurate definitions are not available, or are too costly to apply.

4.1.3 Abstract Expression Evaluation

The concrete semantics of expressions, from Fig. 3.4, evaluates the expression on a single state by induction on the syntax, propagating a set of possible values. This algorithm can be translated easily into the abstract: we take as input an abstract representation of a set of values for each variable and propagate, by induction on the syntax, an abstract representation of sets of values.

Figure 4.1 presents such an abstract evaluation of expressions. It is naturally sound as it composes sound abstraction, i.e.:

Theorem 4.1 (Soundness of abstract expression evaluation)**.**
$\forall e \in \mathit{expr}\colon \forall X^\sharp \in \mathcal{D}^\sharp\colon \forall \rho \in \gamma(X^\sharp)\colon \mathbb{E}[\![\, e \,]\!]\, \rho \subseteq \gamma_b(\mathbb{E}^\sharp[\![\, e \,]\!] X^\sharp)$. ■

Note, however, that this operation may not be optimal, even if every abstract operator in $\mathcal{B}^\sharp$ is optimal. This is in particular the case when one variable occurs several times in the expression, as the algorithm cannot exploit the fact that both occurrences of the variable evaluate to the exact same value in a given state, as demonstrated below:

Example 4.1 (Non-optimality of non-relational abstract evaluation)**.** *Consider an abstract state $X^\sharp$ such that $\gamma(X^\sharp(V)) = \{0, 1\}$, for instance using an interval abstraction*

$$
\begin{array}{lcl}
\mathbb{E}^\sharp[\![\, expr \,]\!] : \mathcal{D}^\sharp \to \mathcal{B}^\sharp & & \\
\hline
\mathbb{E}^\sharp[\![\, V \,]\!]X^\sharp & \stackrel{\text{def}}{=} & X^\sharp(V) \\
\mathbb{E}^\sharp[\![\, c \,]\!]X^\sharp & \stackrel{\text{def}}{=} & c_b^\sharp \\
\mathbb{E}^\sharp[\![\, [c_1, c_2] \,]\!]X^\sharp & \stackrel{\text{def}}{=} & [c_1, c_2]_b^\sharp \\
\mathbb{E}^\sharp[\![\, -e \,]\!]X^\sharp & \stackrel{\text{def}}{=} & -_b^\sharp(\mathbb{E}^\sharp[\![\, e \,]\!]X^\sharp) \\
\mathbb{E}^\sharp[\![\, e_1 + e_2 \,]\!]X^\sharp & \stackrel{\text{def}}{=} & (\mathbb{E}^\sharp[\![\, e_1 \,]\!]X^\sharp) +_b^\sharp (\mathbb{E}^\sharp[\![\, e_2 \,]\!]X^\sharp) \\
\mathbb{E}^\sharp[\![\, e_1 - e_2 \,]\!]X^\sharp & \stackrel{\text{def}}{=} & (\mathbb{E}^\sharp[\![\, e_1 \,]\!]X^\sharp) -_b^\sharp (\mathbb{E}^\sharp[\![\, e_2 \,]\!]X^\sharp) \\
\mathbb{E}^\sharp[\![\, e_1 \times e_2 \,]\!]X^\sharp & \stackrel{\text{def}}{=} & (\mathbb{E}^\sharp[\![\, e_1 \,]\!]X^\sharp) \times_b^\sharp (\mathbb{E}^\sharp[\![\, e_2 \,]\!]X^\sharp) \\
\mathbb{E}^\sharp[\![\, e_1 / e_2 \,]\!]X^\sharp & \stackrel{\text{def}}{=} & (\mathbb{E}^\sharp[\![\, e_1 \,]\!]X^\sharp) /_b^\sharp (\mathbb{E}^\sharp[\![\, e_2 \,]\!]X^\sharp)
\end{array}
$$

Figure 4.1: Abstract semantic of expressions in a non-relational domain.

and $X^\sharp(V) = [0, 1]$. *Consider the expression* $V - V$. *Then, naturally* $\mathbb{E}[\![\, V - V \,]\!]\rho = \{0\}$ *for any state* ρ *and* a forciori *when* $\rho \in \gamma(X^\sharp)$. *However,* $\mathbb{E}^\sharp[\![\, V - V \,]\!]X^\sharp = X^\sharp(V) -_b^\sharp X^\sharp(V)$ *which evaluates using interval arithmetic — as we will see in more details in Sect. 4.5 — to* $[0, 1] -_b^\sharp [0, 1] = [-1, 1]$. *We thus have* $\gamma_b(\mathbb{E}^\sharp[\![\, V - V \,]\!]X^\sharp) = \{-1, 0, 1\}$, *although the concrete result,* $\{0\}$, *can be exactly represented in the interval domain. Hence,* $\mathbb{E}^\sharp[\![\, V - V \,]\!]$ *is not optimal, although* $-_b^\sharp$ *is.* ♦

We can now state a generic, non-optimal but still rather precise, abstraction for the assignment as:

$$
\mathbb{S}^\sharp[\![\, V \leftarrow e \,]\!](X^\sharp) \stackrel{\text{def}}{=} \begin{cases} \bot^\sharp & \text{if } X^\sharp = \bot^\sharp \text{ or } \mathbb{E}^\sharp[\![\, e \,]\!]X^\sharp = \bot_b^\sharp \\ X^\sharp[V \mapsto \mathbb{E}^\sharp[\![\, e \,]\!]X^\sharp] & \text{otherwise} \end{cases} \tag{4.3}
$$

Note the special handling of the case where, due to divisions by zero, the abstract evaluation returns an empty set of values $\bot_b^\sharp$.

A generic algorithm to handle arbitrary tests of the form $\mathbb{C}^\sharp[\![\, e_1 \bowtie e_2 \,]\!]$ along the same principles exists, but it is rather more complicated, and we delay its presentation to Sect. 4.6, where it will be presented in-context with application to the interval domain.

4.1.4 Data-Structures and Cost

When discussing the cost, in memory and time, of non-relational domains, we can assume that an abstract value in $\mathcal{B}^\sharp$ has a constant memory cost, and that every operation in Def. 4.1, which manipulates one or two elements in $\mathcal{B}^\sharp$, has a constant time cost, independently from the program size or the number of variables. As we will see in the following sections, an element in $\mathcal{B}^\sharp$ is generally represented as an element of an enumeration (e.g., a sign) or as one or two numeric values (e.g., a constant, or two interval bounds), and an operation in $\mathcal{B}^\sharp$ reduces to a few arithmetic operations at worst. A discussion about the cost of a non-relational domain $\mathcal{D}^\sharp$ is largely independent from the choice of $\mathcal{B}^\sharp$, but rather depends on the choice of how to represent maps in $\mathcal{D}^\sharp$ and how to implement the generic operations from Def. 4.2 and Fig. 4.1. Note that the total time cost of an analysis also

depends greatly on the number of loop iterations in the abstract, which varies widely and can be unpredictable. Iteration techniques will be addressed in Sect. 4.7, while we discuss here only the time cost of individual operators.

A non-relational element in $\mathcal{D}^\sharp$ associates an element in $\mathcal{B}^\sharp$ to each variable. A natural representation is thus through arrays, or hash tables. This yields a memory cost linear in $|\mathbb{V}|$, and a constant time access to abstract values. The cost of evaluating an expression $\mathbb{E}^\sharp[\![\, e \,]\!]$ is linear in the size of the expression e. The time cost of operations is thus dominated by the need to copy arrays and, for binary operations such as $\cup^\sharp$ and $\sqsubseteq^\sharp$, to scan every array element, hence it is in $\mathcal{O}(|\mathbb{V}|)$. In fact, abstract operators generally update only a limited number of abstract values, and we should exploit this to improve the performance. Consider, for instance, a simple conditional "**if** c **then** s_1 **else** s_2 **endif**". Then, we expect the $\cup^\sharp$ operator applied at the end of the conditional to have a cost that depends only on the number of variables modified in s_1 and s_2, not on the total number of program variables.

To solve this problem, Blanchet et al. [2003] suggest switching from arrays to a functional array data-structure based on balanced binary trees — such as AVL trees [Cormen et al., 2001, Chap. 13] — which have the following characteristics:

1. Access cost is in $\mathcal{O}(\log |\mathbb{V}|)$. This is higher than the constant access of arrays.

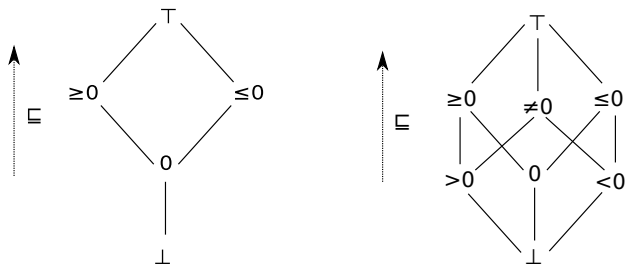
2. Updating a variable while keeping the original requires $\mathcal{O}(\log |\mathbb{V}|)$ in time and space. This is lower than the $\mathcal{O}(|\mathbb{V}|)$ time and space cost of arrays as they require a copy to keep the original.

3. The join $\cup^\sharp$ traverses both trees but, as it is idempotent, sub-trees that are physically equal (i.e., have the same address in memory) in both arguments do not need to be traversed recursively. Observe that, in practice, the join is often applied to elements stemming from a common element, as in $F^\sharp(X^\sharp) \cup^\sharp G^\sharp(X^\sharp)$ for a conditional, where $F^\sharp$ models the **then** branch and $G^\sharp$ models the **else** branch. Then, $F^\sharp(X^\sharp)$ and $G^\sharp(X^\sharp)$ differ from $X^\sharp$ in only $\mathcal{O}(\log |\mathbb{V}|)$ tree nodes times the number n of variable updates by $F^\sharp$ and $G^\sharp$. The remaining tree nodes are shared in memory with $X^\sharp$, and will not be traversed by a smart join. Hence, the cost of $\cup^\sharp$ is in $\mathcal{O}(n \log |\mathbb{V}|)$, which is often much lower than $\mathcal{O}(|\mathbb{V}|)$. Other binary operations, $\cap^\sharp$, $\sqsubseteq^\sharp$, operate similarly.

The higher cost of accesses is more than compensated by the lower cost of updates and binary operations. Additionally, this data-structure is very cache-friendly as it accesses preferentially memory parts that have been accessed recently. Blanchet et al. [2003] observed a significant performance gain when switching from arrays to functional arrays when analyzing large programs. Alternate scalable solutions include the use of sparse arrays, as suggested by Oh et al. [2012].

4.2 The Sign Domain

The first, simplest value abstract domain we present is the sign domain we already mentioned in our informal presentation (Sect. 1.1). The simple sign domain contains only a



(a) (b)

Figure 4.2: Hasse diagrams for: (a) the simple sign domain, and (b) the extended sign domain.

few elements: $\bot_b^\sharp$, 0, (≥ 0), (≤ 0), $\top_b^\sharp$. However, we can also design an extended version that includes strict signs as well as the complementary of 0: $\bot_b^\sharp$, 0, (≥ 0), (> 0), (≤ 0), (< 0), $(\neq 0)$, $\top_b^\sharp$.

The Hasse diagrams for both lattices are presented in Fig. 4.2. The diagrams implicitly define the abstract order $\sqsubseteq_b^\sharp$, the lub $\sqcup_b^\sharp$, and the glb $\sqcap_b^\sharp$. We now define the remaining operations required by Def. 4.1.

Galois connection. Both sign domains enjoy a Galois connection. For the sake of concision, we only present that of the simple sign domain; the case of extended signs is similar:

$$\alpha_b(C) \stackrel{\text{def}}{=} \begin{cases} \bot_b^\sharp & \text{if } C = \emptyset \\ 0 & \text{if } C = \{0\} \\ (\geq 0) & \text{else if } \forall c \in C\colon c \geq 0 \\ (\leq 0) & \text{else if } \forall c \in C\colon c \leq 0 \\ \top_b^\sharp & \text{otherwise} \end{cases} \qquad \begin{array}{lcl} \gamma_b(\bot_b^\sharp) & \stackrel{\text{def}}{=} & \emptyset \\ \gamma_b(0) & \stackrel{\text{def}}{=} & \{0\} \\ \gamma_b(\geq 0) & \stackrel{\text{def}}{=} & \{\, c \in \mathbb{I} \mid c \geq 0 \,\} \\ \gamma_b(\leq 0) & \stackrel{\text{def}}{=} & \{\, c \in \mathbb{I} \mid c \leq 0 \,\} \\ \gamma_b(\top_b^\sharp) & \stackrel{\text{def}}{=} & \mathbb{I} \end{array} \tag{4.4}$$

Abstract value operators. We set $\cup_b^\sharp \stackrel{\text{def}}{=} \sqcup_b^\sharp$ and $\cap_b^\sharp \stackrel{\text{def}}{=} \sqcap_b^\sharp$. The intersection $\cap_b^\sharp$ is exact, which is expected of an abstract domain enjoying a Galois connection (Thm. 2.4). The join $\cup_b^\sharp$ is also exact in both sign domains, which is actually quire rare — the union of the sets represented by two abstract elements is seldom exactly representable.

The optimal version of the arithmetic operators is given by the well-known rule of signs. Figure 4.3 gives three examples: $+_b^\sharp$, $\times_b^\sharp$, and $/_b^\sharp$ on the simple sign domain. Note that $\bot_b^\sharp$ is absorbing: the result is $\bot_b^\sharp$ whenever one argument is $\bot_b^\sharp$. Most of the time, $\top_b^\sharp$ is absorbing as well, except for the case of multiplying by 0, which returns 0 even if the other input is undetermined. The division is similar to the multiplication, except that division by 0 leads to $\bot_b^\sharp$ instead of 0.

We omit the other arithmetic operators, which are similar. Finally, abstracting constants $c_b^\sharp$ and intervals $[c_1, c_2]_b^\sharp$ is straightforward, by looking at the signs of c, c_1, and c_2.

$+$	≥ 0	≤ 0	0	$\top$	$\bot$
≥ 0	≥ 0	$\top$	≥ 0	$\top$	$\bot$
≤ 0	$\top$	≤ 0	≤ 0	$\top$	$\bot$
0	≥ 0	≤ 0	0	$\top$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$\times$	≥ 0	≤ 0	0	$\top$	$\bot$
≥ 0	≥ 0	≤ 0	0	$\top$	$\bot$
≤ 0	≤ 0	≥ 0	0	$\top$	$\bot$
0	0	0	0	0	$\bot$
$\top$	$\top$	$\top$	0	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$/$	≥ 0	≤ 0	0	$\top$	$\bot$
≥ 0	≥ 0	≤ 0	0	$\top$	$\bot$
≤ 0	≤ 0	≥ 0	0	$\top$	$\bot$
0	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\top$	$\top$	$\top$	0	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Figure 4.3: Abstract arithmetic operations the simple sign domain.

Abstracting tests. In order to handle conditions $\mathbb{C}^\sharp[\![\, e_1 \bowtie e_2 \,]\!]$, in the absence of a generic algorithm working for arbitrary expressions e_1 and e_2, a practical solution is to define ad-hoc functions for simple and useful cases, and revert to the identity as fall-back (Def. 4.3). Here is a first example for $V \leq 0$ on the simple signs, which adds the information that V is negative to existing information on V, possibly resulting in V becoming 0:

$$\mathbb{C}^\sharp[\![\, V \leq 0 \,]\!]X^\sharp \stackrel{\text{def}}{=} \begin{cases} X^\sharp[V \mapsto 0] & \text{if } X^\sharp(V) \in \{0, (\geq 0)\} \\ X^\sharp[V \mapsto (\leq 0)] & \text{if } X^\sharp(V) \in \{\top_b^\sharp, (\leq 0)\} \end{cases}$$

This second example, again on the simple signs, shows how, in a test relating two variables, we can use the information available on each variable to refine the other one:

$$\begin{aligned} \mathbb{C}^\sharp[\![\, V \leq W \,]\!]X^\sharp \quad \stackrel{\text{def}}{=} \quad & \begin{cases} X^\sharp[V \mapsto (\leq 0)] & \text{if } X^\sharp(W) \in \{0, (\leq 0)\} \\ X^\sharp & \text{otherwise} \end{cases} \\ & \cap^\sharp \\ & \begin{cases} X^\sharp[W \mapsto (\geq 0)] & \text{if } X^\sharp(V) \in \{0, (\geq 0)\} \\ X^\sharp & \text{otherwise} \end{cases} \end{aligned}$$

We omitted the case where the argument $X^\sharp$ is bottom $\bot^\sharp$, in which case the result is always $\bot^\sharp$: all our test operators are strict.

Convergence of iterations. Both sign domains are finite. They have no infinite increasing chain, so that any increasing iteration will naturally converge. Recall that the widening operator $\triangledown$ is used to stabilize the iterations $F^{\sharp i}(\bot^\sharp)$ of a sound abstraction $F^\sharp$ of a concrete function F into an over-approximation of the least fixpoint of F (Thm. 2.9). Here, it is sufficient to state $\triangledown_b \stackrel{\text{def}}{=} \sqcup_b^\sharp$. This ensures that the iteration $X_{n+1}^\sharp \stackrel{\text{def}}{=} X_n^\sharp \triangledown F^\sharp(X_n^\sharp)$ will be increasing, and thus will converge in finite time, without any hypothesis on $F^\sharp$

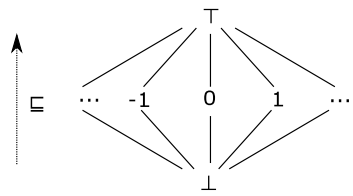


Figure 4.4: Hasse diagram for the constant domain.

— in particular, for the sake of generality, we do not require $F^\sharp$ to be monotonic, so that. while the iteration $X^\sharp_{n+1} \stackrel{\text{def}}{=} F^\sharp(X^\sharp_n)$ is not guaranteed to converge, the iteration $X^\sharp_{n+1} \stackrel{\text{def}}{=} X^\sharp_n \sqcup^\sharp F(X^\sharp_n)$ is.

4.3 The Constant Domain

Another simple static analysis is *constant propagation*. It allows detecting whether some variables or some expressions take only a single, fixed value during all possible executions, and enables compiler optimisation.

This analysis can be cast as a static analysis by Abstract Interpretation. We consider, as abstract domain: $\mathcal{B}^\sharp \stackrel{\text{def}}{=} \mathbb{I} \cup \{\bot^\sharp_b, \top^\sharp_b\}$, meaning that a variable is either a specific constant in $\mathbb{I}$, or has no possible value, $\bot^\sharp_b$, or may possibly take more than one value, $\top^\sharp_b$.

Order structure. The Hasse diagram for $\mathcal{B}^\sharp$ is presented in Fig. 4.4. This corresponds to a very flat, yet infinite complete lattice. We can define the lub and glb as follows:

$$x \sqcup^\sharp_b y \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x = y \\ x & \text{if } y = \bot^\sharp_b \\ y & \text{if } x = \bot^\sharp_b \\ \top^\sharp_b & \text{otherwise} \end{cases} \qquad x \sqcap^\sharp_b y \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x = y \\ x & \text{if } y = \top^\sharp_b \\ y & \text{if } x = \top^\sharp_b \\ \bot^\sharp_b & \text{otherwise} \end{cases}$$

Moreover, the constant domain enjoys a Galois connection:

$$\alpha_b(C) \stackrel{\text{def}}{=} \begin{cases} \bot^\sharp_b & \text{if } C = \emptyset \\ c & \text{if } C = \{c\} \\ \top^\sharp_b & \text{otherwise} \end{cases} \qquad \begin{array}{lcl} \gamma_b(\bot^\sharp_b) & \stackrel{\text{def}}{=} & \emptyset \\ \gamma_b(c) & \stackrel{\text{def}}{=} & \{c\} \\ \gamma_b(\top^\sharp_b) & \stackrel{\text{def}}{=} & \mathbb{I} \end{array}$$

Abstract operators. As for the sign domains, we set $\cup^\sharp_b \stackrel{\text{def}}{=} \sqcup^\sharp_b$ and $\cap^\sharp_b \stackrel{\text{def}}{=} \sqcap^\sharp_b$. The intersection $\cap^\sharp_b$ is exact. The join $\cup^\sharp_b$ is optimal, but not exact: the join of two different constants is $\top^\sharp_b$, which represents all possible values, $\mathbb{I}$.

Figure 4.5 presents a few representative examples of abstract operators: $+^\sharp_b$ and $\times^\sharp_b$. Most of the times, we simply apply the corresponding arithmetic operation on constant

$+$	$\bot$	c	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
c'	$\bot$	$c+c'$	$\top$
$\top$	$\top$	$\top$	$\top$

$\times$	$\bot$	0	$c \neq 0$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	0	0
$c' \neq 0$	$\bot$	0	$c \times c'$	$\top$
$\top$	$\top$	0	$\top$	$\top$

Figure 4.5: Some abstract operators in the constant domain.

arguments, and revert to $\bot_b^\sharp$ and $\top_b^\sharp$ if one of the arguments is not constant. An exception is multiplication, as $0 \times_b^\sharp \top_b^\sharp = 0$. Abstracting constants is straightforward: $c_b^\sharp \stackrel{\text{def}}{=} c$ and $[c_1, c_2]_b^\sharp \stackrel{\text{def}}{=} c_1$ when $c_1 = c_2$, and $\top_b^\sharp$ otherwise.

We abstract tests similarly as in the sign domains, by handling precisely two simple cases and reverting to the identity for all the other cases:

$$\mathbb{C}^\sharp[\![V = c]\!] X^\sharp \stackrel{\text{def}}{=} \begin{cases} \bot_b^\sharp & \text{if } X^\sharp(V) \notin \{c, \top_b^\sharp\} \\ X^\sharp[V \mapsto c] & \text{otherwise} \end{cases}$$

$$\mathbb{C}^\sharp[\![V = W + c]\!] X^\sharp \stackrel{\text{def}}{=} \begin{cases} X^\sharp & \text{if } X^\sharp(W) \in \{\bot_b^\sharp, \top_b^\sharp\} \\ \mathbb{C}^\sharp[\![V = X^\sharp(W) + c]\!] X^\sharp & \text{otherwise} \end{cases} \cap^\sharp \begin{cases} X^\sharp & \text{if } X^\sharp(V) \in \{\bot_b^\sharp, \top_b^\sharp\} \\ \mathbb{C}^\sharp[\![W = X^\sharp(V) - c]\!] X^\sharp & \text{otherwise} \end{cases}$$

The constant domain is infinite in width but very flat. As fixpoint iterations follow increasing chains, and the domain has no infinite increasing chain, we can enforce convergence by stating simply $\nabla_b \stackrel{\text{def}}{=} \sqcup_b^\sharp$.

Example 4.2 (Constant analysis)**.** *Consider the following very simple program:*

```
X ← 0; Y ← 10;
while X < 100 do
    Y ← Y − 3;
    X ← X + Y; ℓ1
    Y ← Y + 3
done
```

An analysis with the non-relational constant domain in denotational style, following Sect. 3.5, will iterate the loop twice. At the first loop iteration, at program point $\ell 1$, we would get $X = 10$, $Y = 7$. During the second iteration, we join at the loop head abstract states where X can have two different values: 0 and 10, hence X is not constant. We get at program point $\ell 1$ that $X = \top_b^\sharp$, $Y = 7$. The result is stable after the second iteration, so that $X = \top_b^\sharp$, $Y = 7$ is a program invariant at $\ell 1$: we have proved that Y equals 7 for every execution whenever it reaches $\ell 1$.

While this result may seem straightforward, we point out that Y varies during the execution of the program: it has a different constant value at different program points,

which our analysis can capture as it is flow-sensitive. More importantly, the value 7 *found at* $\ell 1$ *is not a literal constant of the program, i.e., the analysis is actually semantic and not syntactic.* ♦

4.4 The Constant Set Domain

There are many circumstances where a variable is not constant, but can only take few possible values. We can easily extend the constant domain to track several values instead of one. However, we must be wary of loops generating a large or even unbounded number of different values.

Bounded set domain. A first idea to solve this problem is to bound *a priori* the number of constants for each variable with a value k chosen before the analysis is started. We use, as abstract domain, $\mathcal{B}^\sharp \stackrel{\text{def}}{=} \mathcal{P}_{\leq k}(\mathbb{I}) \cup \{\top_b^\sharp\}$, where $\mathcal{P}_{\leq k}(\mathbb{I})$ denotes the sets containing at most k numbers from $\mathbb{I}$. Note that $\mathcal{P}_{\leq k}(\mathbb{I})$ contains $\emptyset$, so that there is no need to add a $\bot_b^\sharp$ element in $\mathcal{B}^\sharp$. We keep $\top_b^\sharp$ in order to represent $\mathbb{I}$, but also to abstract any set of values with more than k elements. We then have a straightforward order relation and a Galois connection:

$$X^\sharp \sqsubseteq Y^\sharp \iff (Y^\sharp = \top_b^\sharp) \vee (X^\sharp, Y^\sharp \neq \top_b^\sharp \wedge X^\sharp \subseteq Y^\sharp)$$

$$\gamma_b(X^\sharp) \stackrel{\text{def}}{=} \begin{cases} \mathbb{I} & \text{if } X^\sharp = \top_b^\sharp \\ X^\sharp & \text{otherwise} \end{cases}$$

$$\alpha_b(C) \stackrel{\text{def}}{=} \begin{cases} C & \text{if } |C| \leq k \\ \top_b^\sharp & \text{otherwise} \end{cases}$$

and all the operators are similar to the concrete ones, which operate on sets, with the following modifications:

- we must handle the case where one or both arguments is $\top_b^\sharp$, in which case we generally return $\top_b^\sharp$, except in some special cases, such as $\top_b^\sharp +_b^\sharp \emptyset \stackrel{\text{def}}{=} \emptyset$ and $\top_b^\sharp \times \{0\} \stackrel{\text{def}}{=} \{0\}$;
- whenever a set reaches a size larger than k, we return $\top_b^\sharp$ instead.

For instance:

$$X^\sharp \sqcup_b^\sharp Y^\sharp \stackrel{\text{def}}{=} \begin{cases} \top_b^\sharp & \text{if } X^\sharp = \top_b^\sharp \text{ or } Y^\sharp = \top_b^\sharp \\ \top_b^\sharp & \text{else if } |X^\sharp \cup Y^\sharp| > k \\ X^\sharp \cup Y^\sharp & \text{otherwise} \end{cases}$$

$$X^\sharp +_b^\sharp Y^\sharp \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } X^\sharp = \emptyset \text{ or } Y^\sharp = \emptyset \\ \top_b^\sharp & \text{else if } X^\sharp = \top_b^\sharp \text{ or } Y^\sharp = \top_b^\sharp \\ \top_b^\sharp & \text{else if } |\{\, x + y \mid x \in X^\sharp,\, y \in Y^\sharp \,\}| > k \\ \{\, x + y \mid x \in X^\sharp,\, y \in Y^\sharp \,\} & \text{otherwise} \end{cases}$$

This ensures that the sets never exceed k in size. Note that increasing chains have a maximum height of $k+2$: $\emptyset \sqsubseteq_b^\sharp \{c_1\} \sqsubseteq_b^\sharp \{c_1, c_2\} \sqsubseteq_b^\sharp \cdots \sqsubseteq_b^\sharp \{c_1, \ldots, c_k\} \sqsubseteq_b^\sharp \top_b^\sharp$. Hence, we can use, as widening, $\triangledown_b \stackrel{\text{def}}{=} \sqcup_b^\sharp$.

Unbounded set domain. A second idea is to allow sets of any size, as long as they are finite, so that we can represent them in extension in a computer. We note: $\mathcal{B}^\sharp \stackrel{\text{def}}{=} \mathcal{P}_{finite}(\mathbb{I}) \cup \{\top_b^\sharp\}$, We still need $\top_b^\sharp$ to abstract infinite sets, such as $\mathbb{I}$.

The operators are the same as in the previous constant set domain, except that we no longer systematically set abstract elements to $\top_b^\sharp$ after they reach a certain size, for instance:

$$X^\sharp \sqcup_b^\sharp Y^\sharp \stackrel{\text{def}}{=} \begin{cases} \top_b^\sharp & \text{if } X^\sharp = \top_b^\sharp \text{ or } Y^\sharp = \top_b^\sharp \\ X^\sharp \cup Y^\sharp & \text{otherwise} \end{cases}$$

$$X^\sharp +_b^\sharp Y^\sharp \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } X^\sharp = \emptyset \text{ or } Y^\sharp = \emptyset \\ \top_b^\sharp & \text{else if } X^\sharp = \top_b^\sharp \text{ or } Y^\sharp = \top_b^\sharp \\ \{\, x + y \mid x \in X^\sharp,\, y \in Y^\sharp \,\} & \text{otherwise} \end{cases}$$

However, our domain now has infinite increasing chains, and a proper widening is necessary to ensure the convergence of iterates. A simple idea is to ensure that the size of a set output by the widening never exceeds a user-defined constant k by using, as widening, the join $\cup_b^\sharp$ operator from the bounded set domain, which replaces large sets with $\top_b^\sharp$. Hence, our widening is:

$$X^\sharp \triangledown_b Y^\sharp \stackrel{\text{def}}{=} \begin{cases} \top_b^\sharp & \text{if } X^\sharp = \top_b^\sharp \text{ or } Y^\sharp = \top_b^\sharp \\ \top_b^\sharp & \text{else if } |X^\sharp \cup Y^\sharp| > k \\ X^\sharp \cup Y^\sharp & \text{otherwise} \end{cases}$$

While we still use a bound k for set sizes, this bound only concerns, unlike the case of the bounded set domain, the invariants at program points where a widening is applied, while sets can be larger at other program points. We gain in expressiveness by lifting the restriction on set sizes, only enforcing the minimum constraints needed to ensure termination.

The idea of using powersets to construct, from one domain, a more expressive one, is a general idea. It can be applied to other domains than the constant one. The generic construction, called *powerset completion*, will be described in details and applied to more expressive domains in Sect. 6.3.

4.5 The Interval Domain

We already mentioned interval abstractions several times in the previous chapters. Based on interval arithmetic, introduced by Moore [1966] for numeric analysis, and adapted to static analysis since the beginning of Abstract Interpretation by Cousot and Cousot [1976], it remains one of the most popular abstract domains because it is, as all non-relational

domains, both simple and inexpensive, and yet it can express and infer valuable properties for program verification.

The interval domain abstracts the set of possible values of a variable as an interval. The abstract values are either non-empty intervals with finite or infinite bounds, or $\bot_b^\sharp$:

$$\mathcal{B}^\sharp \stackrel{\text{def}}{=} \{\, [a,b] \mid a \in \mathbb{I} \cup \{-\infty\},\ b \in \mathbb{I} \cup \{+\infty\},\ a \leq b \,\} \cup \{\bot_b^\sharp\} \tag{4.5}$$

The greatest element can be represented as the interval $[-\infty, +\infty]$, so there is no need to add a separate $\top_b^\sharp$ element; it will be used to denote $[-\infty, +\infty]$.

4.5.1 Order Structure

We have a lattice structure on $\mathcal{B}^\sharp$ and a concretization function:

$$\begin{array}{rcll}
[a,b] \sqsubseteq_b^\sharp [c,d] & \stackrel{\text{def}}{\iff} & (a \geq c) \wedge (b \leq d) & \\
[a,b] \sqcup_b^\sharp [c,d] & \stackrel{\text{def}}{=} & [\min(a,c), \max(b,d)] & \\
[a,b] \sqcap_b^\sharp [c,d] & \stackrel{\text{def}}{=} & \begin{cases} [\max(a,c), \min(b,d)] & \text{if } \max(a,c) \leq \min(b,d) \\ \bot_b^\sharp & \text{otherwise} \end{cases} & \\
\gamma_b(\bot_b^\sharp) & \stackrel{\text{def}}{=} & \emptyset & \\
\gamma_b([a,b]) & \stackrel{\text{def}}{=} & \{\, x \in \mathbb{I} \mid a \leq x \leq b \,\} &
\end{array}$$

Recall that we often omit the cases where one argument at least is $\bot_b^\sharp$, but these are straightforward: $\forall X^\sharp \in \mathcal{B}^\sharp\colon \bot_b^\sharp \sqsubseteq_b^\sharp X^\sharp$; $\bot_b^\sharp$ is neutral for $\sqcup_b^\sharp$ and absorbing for $\sqcap_b^\sharp$. The Hasse diagram of the lattice was presented in Fig. 2.3 for the case of integers.

When $\mathbb{I} = \mathbb{Z}$ or $\mathbb{I} = \mathbb{R}$, the lattice is complete: we can extend the lub and the glb to infinite families, respectively as $\sqcup_b^\sharp \{\, [a_i, b_i] \mid i \in I \,\} \stackrel{\text{def}}{=} [\min_{i \in I} a_i, \max_{i \in I} b_i]$, and $\sqcap_b^\sharp \{\, [a_i, b_i] \mid i \in I \,\} \stackrel{\text{def}}{=} [\max_{i \in I} a_i, \min_{i \in I} b_i]$ or $\bot_b^\sharp$ when the left bound is greater than the right bound. Indeed, the minimum and the maximum of an arbitrary (possibly infinite) set always exists in $\mathbb{I} \cup \{\pm\infty\}$.

Likewise, we can define an abstraction function so that our domain enjoys a Galois connection:

$$\alpha_b(X) \stackrel{\text{def}}{=} \begin{cases} \bot_b^\sharp & \text{if } X = \emptyset \\ [\min X, \max X] & \text{otherwise} \end{cases} \tag{4.6}$$

Note that this is not possible when $\mathbb{I} = \mathbb{Q}$ as the minimum and maximum is not always defined — consider, for instance, the set $\{\, x \in \mathbb{Q} \mid x^2 \leq 2 \,\}$, which has no maximum in $\mathbb{Q}$. In that case, the lattice is not complete, and some sets may not have a best abstraction.

4.5.2 Abstract Operators

As for the previous domains, we set $\cup_b^\sharp \stackrel{\text{def}}{=} \sqcup_b^\sharp$ and $\cap_b^\sharp \stackrel{\text{def}}{=} \sqcap_b^\sharp$. The intersection $\cap_b^\sharp$ is exact and the join $\cup_b^\sharp$ is optimal, but not exact. Consider for instance, for integer intervals, that $\gamma_b([0,0] \cup_b^\sharp [2,2]) = \gamma_b([0,2]) = \{0,1,2\}$, while $\gamma_b([0,0]) \cup \gamma_b([2,2]) = \{0\} \cup \{2\} = \{0,2\}$.

$$
\begin{array}{rcl}
c_b^\sharp & \stackrel{\text{def}}{=} & [c, c] \\
[c, c']_b^\sharp & \stackrel{\text{def}}{=} & [c, c'] \\
-_b^\sharp [a, b] & \stackrel{\text{def}}{=} & [-b, -a] \\
[a, b] +_b^\sharp [c, d] & \stackrel{\text{def}}{=} & [a + c, b + d] \\
[a, b] -_b^\sharp [c, d] & \stackrel{\text{def}}{=} & [a - d, b - c] \\
[a, b] \times_b^\sharp [c, d] & \stackrel{\text{def}}{=} & [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \\
[a, b] /_b^\sharp [c, d] & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{l}{\begin{cases} [\min(a/c, a/d), \max(b/c, b/d)] & \text{if } 1 \leq c \\ [\min(b/c, b/d), \max(a/c, a/d)] & \text{if } d \leq -1 \\ ([a, b] /_b^\sharp ([c, d] \cap_b^\sharp [1, +\infty])) \cup_b^\sharp ([a, b] /_b^\sharp ([c, d] \cap_b^\sharp [-\infty, -1])) & \text{otherwise} \end{cases}}
\end{array}
$$

Figure 4.6: Abstract arithmetic operators in the interval domain.

Value operators. Figure 4.6 presents the arithmetic operators in the interval domain. We omitted the cases where at least one argument is $\bot_b^\sharp$ as the result is always $\bot_b^\sharp$. Most definitions are straightforward: $c_b^\sharp$, $[c_1, c_2]_b^\sharp$, $+_b^\sharp$, and $-_b^\sharp$ give extract abstractions. Multiplication $\times_b^\sharp$ is exact on rationals and reals, but only optimal on integers — consider for instance $[0, 1] \times_b^\sharp [2, 2] = [0, 2]$ while, in the concrete, $[0, 1] \times [2, 2] = \{0, 2\}$. Division is more involved: we first consider the case where the divisor is necessarily positive or when it is necessarily negative, in which case the result is representable as an interval; when the divisor contains both positive and negative values, we split it into its positive and negative parts, perform both divisions independently, and join the results with $\cup_b^\sharp$ to get a single, approximate interval. Because division in the concrete does not always return a convex set, $/_b^\sharp$ is not exact, even in $\mathbb{R}$ and $\mathbb{Q}$.

An additional complexity comes from handling infinities. We extend the natural operations $+$, $-$, $\times$, $/$ used on bounds from $\mathbb{I}$ to $\mathbb{I} \cup \{\pm\infty\}$ as follows:

- $\forall c \in \mathbb{I} \cup \{+\infty\}$: $c + (+\infty) = +\infty$, $\forall c \in \mathbb{I} \cup \{-\infty\}$: $c + (-\infty) = -\infty$ as expected; note that we always add either two upper bounds or two lower bounds, so that we never add $+\infty$ to $-\infty$, which would be undefined; the subtraction operator $-$ is similar;
- $\forall c > 0$: $c \times (+\infty) = +\infty$, $c \times (-\infty) = -\infty$ while $\forall c < 0$: $c \times (+\infty) = -\infty$, $c \times (-\infty) = +\infty$, following the rule of signs, as expected;
- $0 \times (+\infty) = 0 \times (-\infty) = 0$, which is non-standard; this is required to handle cases such as $[1, +\infty] \times_b^\sharp [0, 1] = [0, +\infty]$; a intuitive interpretation is to consider that $0 \times M = 0$ for any finite M, so that, when M tends towards $+\infty$ or $-\infty$, by continuity, we keep 0 as result;
- division $/$ also follows the rule of signs, such as $\forall c > 0$: $(+\infty)/c = +\infty$;
- we finally state $\forall c$: $c/(+\infty) = c/(-\infty) = 0$, including $(+\infty)/(+\infty) = 0$; the intuition

here is that $a/b = a \times (1/b)$; hence, $(+\infty)/(+\infty) = (+\infty) \times (1/(+\infty)) = (+\infty) \times 0 = 0$, for compatibility with the definition of multiplications.

Condition abstraction. To abstract conditions, we can handle simple cases precisely, as in the previous domains, and revert to the identity for others:

$$\mathbb{C}^\sharp[\![\, V \leq v \,]\!] X^\sharp \stackrel{\text{def}}{=} \begin{cases} X^\sharp[V \mapsto [a, \min(b, v)] & \text{if } a \leq v \\ \bot^\sharp & \text{if } a > v \end{cases}$$

$$\mathbb{C}^\sharp[\![\, V \leq W \,]\!] X^\sharp \stackrel{\text{def}}{=} \begin{cases} X^\sharp[\ V \mapsto [a, \min(b, d)], & \text{if } a \leq d \\ \quad W \mapsto [\max(a, c), d]\ ] & \\ \bot^\sharp & \text{if } a > d \end{cases}$$

where $X^\sharp(V) = [a, b]$ and $X^\sharp(W) = [c, d]$

In the test $V \leq W$, we use the upper bound of W to refine the upper bound of V, since any value of V which is greater than W's upper bound cannot be part of a state satisfying the test. Similarly, we use the lower bound of V to refine the lower bound of W. We will see in Sect. 4.6 a more general test abstraction that can handle arbitrary expressions.

Standard widening. The interval domain has infinite strictly increasing chains, such as for instance $[0, 1]$, $[0, 2]$, ..., $[0, n]$, ... Hence, a proper widening is required to enforce convergence. The standard interval widening consists in replacing any unstable upper bound with $+\infty$, and any unstable lower bound with $-\infty$:

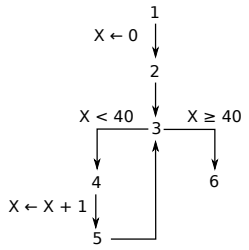
$$[a, b] \triangledown_b [c, d] \stackrel{\text{def}}{=} \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right] \tag{4.7}$$

Once a bound is pushed to infinity, it becomes stable, as the widening never tightens bounds. Each bound of each variable is handled independently. The widening $\triangledown$ on $\mathcal{D}^\sharp$, derived from $\triangledown_b$, will converge in at most $2|\mathbb{V}|$ iterations.

4.5.3 Example Analysis

Example 4.3 (Interval analysis)**.** *Consider a simple loop "$X \leftarrow 0$;* **while** *$X < 40$* **do** *$X \leftarrow X+1$* **done***" to be analyzed using the equation-based semantics (Sect. 3.4) over the interval domain. The control-flow graph generated by the generic method of Fig. 3.9 is presented in Fig. 4.7.(a). In Fig. 4.7.(b), we follow (3.4) to give at each program point ℓ in $[1, 6]$ the abstract iterates, showing only the most interesting ones. Moreover, following Sect. 3.5, we choose to apply the widening at the loop head, that is, control point 2.*

- *At iterate 0, the abstract value of X is $[-\infty, +\infty]$ at point 1 and $\bot_b^\sharp$ elsewhere.*
- *Iterates 1 to 4 propagate the non-$\bot_b^\sharp$ value from point 1 to point 2 after the assignment $X \leftarrow 0$, then to point 3 after the test $X < 40$, then to point 4 after the incrementation $X \leftarrow X + 1$. We only show the end-result at iteration 4 in the figure.*



(a)

ℓ	$\mathcal{X}_\ell^{\sharp 0}$	$\mathcal{X}_\ell^{\sharp 4}$	$\mathcal{X}_\ell^{\sharp 5}$	$\mathcal{X}_\ell^{\sharp 7}$	$\mathcal{X}_\ell^{\sharp 8}$
1	$\top$	$\top$	$\top$	$\top$	$\top$
2	$\bot$	0	0	0	0
3 $\triangledown$	$\bot$	0	$[0,+\infty]$	$[0,+\infty]$	$[0,+\infty]$
4	$\bot$	0	0	$[0,39]$	$[0,39]$
5	$\bot$	1	1	$[1,40]$	$[1,40]$
6	$\bot$	$\bot$	$\bot$	$[40,+\infty]$	$[40,+\infty]$

(b)

Figure 4.7: Interval analysis example: (a) control-flow graph of a simple loop incrementing X, and (b) abstract iterations.

- *Iterate 5 shows the first application of the widening as the interval* $[0,0]$ *gets increased to* $[0,1]$ *after getting some loop feed-back from* $X = 1$ *at point 5. We get* $\mathcal{X}_3^{\sharp 5} \stackrel{\text{def}}{=} \mathcal{X}_3^{\sharp 4} \triangledown_b (\mathcal{X}_2^{\sharp 4} \sqcup_b^\sharp \mathcal{X}_5^{\sharp 4})$*, i.e.,* $[0,0] \triangledown_b ([0,0] \sqcup_b^\sharp [1,1]) = [0,0] \triangledown_b [0,1] = [0,+\infty]$.

- *This information gets propagated in iterations 6 and 7 to enlarge the value of* X *at point 4 after the test* $X < 40$*, and at point 5 after the incrementation* $X \leftarrow X + 1$*. Moreover, starting from iteration 6, the loop invariant* $[0,+\infty]$ *also passes the test* $X \geq 40$ *to give, at point* 6*,* $[40,+\infty]$.

- *Iterate 8 gives the exact same result as iterate 7, hence, we have reached an abstract invariant.*

On this example, the interval analysis proves that X *is always within* $[0,40]$ *in the loop body, and greater than* 40 *after the loop. Note, in particular, that the computation only requires 8 iterations, far less than the 40 iterations in the concrete semantics. More importantly, the number of abstract iterations does not depend on the constant* 40*: we could have a far larger number of iterations in the concrete and still only* 8 *abstract iterations.*

Our result is not the most precise invariant as, actually, $X \in [0,40]$ *also holds at point 3 and, when the loop stops,* $X = 40$ *at point 6. We will see in Sect. 4.7 different iteration strategies to improve this result, as well as the influence of the choice of widening points.* ♦

4.6 Advanced Abstract Tests

We presented in Fig. 4.1 a generic method to abstract assignments in non-relational domains, using evaluation in the abstract value domain by induction on the syntax. The case of tests is slightly different: instead of computing the value of an expression given abstract variable values, we assume some information about the value of two expressions, such as $e_1 \leq e_2$, and wish to refine the abstract values of variables. This is a classic *constraint programming* problem, and we can employ a constraint solving method to solve it.

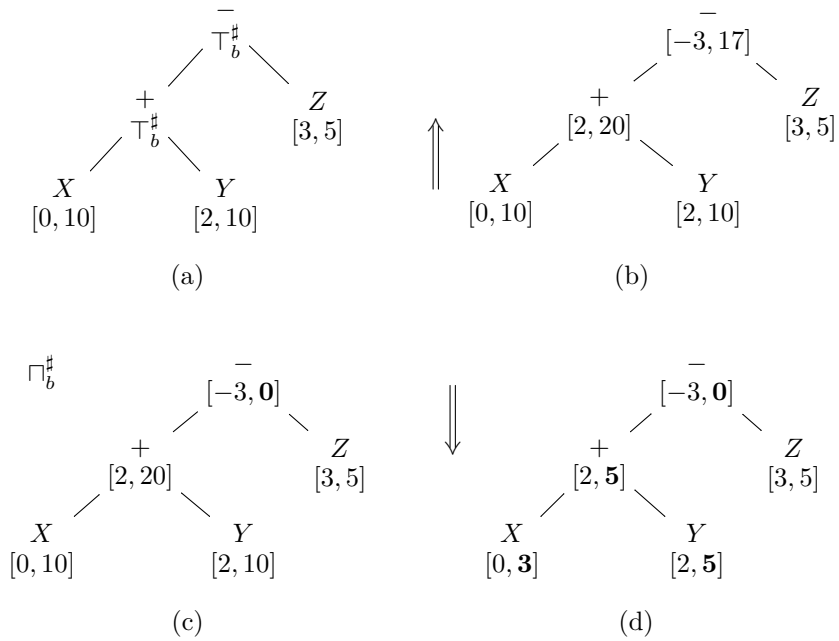


Figure 4.8: Abstract test $\mathbb{C}^\sharp[\![\,(X+Y)-Z\leq 0\,]\!]$ in the interval domain, starting from the abstract state $[X\mapsto[0,10],\ Y\mapsto[2,10],\ Z\mapsto[3,5]]$.

4.6.1 Propagation Algorithm

To simplify, we consider tests of the form $e\leq 0$; other tests can be put into this, or a similar form. Our algorithm is based on HC4-revise, introduced by Benhamou et al. [1999]. It operates in three steps. It is illustrated in Fig. 4.8 on the evaluation of $\mathbb{C}^\sharp[\![\,(X+Y)-Z\leq 0\,]\!]$ in the interval domain, starting from an abstract state $[X\mapsto[0,10],\ Y\mapsto[2,10],\ Z\mapsto[3,5]]$.

Firstly, the expressions are evaluated by induction on the syntax tree, bottom-up, from leaves (variables and constants) to the expression root, similarly to $\mathbb{E}^\sharp[\![\,e\,]\!]$ in Fig. 4.1; however, we remember the abstract value at each syntax tree node. This is illustrated in Fig. 4.8 as steps (a)–(b).

Secondly, the interval at the root, here $[-3,17]$ is intersected with the condition for the test to be true, in this case $[-\infty,0]$, as the result of the expression should be negative. This yields $[-3,0]$ in Fig. 4.8.(c).

Thirdly, this information is propagated *backwards*, top-down towards the leaves, as shown in Fig. 4.8.(d). For instance, $[-3,17]$ spawned from $[2,20]-[3,5]$ but, for $[2,20]-[3,5]$ to be actually included in $[-3,0]$, it is necessary for the left-hand side to be less than 5, so that we replace $[2,20]$ with $[2,5]$. This interval, which corresponds to $[0,10]+[2,10]$ is propagated backward to refine $[0,10]$ into $[0,3]$ and $[2,10]$ into $[2,5]$. Hence, the output of the test is the abstract state $[X\mapsto[0,3],\ Y\mapsto[2,5],\ Z\mapsto[3,5]]$.

4.6.2 Backward Abstract Value Operators

Our algorithm manipulates abstract values, and can be applied to any non-relational domain, and not only intervals. The first part of the algorithm uses the existing abstract arithmetic operators $+^\sharp_b$, etc. (Def. 4.1). The second and third parts use backward versions that, given the original arguments, and some novel information on the result, return refined versions of the arguments, removing as many values as possible that cannot contribute to the new expected result. Hence, we require the following operators, denoted with a leftward arrow to distinguish them from the regular abstract operators:

- $\overleftarrow{\leq} 0^\sharp_b : \mathcal{B}^\sharp \to \mathcal{B}^\sharp$ to add the information that an interval is negative;
- backward unary operators: $\overleftarrow{-}^\sharp_b : \mathcal{B}^\sharp \times \mathcal{B}^\sharp \to \mathcal{B}^\sharp$;
- backward binary operators: $\overleftarrow{+}^\sharp_b, \overleftarrow{-}^\sharp_b, \overleftarrow{\times}^\sharp_b, \overleftarrow{/}^\sharp_b : \mathcal{B}^\sharp \times \mathcal{B}^\sharp \times \mathcal{B}^\sharp \to \mathcal{B}^\sharp \times \mathcal{B}^\sharp$.

with the following soundness conditions:

$$\begin{array}{l} X^{\sharp\prime} = \overleftarrow{\leq} 0^\sharp_b(X^\sharp) \Longrightarrow \\ \quad \{\, x \in \gamma_b(X^\sharp) \mid x \leq 0 \,\} \subseteq \gamma_b(X^{\sharp\prime}) \end{array}$$

$$\begin{array}{l} X^{\sharp\prime} = \overleftarrow{-}^\sharp_b(X^\sharp, R^\sharp) \Longrightarrow \\ \quad \{\, x \mid x \in \gamma_b(X^\sharp),\ -x \in \gamma_b(R^\sharp) \,\} \subseteq \gamma_b(X^{\sharp\prime}) \end{array}$$

$$\begin{array}{l} (X^{\sharp\prime}, Y^{\sharp\prime}) = \overleftarrow{+}^\sharp_b(X^\sharp, Y^\sharp, R^\sharp) \Longrightarrow \\ \quad \{\, x \in \gamma_b(X^\sharp) \mid \exists y \in \gamma_b(Y^\sharp),\ x + y \in \gamma_b(R^\sharp) \,\} \subseteq \gamma_b(X^{\sharp\prime}) \\ \quad \{\, y \in \gamma_b(Y^\sharp) \mid \exists x \in \gamma_b(X^\sharp),\ x + y \in \gamma_b(R^\sharp) \,\} \subseteq \gamma_b(Y^{\sharp\prime}) \end{array}$$

and similarly for the other binary operators, which state that the new inputs $X^{\sharp\prime}, Y^{\sharp\prime}$ still cover all possible ways to satisfy a condition (for ≤ 0) or to generate the new output $R^\sharp$ (for $+$, $-$, etc.).

In the presence of a Galois connection (α_b, γ_b), we can derive systematically the best refinement operators using α_b, for instance:

$$\overleftarrow{-}^\sharp_b(X^\sharp, R^\sharp) \stackrel{\text{def}}{=} \alpha_b(\{\, x \mid x \in \gamma_b(X^\sharp),\ -x \in \gamma_b(R^\sharp) \,\})$$

Alternatively, we can easily synthesize sufficiently precise backward abstract operators from the forward ones used in standard evaluation. Consider, for instance, the case of $\overleftarrow{+}^\sharp_b(X^\sharp, Y^\sharp, R^\sharp)$. We observe that, if $R = X + Y$, then $X = R - Y$, and we can use the new value of R and the previously known value of Y to get the new value of X, i.e., $X^{\sharp\prime} \stackrel{\text{def}}{=} X^\sharp \cap^\sharp_b (R^\sharp -^\sharp_b Y^\sharp)$. Based on such identities, we can derive all backwards operators as presented in Fig. 4.9. The division takes extra care as:

- $R = X/Y$ does not imply $Y = X/R$, but rather $(Y = X/R) \vee (Y = 0)$ as we have to consider the case of a division by zero that does not produce a value in R;

$$
\begin{array}{lcl}
\overleftarrow{\leq}\, 0^\sharp_b(X^\sharp) & \stackrel{\text{def}}{=} & X^\sharp \cap^\sharp_b [-\infty, 0]^\sharp_b \\
\overleftarrow{-}{}^\sharp_b(X^\sharp, R^\sharp) & \stackrel{\text{def}}{=} & X^\sharp \cap^\sharp_b (-^\sharp_b R^\sharp) \\
\overleftarrow{+}{}^\sharp_b(X^\sharp, Y^\sharp, R^\sharp) & \stackrel{\text{def}}{=} & (X^\sharp \cap^\sharp_b (R^\sharp -^\sharp_b Y^\sharp),\ Y^\sharp \cap^\sharp_b (R^\sharp -^\sharp_b X^\sharp)) \\
\overleftarrow{-}{}^\sharp_b(X^\sharp, Y^\sharp, R^\sharp) & \stackrel{\text{def}}{=} & (X^\sharp \cap^\sharp_b (R^\sharp +^\sharp_b Y^\sharp),\ Y^\sharp \cap^\sharp_b (X^\sharp -^\sharp_b R^\sharp)) \\
\overleftarrow{\times}{}^\sharp_b(X^\sharp, Y^\sharp, R^\sharp) & \stackrel{\text{def}}{=} & (X^\sharp \cap^\sharp_b (R^\sharp /^\sharp_b Y^\sharp),\ Y^\sharp \cap^\sharp_b (R^\sharp /^\sharp_b X^\sharp)) \\
\overleftarrow{/}{}^\sharp_b(X^\sharp, Y^\sharp, R^\sharp) & \stackrel{\text{def}}{=} & (X^\sharp \cap^\sharp_b (S^\sharp \times^\sharp_b Y^\sharp),\ Y^\sharp \cap^\sharp_b ((X^\sharp /^\sharp_b S^\sharp) \cup^\sharp_b [0,0]^\sharp_b)) \\
\text{where } S^\sharp & \stackrel{\text{def}}{=} & \begin{cases} R^\sharp & \text{if } \mathbb{I} \neq \mathbb{Z} \\ R^\sharp +^\sharp_b [-1,1]^\sharp_b & \text{if } \mathbb{I} = \mathbb{Z} \end{cases}
\end{array}
$$

Figure 4.9: Synthesizing abstract backward operators from forward ones.

- when considering integers, $R = X/Y$ rounds its result, so, we first have the perform the inverse of the rounding operation, which we model as replacing R with $S \stackrel{\text{def}}{=} R + [-1, 1]$, before proceeding with $X = Y \times S$ and $Y = X/S$.

Note that these formulas may not provide the best backward abstractions; indeed, we are combining several abstract operators, so that, even if they are individually optimal, their combination may not be optimal.

Applying these formulas to the case of intervals gives us, for instance:

$$
\begin{array}{lcl}
\overleftarrow{\leq}\, 0^\sharp_b([a,b]) & \stackrel{\text{def}}{=} & \begin{cases} [a, \min(b,0)] & \text{if } a \leq 0 \\ \bot^\sharp_b & \text{otherwise} \end{cases} \\
\overleftarrow{-}{}^\sharp_b([a,b],[r,s]) & \stackrel{\text{def}}{=} & [a,b] \cap^\sharp_b [-s,-r] \\
\overleftarrow{+}{}^\sharp_b([a,b],[c,d],[r,s]) & \stackrel{\text{def}}{=} & ([a,b] \cap^\sharp_b [r-d, s-c], [c,d] \cap^\sharp_b [r-b, s-a])
\end{array}
$$

4.6.3 Local Iterations

In the concrete world, a test is idempotent, as testing again some condition immediately after testing it a first time gives the identity. However, this may not be the case in the abstract: iterating our algorithm several times may improve the precision. This is particularly the case if some variable X appears twice or more in the expression. In that case, the refinement process will provide several distinct abstract values for X. They are all valid, so that we can store their abstract intersection into the abstract memory state. As we have refined the leaves of the expression tree, running the bottom-up evaluation may provide more precise abstract values which, in turn, will refine the leaves some more, i.e., the abstract memory state.

This also applies to the case where a variable appears in different atomic tests combined with the boolean operators $\wedge$ and $\vee$. Recall that this case is handled by induction on the syntax as, for instance, $\mathbb{C}^\sharp[\![\, c_1 \vee c_2 \,]\!] R^\sharp \stackrel{\text{def}}{=} \mathbb{C}^\sharp[\![\, c_1 \,]\!]\, R^\sharp \cup^\sharp \mathbb{C}^\sharp[\![\, c_2 \,]\!] R^\sharp$. The idea of iterating, for a few times, or until a fixpoint occurs — possibly using extrapolation operators for

decreasing sequences that we will describe in Sect. 4.7.2 — was proposed by Granger [1992] as *local iterations*.

Example 4.4 (Local iterations)**.** *We consider as example the case of a complex boolean condition* $(X \geq Y) \wedge (Z \geq X)$*, in the interval environment* $[X \mapsto [0,10], Y \mapsto [5,15], Z \mapsto [-10,10]]$*. This is modeled as* $\mathbb{C}^\sharp[\![X \geq Y]\!]R^\sharp \cap^\sharp \mathbb{C}^\sharp[\![Z \geq X]\!]R^\sharp$*:*

- *The first application, in parallel, of both atomic tests* $X \geq Y$ *and* $Z \geq X$ *will give respectively:* $[X \mapsto [5,10], Y \mapsto [5,15], Z \mapsto [-10,10]]$*, refining* X*, and* $[X \mapsto [0,10], Y \mapsto [5,15], Z \mapsto [0,10]]$*, refining* Z*.*
- *Their combination through* $\cap^\sharp$ *gives* $[X \mapsto [5,10], Y \mapsto [5,15], Z \mapsto [0,10]]$*.*
- *As the lower bound of* X *has changed, the test* $Z \geq X$ *can gather more information from this new bound. Indeed, we get, after* $\mathbb{C}^\sharp[\![Z \geq X]\!]$*, the abstract state:* $[X \mapsto [5,10], Y \mapsto [5,15], Z \mapsto [5,10]]$*, refining* Z *again.*
- *Further applications of atomic test operators do not change the result; we have reached a fixpoint in two iterations.* ♦

4.7 Advanced Iteration Techniques

Inferring loop invariants is a major challenge in program analysis. Abstract interpretation solves this problem using iteration with widenings. We saw a simple example of this process, which gave rather coarse results in Fig. 4.7. Many techniques have been developed to improve on this. We will present of few of the simplest, most popular techniques.

4.7.1 Choice of Widening Points

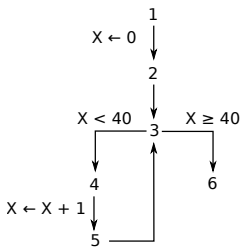
Firstly, we justify our choice of using loop heads as widening points. We consider again the analysis of "$X \leftarrow 0$; **while** $X < 40$ **do** $X \leftarrow X + 1$ **done**" from Fig. 4.7, but apply widening at program point 4 instead of 3. The result is given in Fig. 4.10. Now, the result at program points 4 and 5 are respectively $[0, +\infty]$ and $[1, +\infty]$, instead of $[0, 39]$ and $[1, 40]$, which is far less precise.

The intuitive explanation is that, when widening at program point 3, the propagation of the loss of precision from this point is limited as the following instructions, either staying in or exiting the loop, are tests, which add some bounds to X. However, when the widening occurs at program point 4, it gets propagated by the incrementation $X \leftarrow X + 1$ into program point 5, and then 3, without any intervening test to bound the intervals.

This phenomenon was empirically shown to be significant by Bourdoncle [1993b], hence we follow his recommendation by widening at loop heads.

4.7.2 Decreasing Iterations

A first idea in order to improve the precision of loop analysis is to start from the result of the iteration with widening after stabilization, and try and refine it *a posteriori*.



(a)

ℓ	$\mathcal{X}_\ell^{\sharp 0}$	$\mathcal{X}_\ell^{\sharp 4}$	$\mathcal{X}_\ell^{\sharp 5}$	$\mathcal{X}_\ell^{\sharp 6}$	$\mathcal{X}_\ell^{\sharp 8}$
1	$\top$	$\top$	$\top$	$\top$	$\top$
2	$\bot$	0	0	0	0
3	$\bot$	0	$[0,1]$	$[0,1]$	$[0,+\infty]$
4 $\triangledown$	$\bot$	0	0	$[0,+\infty]$	$[0,+\infty]$
5	$\bot$	1	1	1	$[1,+\infty]$
6	$\bot$	$\bot$	$\bot$	$\bot$	$[40,+\infty]$

(b)

Figure 4.10: Interval analysis of Fig. 4.7 with a different widening point.

Recall that we wish to approximate some concrete fixpoint $\operatorname{lfp} F$, so we iterate $X^\sharp \mapsto X^\sharp \triangledown F^\sharp(X^\sharp)$ until finding $X^\sharp$ such that $X^\sharp = X^\sharp \triangledown F^\sharp(X^\sharp)$. For such a $X^\sharp$, by definition of widening (Def. 2.17), $F^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$, and we know, from Thm. 2.8, that any abstract postfixpoint $X^\sharp$ over-approximates $\operatorname{lfp} F$: $\operatorname{lfp} F \subseteq \gamma(X^\sharp)$. Applying F, we get, on the one hand, $F(\operatorname{lfp} F) = \operatorname{lfp} F$ and, on the other hand, $F(\gamma(X^\sharp)) \subseteq \gamma(F^\sharp(X^\sharp))$, hence, $F^\sharp(X^\sharp)$ is also a sound approximation of $\operatorname{lfp} F$ and, as $F^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$, it can only improve on $X^\sharp$.

We can continue iterating $F^\sharp$: we always get sound approximations of $\operatorname{lfp} F$, but the abstract sequence is not necessarily decreasing, and does not necessarily terminate. To solve the first problem, we can iterate:

$$Y^{\sharp 0} \stackrel{\text{def}}{=} X^\sharp \qquad Y^{\sharp n+1} \stackrel{\text{def}}{=} Y^{\sharp n} \sqcap^\sharp F^\sharp(Y^{\sharp n}) \tag{4.8}$$

where $X^\sharp$ is the limit found by iteration with widening. We assume that a glb operator $\sqcap^\sharp$ exists, which is the case for all the domains we have seen.

In case the abstract domain does not feature infinite decreasing chains, such as the constant or the sign domains, the sequence (4.8) always terminates in finite time. Otherwise, we can use the *narrowing operator* $\vartriangle$, introduced by Cousot and Cousot [1977]. Its definition is superficially similar to that of the widening (Def. 2.17):

Definition 4.4. *A binary operator $\vartriangle : D^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is a narrowing operator in an abstract domain $\mathcal{D}^\sharp$ if:*

1. $\forall X^\sharp, Y^\sharp \in \mathcal{D}^\sharp$: $X^\sharp \sqcap^\sharp Y^\sharp \sqsubseteq^\sharp X^\sharp \vartriangle Y^\sharp \sqsubseteq^\sharp X^\sharp$;
2. *for any sequence $(Y^{\sharp i})_{i\in\mathbb{N}}$ in $\mathcal{D}^\sharp$, the sequence $(Z^{\sharp i})_{i\in\mathbb{N}}$ computed as $Z^{\sharp 0} = Y^{\sharp 0}$, $Z^{\sharp i+1} = Z^{\sharp i} \vartriangle Y^{\sharp i+1}$ stabilizes in finite time: $\exists k \geq 0$: $Z^{\sharp k+1} = Z^{\sharp k}$.* ■

The first point states that $\vartriangle$ refines its left argument while being a sound approximation. The second point enforces termination.

Using a narrowing, we can now replace (4.8) with:

$$Y^{\sharp 0} \stackrel{\text{def}}{=} X^\sharp \qquad Y^{\sharp n+1} \stackrel{\text{def}}{=} Y^{\sharp n} \vartriangle F^\sharp(Y^{\sharp n}) \tag{4.9}$$

and ensure convergence towards a better approximation of $\operatorname{lfp} F$ than the initial $X^\sharp$ obtained using widening only.

For non-relational domains, $\triangle$ can be constructed, as all our binary operators, pointwise by applying a value narrowing $\triangle_b : \mathcal{B}^\sharp \times \mathcal{B}^\sharp \to \mathcal{B}^\sharp$ independently on each variable. Here is an example narrowing for intervals:

$$[a,b] \triangle_b [c,d] \stackrel{\text{def}}{=} \left[\begin{cases} c & \text{if } a = -\infty \\ a & \text{otherwise} \end{cases}, \begin{cases} d & \text{if } b = +\infty \\ b & \text{otherwise} \end{cases} \right] \tag{4.10}$$

This gives naturally an over-approximation of $\sqcap_b^\sharp$ as we sometimes choose to refine the iterate, i.e., the interval $[a,b]$, using the result of a new loop body analysis, i.e., the interval $[c,d]$, and sometimes refrain from refining it. More precisely, we refine a bound in $[a,b]$ only when the bound is infinite. As each infinite bound can be only refined once, this necessarily terminates in at most $2|\mathbb{V}|$ steps.

Equation 4.9 can be directly plugged into the abstract denotational semantics of loops in Fig. 3.10. When employing an equational-style analysis, it is sufficient to employ narrowing only at selected program points $\mathcal{W}$ such that every cycle in the control-flow graph passes through a narrowing point. Equation 3.4 becomes, after replacing widening $\triangledown$ with narrowing $\triangle$:

$$\forall j \in \mathcal{L}\colon \mathcal{X}_j^{\sharp k+1} \stackrel{\text{def}}{=} \begin{cases} I^\sharp & \text{if } j = 1 \\ \mathcal{X}_k^{\sharp k} \triangle \left(\cup^\sharp_{(i,F,j) \in cfg[p]} F^\sharp(\mathcal{X}_i^{\sharp k}) \right) & \text{if } j \in \mathcal{W} \\ \cup^\sharp_{(i,F,j) \in cfg[p]} F^\sharp(\mathcal{X}_i^{\sharp k}) & \text{otherwise} \end{cases} \tag{4.11}$$

Note that we can use, as narrowing points, the exact same points we used for widenings, i.e., the loop heads, reasoning that these are the program points where there was the most loss of precision, and thus, the most in need of refinement.

Example 4.5 (Decreasing iterations)**.** *Consider again the loop "$X \leftarrow 0$;* **while** $X < 40$ **do** $X \leftarrow X + 1$ **done***" analyzed in the interval domain. We start from the result $\mathcal{X}_\ell^{\sharp 8}$ of the analysis after widening at point 3, in Fig. 4.7. We then apply iterations with narrowing (4.11) at point 3, and we get the sequence described in Fig. 4.11. More precisely:*

- *at 3, we get $\mathcal{Y}_3^{\sharp 1} \stackrel{\text{def}}{=} \mathcal{X}_3^{\sharp 8} \triangle_b (\mathcal{X}_2^{\sharp 8} \sqcup_b^\sharp \mathcal{X}_5^{\sharp 8}) = [0,+\infty] \triangle_b ([0,0] \sqcup_b^\sharp [1,40]) = [0,+\infty] \triangle_b [0,40] = [0,40]$, retrieving a precise loop invariant;*

- *this propagates, after the loop exit test $X \geq 40$, to $[40,40]$ at program point 6, hence, we have proved that the program terminates with $X = 40$;*

- *further iterations give the same result, so that the iteration is finished.*

Thanks to narrowing, we have inferred the best invariants at all program points. Note again that the total number of iterations with widening and narrow is much smaller that the number of concrete iterations of the loop, and does not depend on the value of the constant 40. ♦

ℓ	$\mathcal{Y}_\ell^{\sharp 0} = \mathcal{X}_\ell^{\sharp 8}$	$\mathcal{Y}_\ell^{\sharp 1}$	$\mathcal{Y}_\ell^{\sharp 2}$
1	$\top$	$\top$	$\top$
2	0	0	0
3 $\triangledown$	$[0, +\infty]$	$[0, 40]$	$[0, 40]$
4	$[0, 39]$	$[0, 39]$	$[0, 39]$
5	$[1, 40]$	$[1, 40]$	$[1, 40]$
6	$[40, +\infty]$	$[40, +\infty]$	$[40, 40]$

Figure 4.11: Interval analysis of Fig. 4.7 with decreasing iterations.

It is important to note the conceptual difference between widenings and narrowings. A widening performs an extrapolation step which must necessarily be applied until reaching stabilization as, only at this point can we be sure that we have found a sound fixpoint abstraction; previous iterations may be unsound. A narrowing refines an already correct result. In fact, we can stop iterations (4.9) and (4.11) at any point, and still get a refined but correct result. In particular, when a domain does not feature any narrowing, we can simply apply (4.8), using an intersection $\sqcap^\sharp$, for a finite fixed number of steps, which is sound and, by construction, terminates.

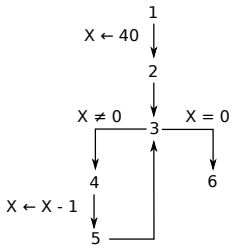
4.7.3 Widening with Thresholds

Another natural solution to improve the precision of loop invariants is to design gentler widenings, that do not immediately abort unstable computations and push bounds to infinity. Consider, for instance, an iteration sequence starting with $[10, 10]$, followed with $[9, 10]$. Then, instead of widening the sequence immediately to $[-\infty, 10]$, we can first widen this sequence to $[0, 10]$ and, only if the lower bound of the next iterate becomes strictly negative, replace it with $-\infty$. The widening will effectively stop at 0 to test its stability on the way to infinity. Formally, we defined $\triangledown_0$ as:

$$[a, b] \triangledown_b^0 [c, d] \stackrel{\text{def}}{=} \left[\begin{cases} a & \text{if } a \leq c \\ 0 & \text{if } 0 \leq c < a \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ 0 & \text{if } 0 \geq d > b \\ +\infty & \text{otherwise} \end{cases} \right] \tag{4.12}$$

Example 4.6 (Widening at zero)**.** *Consider the simple decrementing loop: "$X \leftarrow 40$;* **while** *$X \neq 0$* **do** *$X \leftarrow X - 1$* **done***". We show in, Fig. 4.12, its control-flow graph and the result of an interval analysis with the standard widening $\triangledown_b$ (4.7). At the loop head, the first iteration gives $[40, 40] \triangledown_b [39, 40] = [-\infty, 40]$, which is the loop invariant the analysis outputs.*

For comparison, Fig. 4.12.(b) presents the result of an analysis of the same program with the extended signs of Fig. 4.2.(b). Interestingly, this domain is able to find that $X \geq 0$. Although the interval domain is far more expressive than the sign domain, and the interval domain can exactly represent $X \geq 0$, it fails to infer a simple invariant found by the sign domain. This is naturally due to a too coarse widening. Using our refined



(a)

ℓ	intervals ∇	signs	intervals ∇^0
1	$\top$	$\top$	$\top$
2	40	> 0	40
3 ∇	$[-\infty, 40]$	≥ 0	$[0, 40]$
4	$[-\infty, 40]$	> 0	$[1, 40]$
5	$[-\infty, 39]$	≥ 0	$[0, 39]$
6	0	≥ 0	0

(b)

Figure 4.12: Analysis of a decrementing loop using signs and intervals with various widenings.

widening ∇_b^0 *(4.12) allows the interval domain to find this invariant, though, as we get* $[40, 40] \nabla_b^0 [39, 40] = [0, 40]$*, which is stable.*

Finally, it is interesting to note that the coarse loop invariant $[-\infty, 40]$ *found with* ∇ *cannot be refined using decreasing iterations. Indeed,* $[-\infty, 40]$ *at point 3, is propagated into* $[-\infty, 40]$ *after the test* $X \neq 0$ *and then* $[-\infty, 39]$ *after the assignment* $X \leftarrow X - 1$*, so that the iteration gives* $[-\infty, 40] \triangle_b ([40, 40] \sqcup^\sharp [-\infty, 39]) = [-\infty, 40] \triangle_b [-\infty, 40]$*, i.e., no refinement.* ♦

The idea of testing the stability of some bounds can be generalized to a *widening with thresholds* ∇^T, parameterized with a user-specified finite set T of bounds that includes $\pm\infty$:

$$[a, b] \nabla_b^T [c, d] \stackrel{\text{def}}{=} \left[\begin{cases} a & \text{if } a \leq c \\ \max\{\, x \in T \mid x \leq c \,\} & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ \min\{\, x \in T \mid x \geq d \,\} & \text{otherwise} \end{cases} \right] \tag{4.13}$$

In the worst case, each value of T is tested for each variable bound, hence, we get $2|\mathbb{V}| \times |T|$ iterations, but the analysis nevertheless terminates.

Example 4.7 (Widening with thresholds)**.** *Consider the following loop:*

```
X ← 0;
while [0,1] = 0 do
    if [0,1] = 0 then
        X ← X + 1;
        if X > 40 then X ← 0 endif
    endif
done
```

An interval analysis with standard widening (4.7) will find $X \in [0, +\infty]$ *as loop invariant. Moreover, as in Ex. 4.6, narrowing is ineffective to retrieve any precision because the effect of the loop body on an invariant* $[0, +\infty]$ *is* $[0, +\infty]$*. Indeed, there is a control path in the body where the value of* X *is unchanged. A widening with thresholds* ∇^T *(4.13) will,*

however, find as loop invariant $X \in [0, c]$, where c is the smallest value in T that is greater than 40. If $40 \in T$, we will find the most precise loop invariant $X \in [0, 40]$ but, if $40 \notin T$, we will find a coarser invariant. ♦

Analyzers rely on external heuristics to guess a good threshold set T, on a per variable and per loop basis. One strategy is to use the lexical constants used as array bounds in array declarations (possibly plus or minus 1), when interested in proving the absence of out-of-bound array accesses. When checking for the absence of arithmetic overflows, a good strategy is to consider as T an exponential ramp, such as $\{ \pm 2^i \mid i \in [0, 64] \}$. That way, we can hope to find an approximate bound up to a factor of 2, which is often sufficient as there is generally a significant buffer between the actual maximal value and the overflow threshold.

The ineffectiveness of narrowing in these examples warrants some explanation. The general idea of a decreasing iteration is to start with an abstraction $X^\sharp$ of a postfixpoint $\gamma(X^\sharp)$, and refine it downwards towards the least fixpoint. To ensure soundness, the iteration stays above this least fixpoint. However, the iteration also stays above any other fixpoint that is smaller than $\gamma(X^\sharp)$. So, in case the iteration with widening skipped over several fixpoints above the least one, the decreasing sequence will get stuck at these fixpoints and will not be able to "jump" below them towards the least fixpoint. For instance, in Ex. 4.7, any interval of the form $[0, c]$ with $c \geq 40$ is a concrete fixpoint, hence, the narrowing cannot improve on such an interval. Recently Halbwachs and Henry [2012] proposed more aggressive decreasing iteration techniques, which jump below fixpoints, but then requires a new round of increasing iterations to make sure that the result is above the concrete least fixpoint.

4.7.4 Delayed Widening

Another, complementary way to make the widening gentler is to delay its application. Hence, (2.6) is replaced with:

$$\begin{array}{lcll} X^{\sharp 0} & \stackrel{\text{def}}{=} & \bot^\sharp & \\ X^{\sharp i+1} & \stackrel{\text{def}}{=} & F^\sharp(X^{\sharp i}) & \text{if } i < N \\ X^{\sharp i+1} & \stackrel{\text{def}}{=} & X^{\sharp i} \triangledown F^\sharp(X^{\sharp i}) & \text{if } i \geq N \end{array}$$

for some fixed N. This is particularly useful when the first few iterates of the loop differ from the following ones, and it is always a good idea to start extrapolating only after having accumulated a few iterations. This way, the widening can make a more educated guess about the loop behavior. After a finite, fixed number N of iterations, we revert to widening, so that termination is preserved.

Example 4.8 (Delayed widening)**.** *Consider the following loop:*

```
V ← 0;
while [0,1] = 0 do
    if V = 0 then V ← 1 endif; ...
done
```

The first iteration of the loop sets V from 0 to 1, but then, V remains at 1 for the next iterations — we can think of V as a flag indicating whether we are in the first loop iteration, which is a common pattern in reactive programs consisting in a large loop including an initialization phase. The standard widening $\triangledown$ will see the sequence $[0,0]$, $[0,1]$, then conclude that the upper bound is increasing and replace it with $[0,+\infty]$. When delaying the widening by one step, it is applied to $[0,1]$ and $[0,1]$, which gives back $[0,1]$. Thus, by giving some time to X to stabilize by itself, we avoided a gross over-approximation. ♦

4.7.5 Loop Unrolling

When the first few iterations behave extremely differently from the following ones, it can be useful to analyze them separately. Indeed, when performing iterations, even with delayed widening (Sect. 4.7.4), we ultimately expect to compute a single loop invariant that summarizes the behaviors of all iterates. By keeping a separate abstract values for each iterate for the first few iterates, and an abstract value to summarize the loop behaviors after these iterates, we avoid some loss of precision due to the join.

This technique is easily implemented in the denotational-style abstract analysis. The semantics of loop $\mathbb{S}^\sharp[\![\,\textbf{while}\ c\ \textbf{do}\ s\ \textbf{done}\,]\!]R^\sharp$ from Fig. 3.10, which we recall as:

$$\begin{array}{l} X^{\sharp 0} \stackrel{\text{def}}{=} R^\sharp \\ X^{\sharp n+1} \stackrel{\text{def}}{=} X^{\sharp n} \triangledown F^\sharp(X^{\sharp n}) \\ \text{where } F^\sharp(X^\sharp) \stackrel{\text{def}}{=} R^\sharp \cup \mathbb{S}^\sharp[\![\,s\,]\!](\mathbb{C}^\sharp[\![\,c\,]\!]X^\sharp) \end{array}$$

is changed into:

$$\begin{array}{l} X^{\sharp 0} \stackrel{\text{def}}{=} R^\sharp \\ X^{\sharp n+1} \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![\,s\,]\!]\,(\mathbb{C}^\sharp[\![\,c\,]\!]X^{\sharp n}) \text{ if } n < N \\ X^{\sharp n+1} \stackrel{\text{def}}{=} X^{\sharp n} \triangledown F^\sharp(X^{\sharp n}) \text{ if } n \geq N \\ \text{where } F^\sharp(X^\sharp) \stackrel{\text{def}}{=} R^\sharp \cup \mathbb{S}^\sharp[\![\,s\,]\!](\mathbb{C}^\sharp[\![\,c\,]\!]X^\sharp) \end{array} \tag{4.14}$$

and $X^{\sharp i}$ for $0 \leq i < N$ correspond to the loop executed exactly i times, while the limit $X^\sharp$ of $X^{\sharp n}$ with widening for $n \geq N$ corresponds to the loop executed N times or more.

Although they are computed separately, these invariants must be joined in the end in order to compute the output of the loop:

$$(\cup^\sharp \{\, \mathbb{C}^\sharp[\![\,\neg c\,]\!](X^{\sharp i}) \mid 0 \leq i < N \,\}) \cup^\sharp \mathbb{C}^\sharp[\![\,\neg c\,]\!](X^\sharp)$$

Note that we apply the exit loop condition separately to each invariant, and then join them. Joining as late as possible prevents the loss of precision in non-distributive lattices such as intervals (Sect. 2.1.6).

Example 4.9 (Loop unrolling)**.** *Consider a more refined version of Ex. 4.8, where the*

initialization phase (when $V = 0$) initializes the variable W from unknown to 0:

$$\begin{array}{l} V \leftarrow 0;\, W \leftarrow [-\infty, +\infty]; \\ \textbf{while } [0,1] = 0 \textbf{ do} \\ \quad \textbf{if } V = 0 \textbf{ then } W \leftarrow 0;\, V \leftarrow 1 \textbf{ endif}; \\ \quad W \leftarrow W + 1 \\ \textbf{done} \end{array}$$

The first abstract loop iteration in the interval domain computes $[V \mapsto [1,1], W \mapsto [1,1]]$. However, without loop unrolling, this information is merged with the invariant before any loop iteration takes place, $[V \mapsto [0,0], W \mapsto [-\infty, +\infty]]$, so that we continue iterating with the join $[V \mapsto [0,1], W \mapsto [-\infty, +\infty]]$. Hence, the loop invariant states that $W \in [-\infty, +\infty]$ and, as a consequence, there is no information on W at the position of the incrementation $W \leftarrow W + 1$. In the concrete, however, W is always positive at this point. Applying (4.14) with $N = 1$, the loop invariant we compute starts in the state $[V \mapsto [1,1], W \mapsto [1,1]]$ and goes on iterating with $[V \mapsto [1,1], W \mapsto [1,+\infty]]$. This corresponds only to the loop invariant for the executions reaching the loop header after at least one loop iteration. ♦

4.7.6 Non-Monotonicity of the Widening

Remark that the standard interval widening $\triangledown_b$ (2.6) is monotonic in its second argument, but not in his first argument. As a counter-example for the non-monotonicity, consider that $[1,1] \sqsubseteq_b^\sharp [1,52]$, but $[1,1] \triangledown_b [1,52] = [1,+\infty]$ while $[1,52] \triangledown_b [1,52] = [1,52]$. Given a coarser first argument, we get a more precise result, because the first argument happens now to be stable.

A consequence is that the semantics of a loop in the denotational-style semantics is not monotonic.

Example 4.10 (Non-monotonicity of the widening)**.** *Consider the loop:*

$$\textbf{while } V \leq 50 \textbf{ do } V \leftarrow V + 2 \textbf{ done}$$

which increments V up to 52. The loop semantics is:

$$\mathbb{C}^\sharp[\![\, V > 50 \,]\!](\lim \lambda X^\sharp.\, X^\sharp \triangledown_b (R^\sharp \cup^\sharp \mathbb{S}^\sharp[\![\, V \leftarrow V + 2 \,]\!](\mathbb{C}^\sharp[\![\, V \leq 50 \,]\!](X^\sharp))))$$

where $R^\sharp$ is the abstract state at the loop entry. We can check that:

- *when starting with $R^\sharp = [V \mapsto [1,1]]$, the loop invariant with widening is $[1,+\infty]$, so that the semantics of the loop returns $[51,+\infty]$;*
- *when starting with $R^\sharp = [V \mapsto [1,52]]$, then $X^\sharp$ is stable at the first iteration, so that the loop invariant is $[1,52]$, and we return $[51,52]$, which is much more precise.* ♦

Recall that, although in the concrete we asked for the semantic operators to be monotonic for fixpoints to exist, we took extra care to avoid making any such hypothesis in

the abstract. Indeed, this turns out not to be the case in practice, and the additional complexity of designing loop invariant inference using non-monotonic abstract functions now pays of.

One common case is that of nested loops in the denotational-style semantics. The outer loop must iterate some function $F^\sharp$, which contains the possibly non-monotonic semantics of the inner loop, due to the non-monotonicity of the widening. The case of interval widening is not isolated: Cousot [2015] proved that a widening cannot be monotonic in its first argument, unless the domain has only finite increasing chains — in which case the widening can be replaced with a join anyway.

4.7.7 Widening, Induction, and Invariants

To finish our general discussion on advanced iterations, we review the notions of invariants and inductive reasoning, pervasive in formal program verification, and tie them to the notion of widening, specific to Abstract Interpretation. As we stated in Sect. 3.3, the least fixpoint $\mathrm{lfp}\, F$ we seek is a constructive expression of the tightest loop invariant, which is also the exact reachable set of program states. Due to undecidability, we settle for an invariant, I, that is, any overapproximation $I \supseteq \mathrm{lfp}\, F$.

Hoare-Floyd logic [Hoare, 1969, Floyd, 1967] requires an invariant such that $F(I) \subseteq I$. This is an *inductive invariant*. Note that $F(I) \subseteq I \implies \mathrm{lfp}\, F \subseteq I$ by Tarski's fixpoint theorem (Thm. 2.1), so that any inductive invariant is indeed an invariant. However, the converse is not true: the notion of inductive invariant is stronger than that of plain invariant. It adds the notion of checkability. Proving that I is an inductive invariant is much simpler than proving that $\mathrm{lfp}\, F \subseteq I$: it requires only giving a proof that $F(I) \subseteq I$, i.e., computing F, while checking $\mathrm{lfp}\, F \subseteq I$ would require actually computing $\mathrm{lfp}\, F$, which we wished to avoid in the first place. This explains the focus of deductive methods on inductive invariants.

Abstract Interpretation, however, does not rely on user-given proofs, but on computation in the abstract. Analyzing a loop consists in finding $X^\sharp$ such that $F^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$, i.e., we are looking for an invariant that can be proved to be inductive in the abstract, by actually computing $F^\sharp(X^\sharp)$ and $\sqsubseteq^\sharp$ in the abstract domain which, unlike computing F and $\subseteq$, is feasible automatically and efficiently. We can thus view iteration with widening as a search process that tries to guess some $X^\sharp$ such that $F^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$. The specificity of this method is that it starts evaluating the loop, accumulates reachability information, and tries to extrapolate from it. It is a form of *inductive reasoning* in the classic sense of philosophical logic: a generalization from a finite set of observations. It should not be confused with mathematical induction, which actually consists in applying a well-defined rule or axiom scheme, and is thus deductive in nature.

Usually, inductive reasoning is known to give incorrect results (e.g., deducing that the sun will raise every day, for all eternity, from the fact that we have observed it do so, so far). However, Abstract Interpretation turns this very human and error-prone reasoning method into a formally valid method because iterations always reach an abstract postfixpoint in finite time, thanks to widening. This is possible because our abstract domains have a $\top^\sharp$

element, meaning "no information" and a widening can default to $\top^\sharp$ if no meaningful inductive invariant could be inferred.

4.8 The Congruence Domain

The last non-relational abstract domain we present is a congruence domain, specific to integers, introduced by Granger [1989]. We thus assume $\mathbb{I} = \mathbb{Z}$, and infer value abstractions of the form $X \in a\mathbb{Z} + b$, meaning that X equals some multiple of a plus b. This domain is particularly useful to track pointer offsets in arrays and data-structures, prove alignment properties, or infer array access patterns. We set as abstract values:

$$\mathcal{B}^\sharp \stackrel{\text{def}}{=} \{\, (a\mathbb{Z} + b) \mid a \in \mathbb{N},\ b \in \mathbb{Z} \,\} \cup \{\, \bot_b^\sharp \,\} \tag{4.15}$$

The greatest element can be represented as $1\mathbb{Z} + 0$, which equals $\mathbb{Z}$, while a singleton $\{c\}$ can be represented as $0\mathbb{Z} + c$. However, an element $\bot_b^\sharp$ representing $\emptyset$ is explicitly added.

Order structure. The lattice structure is depicted in Fig. 4.13. This lattice is based on some basic arithmetic results concerning divisors and so-called *cosets*. As we mentioned in Ex. 2.6, non-zero integers form a (non complete) lattice for divisibility: $(\mathbb{N}^*, |, \mathrm{lcm}, \mathrm{gcd})$, where $|$ is the "divides" partial order relation, lcm and gcd are the least common multiple and the greatest common divisor. We first have to extend this lattice to the case 0, which is useful in the domain to represent singletons:

- $y|y' \stackrel{\text{def}}{\iff} y$ divides y', i.e., $\exists k \in \mathbb{N}\colon y' = ky$, including $\forall y\colon y|0$;
- $x \equiv x'\ [y] \stackrel{\text{def}}{\iff} y|(|x - x'|)$, in particular, $x \equiv x'\ [0] \iff x = x'$;
- $\vee$ is the lcm is extended with $y \vee 0 \stackrel{\text{def}}{=} 0 \vee y \stackrel{\text{def}}{=} 0$;
- $\wedge$ is the gcd extended with $y \wedge 0 \stackrel{\text{def}}{=} 0 \wedge y \stackrel{\text{def}}{=} y$.

Then, $(\mathbb{N}, |, \vee, \wedge, 1, 0)$ is a complete distributive lattice. We can define on top of this arithmetic lattice a complete lattice structure over $(\mathcal{B}^\sharp, \sqsubseteq_b^\sharp, \sqcup_b^\sharp, \sqcap_b^\sharp, \bot_b^\sharp, (1\mathbb{Z} + 0))$, where:

- $(a\mathbb{Z} + b) \sqsubseteq_b^\sharp (a'\mathbb{Z} + b') \stackrel{\text{def}}{\iff} a'|a$ and $b \equiv b'\ [a']$
- $(a\mathbb{Z} + b) \sqcup_b^\sharp (a'\mathbb{Z} + b') \stackrel{\text{def}}{=} (a \wedge a' \wedge |b - b'|)\mathbb{Z} + b$
- $(a\mathbb{Z} + b) \sqcap_b^\sharp (a'\mathbb{Z} + b') \stackrel{\text{def}}{=} \begin{cases} (a \vee a')\mathbb{Z} + b'' & \text{if } b \equiv b'\ [a \wedge a'] \\ \bot_b^\sharp & \text{otherwise} \end{cases}$
  where b'' is such that $b'' \equiv b\ [a \vee a'] \equiv b'\ [a \vee a']$, and is given by Bézout's identity and can be computed using the extended Euclidean algorithm.

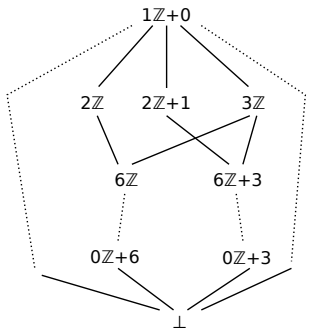


Figure 4.13: Hasse diagram for the congruence domain.

Galois connection. We can construct a Galois connection as follows:

$$\gamma_b(X_b^\sharp) \stackrel{\text{def}}{=} \begin{cases} \{\, ak + b \mid k \in \mathbb{Z} \,\} & \text{if } X_b^\sharp = (a\mathbb{Z} + b) \\ \emptyset & \text{if } X_b^\sharp = \bot_b^\sharp \end{cases}$$

$$\alpha_b(C) \stackrel{\text{def}}{=} \bigsqcup_{c \in C}^\sharp (0\mathbb{Z} + c)$$

Note that γ_b is not injective as several different abstract elements can represent the same concrete set, e.g., $2\mathbb{Z} + 0$ and $2\mathbb{Z} + 2$. However, $\sqsubseteq_b^\sharp$ corresponds exactly to the inclusion: $\gamma_b(X^\sharp) \subseteq \gamma_b(Y^\sharp) \iff X^\sharp \sqsubseteq_b^\sharp Y^\sharp$. We can test for $\gamma_b(X^\sharp) = \gamma_b(Y^\sharp)$ by checking both $X^\sharp \sqsubseteq_b^\sharp Y^\sharp$ and $Y^\sharp \sqsubseteq_b^\sharp X^\sharp$ — technically, $\sqsubseteq_b^\sharp$ is not a partial order but a pre-order, by lack of anti-symmetry, but this has no consequence in practice in the design of our abstractions. Alternatively, we can ensure a unique representation by requiring that, in $a\mathbb{Z} + b$, either $a = 0$ (singleton), or $0 \leq b < a$ (infinite integer set).

Abstract operators. As in the previous domains, we set $\cup_b^\sharp \stackrel{\text{def}}{=} \sqcup_b^\sharp$ and $\cap_b^\sharp \stackrel{\text{def}}{=} \sqcap_b^\sharp$, and the intersection $\cap_b^\sharp$ is exact while the join $\cup_b^\sharp$ is optimal, but not exact — for instance, $\gamma_b(3\mathbb{Z}) \cup \gamma_b(3\mathbb{Z} + 1)$ cannot be exactly represented and is abstracted as $\mathbb{Z}$.

Figure 4.14 presents the abstract arithmetic operators needed to evaluate expressions. All operators except the division are optimal, but generally not exact. For the division, we handle simple cases, such as divisions by 0, or by a singleton value that divides exactly the first argument, and we revert to no information $(1\mathbb{Z} + 0)$ in other cases.

For tests, we can use the method described in Sect. 4.6 using the backward operators synthesized from the forward ones in Fig. 4.9. Note, however, that the formula $\overleftarrow{\leq 0}_b^\sharp(X^\sharp) \stackrel{\text{def}}{=} X^\sharp \cap_b^\sharp [-\infty, 0]_b^\sharp$ gives the identity as $[-\infty, 0]_b^\sharp = 1\mathbb{Z} + 0$; hence, it is worth using a domain-specific version of $\overleftarrow{\leq 0}_b^\sharp(X^\sharp)$ that returns $\bot_b^\sharp$ when $X^\sharp = 0\mathbb{Z} + c$ with $c > 0$.

Although the domain has an infinite height, there is no infinite strictly increasing chain. Indeed, the a component in $a\mathbb{Z} + b$ in any strictly increasing chain must strictly decrease, and it is bounded by 1. We can thus use, as widening: $\nabla_b \stackrel{\text{def}}{=} \sqcup_b^\sharp$. The domain however has infinite strictly decreasing chains, such as $2\mathbb{Z}$, $4\mathbb{Z}$, $\ldots$, $2^n\mathbb{Z}$, $\ldots$. Hence, it is useful to define a narrowing operator $\triangle_b$ to stabilize decreasing sequences in finite time. We present

$$
\begin{array}{rcl}
c_b^\sharp & \stackrel{\text{def}}{=} & 0\mathbb{Z} + c \\
[c, c']_b^\sharp & \stackrel{\text{def}}{=} & \begin{cases} 0\mathbb{Z} + c & \text{if } c = c' \\ 1\mathbb{Z} + 0 & \text{otherwise} \end{cases} \\
-_b^\sharp(a\mathbb{Z} + b) & \stackrel{\text{def}}{=} & a\mathbb{Z} + (-b) \\
(a\mathbb{Z} + b) +_b^\sharp (a'\mathbb{Z} + b') & \stackrel{\text{def}}{=} & (a \wedge a')\mathbb{Z} + (b + b') \\
(a\mathbb{Z} + b) -_b^\sharp (a'\mathbb{Z} + b') & \stackrel{\text{def}}{=} & (a \wedge a')\mathbb{Z} + (b - b') \\
(a\mathbb{Z} + b) \times_b^\sharp (a'\mathbb{Z} + b') & \stackrel{\text{def}}{=} & (aa' \wedge ab' \wedge a'b)\mathbb{Z} + bb' \\
(a\mathbb{Z} + b) /_b^\sharp (a'\mathbb{Z} + b') & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{c}{\begin{cases} \bot_b^\sharp & \text{if } a' = 0 \text{ and } b' = 0 \\ (a/|b'|)\mathbb{Z} + (b/b') & \text{if } a' = 0,\ b' \neq 0,\ b'|a, \text{ and } b'|b \\ 1\mathbb{Z} + 0 & \text{otherwise} \end{cases}}
\end{array}
$$

Figure 4.14: Abstract arithmetic operators in the congruence domain.

here a simple choice:

$$
(a\mathbb{Z} + b) \triangle_b (a'\mathbb{Z} + b') \stackrel{\text{def}}{=} \begin{cases} a'\mathbb{Z} + b' & \text{if } a = 1 \\ a\mathbb{Z} + b & \text{otherwise} \end{cases}
$$

i.e., we only refine the left argument when it is the greatest value, $1\mathbb{Z} + 0$.

Example 4.11 (Congruence analysis)**.** *Consider the program:*

```
X ← 0; Y ← 2;
while X < 40 do
    X ← X + 2;
    if X < 5 then Y ← Y + 18 endif;
    if X > 8 then Y ← Y − 30 endif
done
```

Note that, except when an abstract value represents a constant, any test of the form $X < 40$ or $X > 8$ is abstracted as the identity. Our static analysis will infer, as loop invariant, $X \in 2\mathbb{Z} + 0$, i.e., X is even, and $Y \in 6\mathbb{Z} + 2$, where 6 is the gcd *of 18 and 30 as, at any loop iteration, Y may be incremented by 18 or decremented by 30.* ♦

Rational congruences. The congruence domain can be extended to rationals, instead of integers, as proposed by Granger [1997]. For this domain, $\mathbb{I} = \mathbb{Q}$, and we have the following abstract values:

$$
\mathcal{B}^\sharp \stackrel{\text{def}}{=} \{\, (a\mathbb{Z} + b) \mid a \in \mathbb{Q}^+,\ b \in \mathbb{Q} \,\} \cup \{\, \bot_b^\sharp, \top_b^\sharp \,\}
$$

i.e., we represent sets of the form $\{\, ak + b \mid k \in \mathbb{Z} \,\}$ where a and b can now be rationals instead of only integers. Such sets include singletons as $0\mathbb{Z} + b$. We add a representation

$\bot_b^\sharp$ for $\emptyset$ and $\top_b^\sharp$ for $\mathbb{Q}$. Many arithmetic notions, such as the divides relation $|$, gcd, and lcm extend naturally to rationals. For instance, $\gcd(\frac{a}{b}, \frac{c}{d}) \stackrel{\text{def}}{=} \frac{\gcd(ad,bc)}{bd}$. Likewise, abstract arithmetic operators are similar to the integer case. However, now that the domain has both increasing and decreasing infinite chains, we need both a widening and a narrowing.

This domain has less applications than classic, integer congruences, so, we will not detail its operators further and refer instead the reader to Granger [1997].

4.9 The Cartesian Abstraction

To conclude this presentation of classic non-relational domains, and before moving on to relational domains, we return to the generic construction of the state domain from the value domain from Def. 4.2 and justify the term "non relational".

Consider setting $\mathcal{B}^\sharp \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{I})$. We do not obtain an effective static analyzer, but we can study, from a semantic point of view, the loss of precision due to the non-relational aspect before value sets are further abstracted. Using the identity as γ_b and α_b in Def. 4.2, we get the following Galois connection:

$$\gamma(X^\sharp) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } X^\sharp = \bot^\sharp \\ \{\, \rho \in \mathcal{E} \mid \forall V \in \mathbb{V}\colon \rho(V) \in X^\sharp(V) \,\} & \text{otherwise} \end{cases}$$

$$\alpha(R) \stackrel{\text{def}}{=} \begin{cases} \bot^\sharp & \text{if } R = \emptyset \\ \lambda V \in \mathbb{V}.\, \{\, \rho(V) \mid \rho \in R \,\} & \text{otherwise} \end{cases}$$

and:

$$(\gamma \circ \alpha)(R) = \{\, \rho \in \mathcal{D} \mid \forall V \in \mathbb{V}\colon \exists \rho' \in R\colon \rho(V) = \rho' V \,\} \tag{4.16}$$

This function characterizes the loss of precision due to non-relationality, and is called the *Cartesian abstraction.*

Example 4.12 (Cartesian abstraction)**.** *Consider the set of states in* $\mathbb{Z}^2$*:* $R \stackrel{\text{def}}{=} \{(0,0),(0,2),(2,0)\}$*. Then,* $R' = (\gamma \circ \alpha)(R) = \{(0,0),(0,2),(2,0),(2,2)\}$*, i.e., we added a spurious point at* $(2,2)$*, because there exists a state where the first variable is 2 and a state where the second variable is 2, notwithstanding the fact that there is no state in* R *where both variables are* 2 *simultaneously.*

Note that R *can be expressed in logic form as* $X \in \{0,2\} \wedge Y \in \{0,2\} \wedge X + Y \leq 2$*. To express* R' *in logic, it is sufficient to remove any relation between* X *and* Y*, i.e., we discard* $X + Y \leq 2$ *to get:* $X \in \{0,2\} \wedge Y \in \{0,2\}$*. Hence the term* non-relational. ♦

From a purely semantic point of view, disregarding algorithmic and representation issues, we can characterize non-relational domains as the domains where all elements representable in the abstract are left invariant by (4.16).

4.10 Summary

This chapter introduced several numeric abstract domains obeying Def. 3.1, thus completing the design of an effective static analyzer by Abstract Interpretation started in the previous chapter. We restricted ourselves to non-relational domains, which abstract the set of possible values of each variable independently from the other variables.

The following table sums up the different domains we proposed, and their expressiveness:

signs	$0, (\geq 0), (>0), (\leq 0), (<0), (\neq 0)$	Sect. 4.2
constants	$c \in \mathbb{I}$	Sect. 4.3
constant sets	$C \in \mathcal{P}_{finite}(\mathbb{I})$	Sect. 4.4
intervals	$[a, b]$	Sect. 4.5
congruences	$a\mathbb{Z} + b$	Sect. 4.8

The data-structures and algorithms for these domains are largely independent from the choice of domain and can be factored-out. Binary operators — ordering, meet, join, widening — are defined point-wise; assignments are based on abstract expression evaluation by structural induction; conditionals employ constraint-programming methods. We advocated for the use of a functional array data-structure, which allows implementing atomic operators that have a sub-linear cost with respect to the total number of variables. We tried, as much as possible, to provide Galois connections and optimal value operators — based, for instance, on interval arithmetic, or basic number theory for congruences — but, keeping in mind that optimality does not compose, the assignments and conditions and, ultimately, the result of the full analysis is seldom optimal.

The interval domain features infinite increasing and descending chains, which led us to a lengthy discussion about the nature of fixpoints and how to approximate them in finite time. We proposed several acceleration techniques based on widening and narrowing that give good results in some practical cases, being understood that the problem of optimal fixpoint approximation of general semantic functions in infinite-height domains is undecidable anyway.

We can sum up the design of an abstract domain as the choice of a static approximation: the set of properties representable in the abstract; and a choice of a dynamic approximation: a convergence acceleration technique. Even after the former is fixed, there is room for improvement in the later — e.g., because we know that the program invariants at all points are expressible in the domain, but the widening causes too much over-approximation to find the expected inductive loop invariants, so that it must be refined.

4.11 Bibliographic Notes

The sign and constant domains are generally not expressive enough for program verification; they are often presented as first examples of abstractions for pedagogical purpose, as in [Cousot and Cousot, 1977]. Nevertheless, the constant domain also allows recasting, as Abstract Interpretation, constant propagation, an instance of data-flow analysis used

in program optimization and formalized by Kildall [1973] — other, non-numeric dataflow analyses used in program optimization can also be seen as Abstract Interpretation restricted to finite-height lattices.

The interval domain, on the other hand, is one of the most widely used domain in program verification. It is based on interval arithmetic by Moore [1966], and it is introduced as a complete abstract domain with widening early at the beginning of Abstract Interpretation, by Cousot and Cousot [1976]. While we only focused on programs manipulating prefect integers, rationals, and reals, intervals are also useful to analyze more realistic numeric types. In particular, the monotonicity of floating-point rounding functions makes it easy to use intervals to abstract floating-point computations, as recalled for instance by Miné [2004]. The analysis of machine integers is made slightly more difficult by the wraparound of computations in case of overflows; this leads to alternate versions of intervals, such as wrapped intervals by Gange et al. [2015], or modular intervals by Miné [2012]. The congruence domain is introduced by Granger [1989], and later adapted to rational congruences by Granger [1997]. Other, less used non-relational domains, include algebraic powers by Mastroeni [2004]. Non-relational domains are also used in non-numeric settings; for instance, typing can be seen as a form of abstraction, as explained by Cousot [1997], and classic monomorphic typing becomes an instance of non-relational Abstract Interpretation.

The problem of computing fixpoint approximations in infinite-height domains, and in particular the interval domain, has been widely studied. The iteration with widening and narrowing originally introduced in Cousot and Cousot [1976] has been refined, and we discussed several improvements. We refer the reader to the description of the Astrée analyzer by Bertrane et al. [2010] for additional information on how the classic iteration scheme can be made precise and efficient in practical analysis settings. New approaches have also been advocated. Halbwachs and Henry [2012] proposed improvements based on more aggressive decreasing sequences. Policy iteration has been proposed by Costan et al. [2005] as an alternative to iteration with widening; this technique, based on game theory, can compute the exact least fixpoint in the interval domain given some restrictions on the equation system. The connection between widening and narrowing from Abstract Interpretation and interpolation from logic is discussed by Cousot [2015], which leads to novel iteration techniques.

Chapter 5

Relational Abstract Domains

While the previous chapter focused on non-relational numeric domains, we present here relational domains, able to infer relationships between variables. More precisely, we focus on domains inferring *affine relationships*. which are the most widely used relational domains. On the one hand, a large portion of useful program invariants involve affine expressions, and we will see several examples. On the other hand, the algorithms underlying affine domains still have a reasonable complexity, compared to non-affine domains.

After some motivating examples, we present several affine domains that achieve different trade-offs between cost and expressiveness: an affine equalities domain, an affine inequalities domain (also known as polyhedra domain), and restricted forms, sub-polyhedra, which trade precision for efficiency.

5.1 Motivation

For many static analysis applications, inferring a bound information is sufficient (e.g., arithmetic overflow detection, optimization, etc.). Hence, the interval domain, presented in details in the previous chapter, seems *expressive* enough for this task. However, the interval domain is not guaranteed to find the tightest possible bounds, and it seldom does in practice, except for the simplest programs. We show here a few examples where, although our ultimate goal is to infer bounds, the interval domain is not *precise* enough, and we have to resort to more expressive, more expensive abstract domains.

5.1.1 Relational Tests

One cause of precision loss is that the combination of optimal operators is not necessarily optimal (Ex. 2.15). A common case is when we need, locally, a more expressive invariant, as shown in the following example:

Example 5.1 (Relational test)**.** *Consider the following program, which computes in X*

the minimum of X and Y, and then subtracts it from Y:

$$
\begin{array}{l}
X \leftarrow [0,10]; \\
Y \leftarrow [0,10]; \\
\textbf{if } X \geq Y \textbf{ then } X \leftarrow Y \textbf{ endif}; \\
D \leftarrow Y - X; \\
\textbf{assert } D \geq 0
\end{array}
$$

As a consequence, $Y - X$ is always positive, and there is no assertion failure. In the interval domain, however, the test $\mathbb{C}^\sharp[\![X \geq Y]\!]$ will be abstracted as the identity, as both X and Y have the same initial bounds, $[0,10]$. Hence, in $Y - X$, we get that $X \in [0,10]$ and $Y \in [0,10]$, and so, $D = Y - X \in [-10,10]$. The assertion $D \geq 0$ is not proved correct, although it is expressible in the interval domain.

To prove the assertion, it is necessary to deduce that, at the end of the **then** *branch of* **if** $X \geq Y$ **then** $X \leftarrow Y$ **endif** *we have $X = Y$ and, at the end of the implicit* **else** *branch, we have $X < Y$ so that, after joining the two branches, $X \leq Y$ holds. This requires, in particular, a precise handling of $\mathbb{C}^\sharp[\![\neg(X \geq Y)]\!]$ and of $\mathbb{S}^\sharp[\![X \leftarrow Y]\!]$. Then, it is necessary to exploit the invariant $X \leq Y$ in the evaluation of $Y - X$ to deduce that it is positive. The polyhedra domain, that we will present in Sect. 5.3, is able to do this reasoning. In this example, the shape of the test $X \geq Y$ and the assignment $D \leftarrow Y - X$ alone are good indicators that we need a relational domain.* ♦

5.1.2 Relational Loop Invariants

Another, less obvious case where we need, locally, a more expressive domain than required to express the invariant we seek is that of loops, as shown in the following example:

Example 5.2 (Relational loop invariant)**.** *Consider the following loop from $I = 1$ to 1000 that also increments X at each loop iteration:*

$$
\begin{array}{l}
I \leftarrow 1; \\
X \leftarrow 0; \\
\textbf{while } I \leq 1000 \textbf{ do} \\
\quad I \leftarrow I + 1; \\
\quad X \leftarrow X + 1 \\
\textbf{done}; \\
\textbf{assert } X \leq 1000
\end{array}
$$

The interval analysis of the previous chapter, with widening and narrowing (Sect. 4.7.2), is able to prove that $I \in [1,1001]$ is a loop invariant, and that $I = 1001$ after the loop. However, it finds that $X \in [0,+\infty]$ at both program points, which is correct but imprecise. In particular, neither the decreasing sequence with narrowing, nor the other advanced iteration methods we presented in Sect. 4.7, are able to infer an upper bound for X. The problem is that, as each variable is handled independently, the analysis is similar for X to that of the following program slice: "$X \leftarrow 0$; **while** *$[0,1] = 0$* **do** *$X \leftarrow X + 1$* **done***" where, indeed, $X \in [0,+\infty]$. The reason we can find a precise answer for I and not X is that there is a test, $I \leq 1000$, explicitly bounding I, but no such test exists for X. The only*

```
max(X, Y)
if X > Y then Y ← X endif;
if Y < 0 then Y ← 0 endif
```

(a)

```
max(X, Y)
X' ← X;
Y' ← Y;
if X' > Y' then Y' ← X' endif;
if Y' < 0 then Y' ← 0 endif
```

(b)

Figure 5.1: A function computing the maximum of X, Y and 0 into Y: (a) in original form and (b) instrumented for modular analysis.

way for the interval domain to find a precise bound for X would be to actually unroll the loop 1000 times, effectively executing the program in the concrete, which is not an option, especially if 1000 *is replaced with a larger constant.*

Relational domains can help by inferring a relational loop invariant*: $I = X + 1 \wedge I \in [1, 1001]$. This invariant is inductive, hence, it can be found by iteration with widening. As we will see in Sect. 5.3, the polyhedra domain can infer it in very few iterations, without any unrolling. The number of abstract iterations is small, and independent from the number of concrete iterations, 1000. This invariant implies $X \in [0, 1000]$ at the loop head and, after the loop, $X = 1000$.* ♦

5.1.3 Modular Analyses

A last application of relational domains is the modular analysis of programs. In a modular analysis, we would like to analyze separately each program part and combine the analysis results to get the analysis of the full program. In particular, we would analyze a function only once, and deduce a function summary that can be used at each call context, without having to reanalyze fully the function body each time, hence improving the scalability of the analysis.

Example 5.3 (Modular analysis)**.** *Consider the simple function in Fig. 5.1.(a). It takes as arguments X and Y, and stores into Y the maximum of X, Y, 0. A summary for this function is an input-output relation, able to express the changes made by the function in a symbolic way, without information on the input.*

To infer such a summary, we construct an instrumented version of the function, as shown in Fig. 5.1.(b): we add primed versions of each variable, X' and Y', initialized to X and Y, and modify the code to update X' and Y' instead of X and Y. The invariant at the end of the function provides information about both the memory state at the beginning of the function and that at the end of the function. Using the polyhedra domain of Sect. 5.3, we get $X = X' \wedge Y' \geq Y \wedge Y' \geq X \wedge Y' \geq 0$, stating that X is not changed, while Y is changed to become greater than both X, the initial value of Y, and 0.

While a non-relational domain can be used for the analysis, this will not be of great interest as there is no information on X and Y to propagate; we will only be able to deduce that $Y' \geq 0$. On the other hand, a relational domain can infer — without hypotheses on X

and Y — relationships between X, Y and X', Y'. It will thus output a summary reusable in all contexts. ♦

5.2 The Affine Equalities Domain (Karr's Domain)

The first relational domain we propose focuses on inferring affine equalities. This domain was introduced by Karr [1976].

More precisely, the invariants we infer have the form:

$$\bigwedge_{j=1}^{m} \sum_{i=1}^{|\mathbb{V}|} \alpha_{ij} V_i = \beta_j,\ \alpha_{ij}, \beta_j \in \mathbb{I}$$

i.e., a conjunction of affine equalities. The number m of equalities, as well as the exact value of the coefficients α_{ij} and β_j, is inferred by the analysis. Another, more geometric view, is to consider that abstract elements are affine subsets:

$$\mathcal{D}^\sharp \simeq \{\, \text{affine subspaces of } \mathbb{V} \to \mathbb{I} \,\}$$

We use an equivalence $\simeq$ here because the actual machine representation will use matrices, as detailed in the next section.

The domain and its operators are based on basic affine algebra and affine spaces. We assume some familiarity with these theories and only recall the relevant results useful here. Further information can be found in textbooks, such as Lang [1997]. A first important remark is that we use algorithms specific to fields, which do not work on integers. Hence, we assume here that $\mathbb{I} \in \{\mathbb{Q}, \mathbb{R}\}$. Secondly, as mentioned in Chap. 2, we can assimilate memory states in $\mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{I}$ to points or vectors in a vector space. More precisely we define: $\mathbb{P} \stackrel{\text{def}}{=} \mathbb{I}^{|\mathbb{V}|}$.

5.2.1 Abstract Representation

Constraint representation. An affine subspace is represented either by $\bot^\sharp$, to denote the empty set, or by a pair $\langle \mathbf{M}, \vec{C} \rangle$ where:

- $\mathbf{M} \in \mathbb{I}^{m \times n}$ is a $m \times n$ matrix, where $n \stackrel{\text{def}}{=} |\mathbb{V}|$ and $m \leq n$;
- $\vec{C} \in \mathbb{I}^m$ is a column vector with m rows.

In this representation, each row of $\mathbf{M}$ and the corresponding row of $\vec{C}$ represents an affine equality constraint, while each column of $\mathbf{M}$ corresponds to a different variable. Moreover, a set of constraints stands for the conjunction of these constraints. We call $\langle \mathbf{M}, \vec{C} \rangle$ a *constraint representation*. We have the following concretization function:

$$\begin{array}{lcl} \gamma(\bot^\sharp) & \stackrel{\text{def}}{=} & \emptyset \\ \gamma(\langle \mathbf{M}, \vec{C} \rangle) & \stackrel{\text{def}}{=} & \{\, \vec{V} \in \mathbb{P} \mid \mathbf{M} \times \vec{V} = \vec{C} \,\} \end{array} \tag{5.1}$$

There are often several ways to represent the same affine subspace as a matrix-vector pair. Hence, we impose the following additional condition:

Definition 5.1 (Row-echelon form)**.** $\mathbf{M}$ *is in* row-echelon form *when* $\forall i \leq m\colon \exists k_i\colon M_{ik_i} = 1$ *and* $\forall c < k_i\colon M_{ic} = 0 \wedge \forall t \neq i\colon M_{tk_i} = 0$*. Moreover* $\forall i < i'\colon k_i < k_{i'}$*.* ■

In row-echelon form, every row starts with some 0, then features a 1, followed by arbitrary coefficients. The column with this 1 is called the variable in *leading position*. If a variable appears in leading position in some row, by Def. 5.1, it cannot occur in any other row, i.e., the coefficient of the variable is 0 in the other equations. In particular, a variable can be leading only in one row. Note, however, that it is possible for a variable not to be leading in any row, and thus appear in zero, one, or several rows, with coefficients different from 1. The name *echelon* comes from the fact that rows are organized by increasing leading variable column.

Example 5.4 (Row-echelon form)**.** *The following matrix:*

$$\begin{bmatrix} \mathbf{1} & 0 & 0 & 5 & 0 \\ 0 & \mathbf{1} & 0 & 6 & 0 \\ 0 & 0 & \mathbf{1} & 7 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} \end{bmatrix}$$

corresponds to the equation system $V_0 + 5V_3 = c_0 \wedge V_1 + 6V_3 = c_1 \wedge V_2 + 7V_3 = c_2 \wedge V_4 = c_3$*, given* $\mathbb{V} \stackrel{\text{def}}{=} \{V_0, V_1, V_2, V_3, V_4\}$ *and the vector of constant coefficients* $\vec{C} \stackrel{\text{def}}{=} [c_0,\, c_1,\, c_2,\, c_3]$*. While* V_0*,* V_1*, and* V_2 *appear in leading position, this is not the case for* V_3*, which can thus appear in several equations, with coefficients different from* 1*.* ♦

The row-echelon form has many useful properties:

- every non-empty affine subspace can be represented in row-echelon form; the full space $\mathbb{P}$ is represented as the empty matrix (no row implies no constraint); the empty set cannot be represented, hence we add $\bot^\sharp$;
- the representation is unique; hence, row-echelon forms enriched with $\bot^\sharp$ constitute a normal form for affine subspaces;
- a row-echelon form has at most $|\mathbb{V}|$ rows; hence, we can bound the size of our abstract representation by $|\mathbb{V}|^2$.

We thus state:

$$\mathcal{D}^\sharp \stackrel{\text{def}}{=} \{\bot^\sharp\} \cup \{\, \langle \mathbf{M}, \vec{C} \rangle \mid \begin{array}{l} \mathbf{M} \in \mathbb{I}^{m \times n},\, \vec{C} \in \mathbb{I}^m,\, 0 \leq m \leq n = |\mathbb{V}|, \\ \mathbf{M} \text{ is in row-echelon form} \,\} \end{array}$$

Normalization. Sometimes, as a consequence of applying some operator, we get a pair $\langle \mathbf{M}, \vec{C} \rangle$ which is not in row-echelon form. However, it can be easily normalized into row-echelon form using Gaussian elimination. We do not detail here this well-known algorithm. Suffice to say that it combines rows (replacing rows with affine combination of rows, thus preserving the affine subspace represented) to put as many variables as possible in leading position. The complexity of this algorithm is cubic, in $\mathcal{O}(|\mathbb{V}|^3)$. It is the most costly algorithm used in the domain, and the bottleneck for its complexity.

Generator representation. There exists an alternate way to represent affine subspaces, using a point $\vec{O} \in \mathbb{P}$, called the origin, and a set of vectors $\mathbf{G} \stackrel{\text{def}}{=} \{\vec{G}_1, \ldots, \vec{G}_m\}$, called the basis. Then, the pair $[\mathbf{G}, \vec{O}]$, put in brackets to avoid ambiguity with the constraint representation $\langle \mathbf{M}, \vec{C} \rangle$ seen above, and called the *generator representation*, represents the affine subspace obtained by linear combinations of the vectors and the origin:

$$\gamma([\mathbf{G}, \vec{O}]) \stackrel{\text{def}}{=} \{ \vec{O} + \mathbf{G} \times \vec{\lambda} \mid \lambda \in \mathbb{I}^m \} \tag{5.2}$$

Without loss of generality, we can assume the vectors in $\mathbf{G}$ to be linearly independent, which implies that $m \leq |\mathbb{V}|$.

It is possible to switch from one representation to the other, by solving equations. For instance, to convert a generator representation $[\mathbf{G}, \vec{O}]$ to constraints, we can see (5.2) as an equation system: $\exists \vec{\lambda}\colon \vec{V} = \vec{O} + \mathbf{G} \times \vec{\lambda}$ and solve it in terms of $\vec{\lambda}$ using Gaussian elimination, keeping only the equations where no λ appears.

In his original work, Karr [1976] advocated for the use of the constraint representation only, arguing that, in practice, program invariants are likely to require only few constraints but many generators to be represented. Hence, we will not discuss generators further, until Sect. 5.3 at least.

5.2.2 Lattice Structure

Intersection. Affine subspaces form a lattice for inclusion. Affine subspaces are naturally closed by intersection. As the constraint representation is a conjunction of constraints, it is easy to construct an intersection by simply putting all the constraints from both arguments together, and applying a normalization step afterwards (which we leave implicit in our formulas):

$$\langle \mathbf{M^1}, \vec{C^1} \rangle \sqcap^\sharp \langle \mathbf{M^2}, \vec{C^2} \rangle \stackrel{\text{def}}{=} \left\langle \begin{bmatrix} \mathbf{M^1} \\ \mathbf{M^2} \end{bmatrix} \begin{bmatrix} \vec{C^1} \\ \vec{C^2} \end{bmatrix} \right\rangle$$

Partial order. Inclusion can be tested by using the classic set identity $A \subseteq B \iff A \cap B = A$ and recalling that, as we have a normal form, testing whether $\langle \mathbf{M^1}, \vec{C^1} \rangle$ and $\langle \mathbf{M^2}, \vec{C^2} \rangle$ represent the same set amounts to comparing the matrices and the vectors element-wise. Hence:

$$\langle \mathbf{M^1}, \vec{C^1} \rangle \sqsubseteq^\sharp \langle \mathbf{M^2}, \vec{C^2} \rangle \stackrel{\text{def}}{\iff} \langle \mathbf{M^1}, \vec{C^1} \rangle \sqcap^\sharp \langle \mathbf{M^2}, \vec{C^2} \rangle = \langle \mathbf{M^1}, \vec{C^1} \rangle$$

Join. Constructing the join, i.e., the smallest affine space that contains two affine spaces, is a little more involved. Karr [1976] introduces a rather complex but efficient algorithm based on iteratively relaxing both argument matrices, column by column, until they are equal. We propose here a simpler, more algebraic take on union, based on the work of Benoy et al. [2005], which we will mention again when discussing the more complex case of polyhedra (Sect. 5.3) on which this technique was originally developed.

A point $\vec{V} \in \mathbb{P}$ is in the affine join if it is an affine combination of a point $\vec{V^1} \in \gamma(\langle \mathbf{M^1}, \vec{C^1} \rangle)$ and a point $\vec{V^2} \in \gamma(\langle \mathbf{M^2}, \vec{C^2} \rangle)$, i.e., $\exists \lambda^1, \lambda^2 \in \mathbb{R}\colon \vec{V} = \lambda^1 \vec{V^1} + \lambda^2 \vec{V^2}$ and

$\lambda^1 + \lambda^2 = 1$. Hence, we have the following equation system expressing that $\vec{V}$ is in the join:

$$\begin{cases} \mathbf{M^1} \times \vec{V^1} = \vec{C^1} \\ \mathbf{M^2} \times \vec{V^2} = \vec{C^2} \\ \vec{V} = \lambda^1 \vec{V^1} + \lambda^2 \vec{V^2} \\ \lambda^1 + \lambda^2 = 1 \end{cases}$$

which, once λ^1, λ^2, $\vec{V^1}$, and $\vec{V^2}$ are eliminated, gives a constraint system expressing the join. Note, however that this system is not linear, due to the products $\lambda^1\vec{V^1}$ and $\lambda^2\vec{V^2}$, where several variables appear. We can solve this problem with a change of variables. Let us write $\vec{W^1} \stackrel{\text{def}}{=} \lambda^1\vec{V^1}$ and $\vec{W^2} \stackrel{\text{def}}{=} \lambda^2\vec{V^2}$. Then, by multiplying the first equation by λ^1 and the second equation by λ^2, we can get rid of $\vec{V^1}$ and $\vec{V^2}$, and replace them with $\vec{W^1}$ and $\vec{W^2}$, to get:

$$\begin{cases} \mathbf{M^1} \times \vec{W^1} = \lambda^1 \vec{C^1} \\ \mathbf{M^2} \times \vec{W^2} = \lambda^2 \vec{C^2} \\ \vec{V} = \vec{W^1} + \vec{W^2} \\ \lambda^1 + \lambda^2 = 1 \end{cases} \tag{5.3}$$

We can now eliminate $\vec{W^1}$, $\vec{W^2}$, λ^1, and λ^2 from this system using Gaussian elimination, as it is linear in these variables. We then get an equation system in $\vec{V}$ expressing that $\vec{V}$ is in the join of $\langle\, \mathbf{M^1}, \vec{C^1} \,\rangle$ and $\langle\, \mathbf{M^2}, \vec{C^2} \,\rangle$, i.e., we have constructed a constraint representation of the join $\sqcup^\sharp$.

Example 5.5 (Join of affine subspaces)**.** *Consider the simple case of joining two points:* $\vec{P^1} \stackrel{\text{def}}{=} (1, 10)$ *and* $\vec{P^2} \stackrel{\text{def}}{=} (2, 12)$*, represented as* $\left\langle \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 10 \end{bmatrix} \right\rangle$ *and* $\left\langle \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 12 \end{bmatrix} \right\rangle$. *We have to eliminate all* W *and* λ *in the following system, instantiated from (5.3):*

$$\begin{cases} W_1^1 = \lambda^1 \\ W_2^1 = 10\lambda^1 \\ W_1^2 = 2\lambda^2 \\ W_2^2 = 12\lambda^2 \\ V_1 = W_1^1 + W_1^2 \\ V_2 = W_2^1 + W_2^2 \\ \lambda^1 + \lambda^2 = 1 \end{cases}$$

This gives $2V_1 + 8 = V_2$*, which is indeed the smallest affine subspace containing both* $\vec{P^1}$ *and* $\vec{P^2}$*.*

Similar examples occur frequently. For instance, $\vec{P^1}$ *and* $\vec{P^2}$ *may be memory states constructed after 0 and 1 iteration of a loop, and the join is performed at the loop head as part of computing a loop invariant accumulating all iterates. This example illustrates how non-trivial relational invariants (here, an equality) are inferred from non-relational ones (here, two constant points): through the join operation.* ♦

Galois connection. We can use the join $\sqcup^\sharp$ to get the smallest affine subspace $\alpha(S)$ that contains a given set of points S:

$$\alpha(S) \stackrel{\text{def}}{=} \sqcup^\sharp \{ \langle \mathbf{I}, \vec{P} \rangle \mid \vec{P} \in S \}$$

where $\mathbf{I}$ is the identity matrix of size $|\mathbb{V}|$, so that $\langle \mathbf{I}, \vec{P} \rangle$ represents the singleton $\{\vec{P}\}$. Note that $\alpha(S)$ is well-defined, even if S is infinite. Indeed, we observe that $X^\sharp \sqcup^\sharp Y^\sharp$ either equals $X^\sharp$, or generates an affine subspace of strictly greater dimension. As the dimension is bounded by $|\mathbb{V}|$, the process of adding new points always terminates. Hence, we have a Galois connection $(\mathcal{P}(\mathbb{P}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$.

5.2.3 Abstract Operators

As for previous domains, we set $\cup_b^\sharp \stackrel{\text{def}}{=} \sqcup_b^\sharp$, which is optimal, and $\cap_b^\sharp \stackrel{\text{def}}{=} \sqcap_b^\sharp$, which is exact.

Conditions. For abstract tests $\mathbb{C}^\sharp[\![c]\!]$, we handle precisely only the affine case, i.e.: $\sum_j \alpha_j V_j = \beta$, which is easy as such a test can be exactly represented in our domain:

$$\mathbb{C}^\sharp[\![\textstyle\sum_j \alpha_j V_j = \beta]\!] \langle \mathbf{M}, \vec{C} \rangle \stackrel{\text{def}}{=} \left\langle \begin{bmatrix} \mathbf{M} \\ \alpha_1 \cdots \alpha_n \end{bmatrix}, \begin{bmatrix} \vec{C} \\ \beta \end{bmatrix} \right\rangle$$

hence, this operator is exact. In all other cases, we revert to the sound, but coarse, identity: $\mathbb{C}^\sharp[\![c]\!] X^\sharp \stackrel{\text{def}}{=} X^\sharp$.

Assignments. Likewise, we handle exactly only affine assignments. For the other cases, we revert to the non-deterministic assignment $\mathbb{S}^\sharp[\![V_j \leftarrow [-\infty, +\infty]]\!]$, which is always a sound abstraction of $\mathbb{S}^\sharp[\![V_j \leftarrow e]\!]$, for any e.

The non-deterministic assignment $V_j \leftarrow [-\infty, +\infty]$ can be modeled by removing all the constraints involving V_j. If V_j appears in leading position, there is only one constraint featuring V_j, and we remove it to obtain the exact abstraction of the non-deterministic assignment. If V_j appears in non-leading position, it may appear in several constraints. Removing all these constraints would lead to a sound approximation, but a coarse one: we know that when assigning a variable, we gain only one degree of freedom, i.e., the resulting equation system should have only one less equation than the original one. Our solution is to use one equation where V_j appears, and combine it with other equations to remove all other occurrences of V_j; then, this equation is the only remaining one with V_j, and it is removed. Additionally, choosing to use the latest row where V_j appears to eliminate all others and be removed ensures that the system stays in row-echelon form. Geometrically, this operation is a projection — projecting out V_j. In logic, this operation is a quantifier elimination — removing the existential quantifier in $\exists V_j \colon \mathbf{M} \times \vec{V} = \vec{C}$.

Example 5.6 (Non-deterministic assignment)**.** *Consider modeling $V_2 \leftarrow [-\infty, +\infty]$ in the following equation system:*

$$\begin{cases} V_0 \quad + V_2 = 10 \\ \quad V_1 + V_2 = 7 \end{cases}$$

We first subtract the second equation from the first one, and then keep only the first equation. We get: $V_0 - V_1 = 3$. *We have fully eliminated* V_2 *while still keeping one equation.* ♦

Handling affine assignments is better done backwards. The reader familiar with Hoare logic [Hoare, 1969] will recall the rule for assignment: $\{P[V/e]\}V \leftarrow e\{P\}$. It states that, if we have a program invariant P valid after the assignment, then we can infer a program invariant $P[V/e]$ valid before the assignment (it is actually the weakest precondition, as shown by Dijkstra [1975]) by *substitution*. Indeed, if some property is true of V after the assignment, it is true of e before. The constraint representation of our abstract elements is very similar to a logical formula, and so, the same substitution principle applies. However, unlike Hoare and Dijsktra's logic, we operate forward as we must deduce an invariant after the assignment from an invariant before the assignment. To solve this problem, we have to distinguish two cases:

- In the assignment $V_j \leftarrow \sum_i \alpha_i V_i + \beta$, if $\alpha_j \neq 0$, the assignment is said to be *invertible*: we can express the current value of V_j as a function of its value after the assignment: $e \stackrel{\text{def}}{=} (V_j - \sum_{i \neq j} \alpha_i V_i - \beta)/\alpha_j$. The result of the assignment is then obtained by substituting, in the argument, every occurrence of V_j with e. Naturally, the system remains an affine system, and the abstract operator is exact.

- In case $\alpha_j = 0$, inverting the assignment is not possible: the assignment is said to be *non-invertible*. More precisely, a non-invertible assignment is one where it is impossible to express the value of the variable before the assignment as a function of the value of variables after the assignment: the value is irremediably lost. We know, however, that the value of V_j in the output is defined by the affine constraint $c \stackrel{\text{def}}{=} (V_j - \sum_i V_i \alpha_i - \beta = 0)$. Hence, the assignment is handled as: $\mathbb{C}^\sharp[\![c]\!] \circ \mathbb{S}^\sharp[\![V_j \leftarrow [-\infty, +\infty]]\!]$. This is also an exact operator.

Example 5.7 (Affine assignments)**.** *Consider the abstract element represented, in logical rather than matrix form for compactness, as* $X + Z = 1 \wedge Y + 2Z = 0$.

The assignment $Z \leftarrow Z + 1$, *which is invertible, is handled by substituting* Z *with* $Z - 1$, *and we get* $X + Z = 2 \wedge Y + 2Z = 2$.

The assignment $X \leftarrow Y$, *which is not invertible, is handled by forgetting* $X + Z = 1$ *where* X *occurs, and adding the constraint* $X - Y = 0$. *Hence, we get* $X - Y = 0 \wedge Y + 2Z = 0$ *which gives, after putting the result into row-echelon form:* $X + 2Z = 0 \wedge Y + 2Z = 0$. ♦

Widening. We have already mentioned that the affine equalities domain has no infinite chain, as any two distinct elements in the chain must also have distinct dimensions, i.e., distinct numbers of rows in the row-echelon normal form. As the dimension is bounded in $[0, |\mathbb{V}|]$, the chain is finite. Thus, we can use as widening the join, $\triangledown \stackrel{\text{def}}{=} \sqcup^\sharp$, to enforce the monotonicity of abstract iterates without requiring extra steps to enforce convergence. Likewise, to perform decreasing iterations, we can simply use the intersection as narrowing $\triangle \stackrel{\text{def}}{=} \sqcap^\sharp$.

5.2.4 Affine Equalities Analysis Example

Example 5.8 (Affine analysis)**.** *Consider again Ex. 5.2, which motivated the introduction of relational domains:* $^{\ell 1}\, I \leftarrow 1;\ X \leftarrow 0;\ ^{\ell 2}$ **while** $^{\ell 3}\, I \leq 1000$ **do** $^{\ell 4}\, I \leftarrow I + 1;\ X \leftarrow X{+}1\ ^{\ell 5}$ **done**$^{\ell 6}$ *. We analyze this program using the equational-style analysis with widening (implemented as* $\sqcup^\sharp$*) at loop head (program point 3). The most relevant iterates are as follows:*

ℓ	$\mathcal{X}_\ell^{\sharp 0}$	$\mathcal{X}_\ell^{\sharp 4}$	$\mathcal{X}_\ell^{\sharp 5}$	$\mathcal{X}_\ell^{\sharp 8}$
1	$\top$	$\top$	$\top$	$\top$
2	$\bot$	$I = 1,\ X = 0$	$I = 1,\ X = 0$	$I = 1,\ X = 0$
3 $\triangledown$	$\bot$	$I = 1, X = 0$	$I = X + 1$	$I = X + 1$
4	$\bot$	$I = 1,\ X = 0$	$I = 1,\ X = 0$	$I = X + 1$
5	$\bot$	$I = 2,\ X = 1$	$I = 2,\ X = 1$	$I = X + 1$
6	$\bot$	$I = 1,\ X = 0$	$I = 1,\ X = 0$	$I = X + 1$

- *At iteration 4, the effect of the assignments* $I \leftarrow 1$ *and* $X \leftarrow 0$ *have been propagated over the first loop iteration. Note that both* $\mathbb{C}^\sharp[\![\, I \leq 1000 \,]\!]$ *and* $\mathbb{C}^\sharp[\![\, \neg(I \leq 1000) \,]\!]$ *are handled as the identity.*

- *At iteration 5, the join between* $I = 1 \wedge X = 0$ *from* $\mathcal{X}_2^{\sharp 4}$ *and* $I = 2 \wedge X = 1$ *from* $\mathcal{X}_5^{\sharp 4}$ *gives the relation* $I = X + 1$ *for program point 3.*

- *This relation is propagated throughout the following iterations until, at iteration 8, the invariants are stable.*

At the end of the analysis, we find as loop invariant the relation $I = X + 1$*. Note that this is the best loop invariant expressible in the domain, but we miss the bounds on* I *that would require expressing inequalities. At the end of the program, as* $\mathbb{C}^\sharp[\![\, \neg(I \leq 1000) \,]\!]$ *is the identity, we also find* $I = X{+}1$*. The most precise invariant would be* $I = 1001 \wedge X = 1000$*, which is expressible in the affine domain but, due to approximations in the loop invariants, cannot be found by the domain.* ♦

5.2.5 Handling Integers

Up to now, we have assumed $\mathbb{I} \in \{\mathbb{Q}, \mathbb{R}\}$: variables take real or rational values, as do the matrix and vector coefficients in the abstract representation. Assume now that $\mathbb{I} = \mathbb{Z}$, i.e., variables take integer values. We cannot use integers as coefficients in the abstract representation as it would lead to unsound algorithms. For instance, it is not possible to put an integer system into row-echelon form as the integer division truncates its result: $2X + Y = 19$ would lead, by normalization of the leading coefficient, to $X = 9$, which is not equivalent to $2X + Y = 19$.

Integer concretization. To handle integers, our solution is to keep an abstract representation in $\mathbb{Q}$, but state that it does not represent the affine subspace, but rather the integer-valued memory states in this affine subspace. We change γ from (5.1) into $\gamma_\mathbb{Z}$ as

follows:

$$\gamma_{\mathbb{Z}}(X^\sharp) \stackrel{\text{def}}{=} \gamma(X^\sharp) \cap \mathbb{Z}^{|\mathbb{V}|} \tag{5.4}$$

By changing the definition of γ, we also change the notion of soundness, optimality, and exactness of the abstract operators. We can check that the operators we proposed are still sound with respect to $\gamma_{\mathbb{Z}}$. They are not exact nor optimal as often as in the rational case, though, as shown in the following example:

Example 5.9 (Non-exactness of the integer semantics)**.** *Consider the abstract state* $X^\sharp \stackrel{\text{def}}{=} \langle\, [2\ -1],\ [0]\,\rangle$ *representing the equation* $2V_0 - V_1 = 0$*, and the assignment* $V_0 \leftarrow 0$*. Assuming real or rational variable values, in the concrete, we get* $\mathbb{S}[\![\, V_0 \leftarrow 0 \,]\!]\gamma(X^\sharp) = \{0\} \times \mathbb{I}$*:* V_0 *is null and* V_1 *can have any value. In the abstract, we get* $\gamma(\mathbb{S}^\sharp[\![\, V_0 \leftarrow 0 \,]\!]X^\sharp) = \gamma(\langle\, [1\ 0],\ [0]\,\rangle) = \{0\} \times \mathbb{I}$*, which represents exactly the concrete result.*

Assume now an integer semantics. We get in the concrete $\mathbb{S}[\![\, V_0 \leftarrow 0 \,]\!]\gamma_{\mathbb{Z}}(X^\sharp) = \{0\} \times 2\mathbb{Z}$*, i.e.,* V_1 *is necessarily even. In the abstract* $\gamma_{\mathbb{Z}}(\mathbb{S}^\sharp[\![\, V_0 \leftarrow 0 \,]\!]X^\sharp) = \gamma_{\mathbb{Z}}(\langle\, [1\ 0],\ [0]\,\rangle) = \{0\} \times \mathbb{Z}$*, which is strictly larger. This is expected as our abstract domain cannot represent the information that* V_1 *is even.*

We can construct similar examples where the result is optimal in $\mathbb{Q}$ *and* $\mathbb{R}$ *but not optimal in* $\mathbb{Z}$*.* ♦

Hence, we can use the affine domain directly to soundly analyze integer programs, paying only a small price in precision. Intuitively, the analysis is not able to exploit the "integerness" of our numbers. Interestingly, we see here rationals and reals as abstractions of integers — not the converse — as they forget this "integerness" property.

Linear congruence equality domain. Another possible adaptation of affine equalities to the analysis of integer programs is to embrace the additional expressive power of integers: we enrich the domain to represent not only affine equalities but also congruence information. Granger [1991] proposed such an extension: the domain of *linear congruence equalities*, able to represent invariants of the form:

$$\bigwedge_j \sum_{i=1}^{|V|} \alpha_{ij} V_i \equiv \beta_j\ [k_j],\ \alpha_{ij}, \beta_j, k_j \in \mathbb{Z}$$

Such invariants are closed under more operations than the affine equalities domain and, as a consequence, it is possible to define exact abstractions for many more operators, such as non-invertible assignments, which limits the loss of precision. The domain is based on an extended version of Euclid's algorithm, instead of Gaussian elimination, in order to combine rows and eliminate variables. We will not detail here the, more complex, algorithms, and refer instead the read to [Granger, 1991]. Suffice to say that the domain is more expensive due to the more involved algorithms.

In practice, when it is necessary to analyze integer programs, it is far more convenient to opt for the first, ad-hoc and approximate solution: use rational-valued affine equalities to approximate sets of integer points.

5.3 The Affine Inequalities Domain (Polyhedra Domain)

The *polyhedra domain* has been introduced by Cousot and Halbwachs [1978] in order to infer affine inequalities among variables. It combines the ability to represent bounds, which was already the case for the interval domain we presented in Sect. 4.5, with the ability to infer relations. More precisely, the polyhedra domain infers invariants of the form:

$$\bigwedge_j \sum_{i=1}^{|\mathbb{V}|} \alpha_{ij} V_i \geq \beta_j,\ \alpha_{ij}, \beta_j \in \mathbb{I} \tag{5.5}$$

Note that this domain subsumes both intervals and affine equalities, being strictly more expressive than both. While the interval domain is one of the most used non-relational abstract domain, the polyhedra domain is one of the most widespread relational domain.

All the properties and algorithms of this domain are based on the theory of convex polyhedra and linear programming. We will use advanced results without proof, and refer the reader to classic textbooks on the subject, such as [Schrijver, 1986], for more information on the underlying theory.

Similarly to linear algebra, we must assume we work in a field, i.e., we assume $\mathbb{I} \in \{\mathbb{Q}, \mathbb{R}\}$ — although, as we will see, integer programs can be analyzed with rational polyhedra, albeit losing precision guarantees.

Equation 5.5 represents, in general, a convex, topologically closed polyhedron. Note that the polyhedron can be bounded (i.e., a polytope), but also unbounded (such as $X \geq 0$):

$$\mathcal{D}^\sharp \simeq \{\, \text{closed convex polyhedra of } \mathbb{P} \,\}$$

where we recall that $\mathbb{P} \stackrel{\text{def}}{=} \mathbb{I}^{|\mathbb{V}|}$.

5.3.1 Dual Representations

A key result of polyhedra theory, the Weyl-Minkowski Theorem, states that polyhedra have dual representations: one using constraints, and one using generators. The classic presentation and implementation of the polyhedra domain uses both representations, as most operators have one preferred representation on which they are much easier to compute than on the other. Which representation is easier, however, varies from one operator to the other. Maintaining and making use of both representations simultaneously results in the *double description* method for polyhedra.

Constraint representation. A first, straightforward representation is in term of affine inequality constraints, which can be written in matrix form: a pair $\langle \mathbf{M}, \vec{C} \rangle$ of a matrix $\mathbf{M} \in \mathbb{I}^{m \times n}$ and a vector $\vec{C} \in \mathbb{I}^m$, where $n \stackrel{\text{def}}{=} |\mathbb{V}|$ is the number of variables and m is the number of constraints. Such a pair denotes the following set:

$$\gamma(\langle \mathbf{M}, \vec{C} \rangle) \stackrel{\text{def}}{=} \{\, \vec{V} \in \mathbb{P} \mid \mathbf{M} \times \vec{V} \geq \vec{C} \,\} \tag{5.6}$$

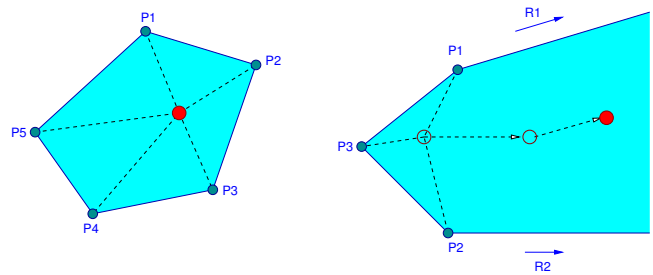


Figure 5.2: Generator representation for bounded polyhedra (left) and unbounded polyhedra (right).

where, similarly to affine equalities, each row of $\mathbf{M}$ and the associated element of $\vec{C}$ correspond to a constraint, and each column of $\mathbf{M}$ represents a variable.

When more convenient, though, we may use a constraint set notation, $\{\sum_i \alpha_{ij} V_i \geq \beta_j \mid j \in [1, m]\}$, or a logical notation, $\bigwedge_{j=1}^{m} \sum_i \alpha_{ij} V_i \geq \beta_j$, which are equivalent to the matrix notation. Using the dot product notation, we can alternative write these constraints as $\{\vec{\alpha}_j \cdot \vec{V} \geq \beta_j \mid j \in [1, m]\}$ and $\bigwedge_{j=1}^{m} \vec{\alpha}_j \cdot \vec{V} \geq \beta_j$.

Generator representation. The second representation is based on so-called *generators*, that is, vectors representing either *vertices* or *rays*. A bounded polyhedron can be seen as the convex hull of a finite set of vertices: there is a set $\mathbf{P} = \{\vec{P}_1, \ldots, \vec{P}_p\} \subseteq \mathbb{P}$ of vertices such that every point in the polyhedron is an affine combination of $\mathbf{P}$. To be able to represent an unbounded polyhedron, it is necessary to consider, additionally, rays, that is, a finite set of directions $\mathbf{R} = \{\vec{R}_1, \ldots, \vec{R}_r\} \subseteq \mathbb{P}$ that can be followed at any length. More precisely, given a pair $[\mathbf{P}, \mathbf{R}]$, put in brackets to avoid any ambiguity with the constraint representation $\langle \mathbf{M}, \vec{C} \rangle$, we assign the following meaning:

$$\gamma([\mathbf{P}, \mathbf{R}]) \stackrel{\text{def}}{=} \left\{ \left(\sum_{j=1}^{p} \alpha_j \vec{P}_j \right) + \left(\sum_{j=1}^{r} \beta_j \vec{R}_j \right) \mid \forall j\colon \alpha_j, \beta_j \geq 0,\ \sum_{j=1}^{p} \alpha_j = 1 \right\} \tag{5.7}$$

Figure 5.2 gives two examples of polyhedra defined by generators. On the left, a bounded polyhedron with generators $[\{\vec{P}_1, \ldots, \vec{P}_5\}, \emptyset]$, and an example affine combination, in red, of these generators. On the right, an unbounded polyhedron with generators $[\{\vec{P}_1, \vec{P}_2, \vec{P}_3\}, \{\vec{R}_1, \vec{R}_2\}]$: a polyhedron point, in red, is an affine combination of $\{\vec{P}_1, \vec{P}_2, \vec{P}_3\}$, translated by some positive factor of either or both rays in $\{\vec{R}_1, \vec{R}_2\}$.

Redundancy. Constraint and generator representations are not unique. Figure 5.3 gives three syntactically different representations for the same polyhedron, the point $(0, 0)$, in logical form. Note that Fig. 5.3.(a) contains four constraints, one of which, $y \geq -5$, is clearly useless as we already know that $y \geq 0$ from $y + x \geq 0$ and $y - x \geq 0$. More generally, a *redundant constraint* is a constraint that can be removed without changing the concretization. A *minimal representation*, as presented in Fig. 5.3.(b), is a representation where no constraint can be removed without changing the concretization.

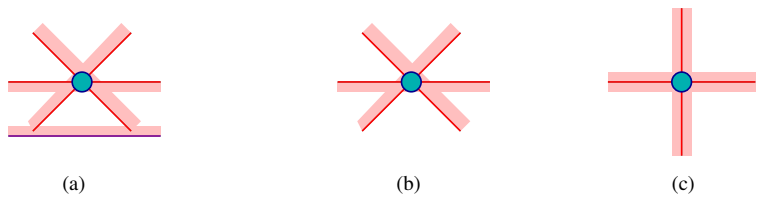


Figure 5.3: Three constraint representations for a single point $(0,0)$: (a) $y+x \geq 0 \wedge y-x \geq 0 \wedge y \leq 0 \wedge y \geq -5$; (b) $y+x \geq 0 \wedge y-x \geq 0 \wedge y \leq 0$; (c) $x \leq 0 \wedge x \geq 0 \wedge y \leq 0 \wedge y \geq 0$

Minimal representations are desirable as they reduce the memory footprint. Minimal representations are not, however, normal forms, which will make testing for equality more involved than for affine equalities. In fact, there can exist polyhedra for which the different minimal representations do not even feature the same number of constraints. This is exemplified by Fig. 5.3.(c) which, as Fig. 5.3.(b), is a minimal representation for $(0,0)$, but has more constraints.

Naturally, the notion of redundant and minimal representations carries to generator representations as well, and we can have spurious, useless vertices and rays.

Empty polyhedron. There is one generator representation of the empty set, with no generator $[\emptyset, \emptyset]$, and infinitely many constraint representations for the empty set. As for intervals, it is easier to use a separate element $\bot^\sharp$ to represent the empty set. In the following, when defining the various operators, we always assume that the arguments are not $\bot^\sharp$. Their generalisation to $\bot^\sharp$ is straightforward; most operators are strict ($F^\sharp(\bot^\sharp) \stackrel{\text{def}}{=} \bot^\sharp$), with the exception of join $\cup^\sharp$ and widening $\triangledown$ where $\bot^\sharp$ is a neutral element.

Absence of Galois connection. Unlike the affine equalities domain, there is no upper bound on the size of polyhedra representations, even minimal ones: we can imagine polyhedra with as many constraints as we want. A classic example is the sequence of regular polygons tangent to a disc $D \stackrel{\text{def}}{=} \{\, (x,y) \mid x^2 + y^2 \leq 1 \,\}$: we can construct a polygon with as many (non-redundant) constraints as we wish. This construction also illustrates the fact that there is no abstraction function α, and so, no Galois connection for polyhedra as, for instance, D has no best abstraction as a polyhedron.

Duality. The constraint and generator representations are not independent; they are linked by a useful notion of *duality*. We only provide some intuition here as the precise development can be technical [Schrijver, 1986, LeVerge, 1992].

Duality is best illustrated on the restricted case of polyhedra cones. A cone is a polyhedron that is defined only by linear constraints, i.e., $\vec{C} = \vec{0}$, and we have $\gamma(\mathbf{M}) \stackrel{\text{def}}{=} \{\, \vec{V} \in \mathbb{P} \mid \mathbf{M} \times \vec{V} \geq \vec{0} \,\}$. The generator representation of a cone contains only a set of rays: $\gamma(\mathbf{R}) \stackrel{\text{def}}{=} \{\, \sum_{j=1}^{r} \beta_j \vec{R}_j \mid \forall j \colon \beta_j \geq 0 \,\}$, with an implicit vertex $\vec{0}$ at the origin. Linear constraints and rays can be seen uniformly as vectors in $\mathbb{P}$. Then, given the constraints of a cone C, interpreting them as generators gives the dual cone C^*. More precisely, the cone

dual C^* of a cone C is defined as: $C^* \stackrel{\text{def}}{=} \{\, \vec{x} \in \mathbb{P} \mid \forall \vec{c} \in C \colon \vec{c} \cdot \vec{x} \geq 0 \,\}$. A classic result states that $C^{**} = C$, hence the generators of C are also the constraints of C^*. This result extends to arbitrary (possibly non-conic) polyhedra, following an encoding of polyhedra into cones as described, for instance, by LeVerge [1992]. In the polyhedra domain, the main use of duality is to justify that, when considering the problem of converting from one representation to the other, we only need to consider the case of converting from constraints to generators. Indeed, by duality, applying the very same algorithm on a generator representation gives back a constraint representation.

Beyond representation, duality extends to operators as well: given an operation on a constraint representation, it can be useful to study the effect of the very same operation on a generator representation. For instance, we will see that, while joining constraints models the intersection, joining generators models the convex hull, hence, intersection and convex hull can be thought as dual operations.

5.3.2 Representation Conversion: Chernikova's Algorithm

As we will see, given the right representation, most operators are extremely simple and efficient. The best representation varies from one operator to the other, so, it is important to have a way to convert from one representation to the other. This conversion operation is complex and expensive: all the algorithmic complexity of the polyhedra domain is crystallized in this operation. The standard algorithm used in polyhedra libraries is due originally to Chernikova [1968] and significantly improved by LeVerge [1992]. We only illustrate its principles here, by presenting a simplified version.

Given a constraint representation $\langle \mathbf{M}, \vec{C} \rangle$, the algorithm constructs an equivalent generator representation $[\mathbf{P}, \mathbf{R}]$ incrementally, by starting from a generator representation of the whole space, and adding the constraints one by one. More precisely, at step 0, the generator representation for the whole space is simply:

$$\begin{cases} \mathbf{P}_0 = \{\, \vec{0} \,\} & \textit{(origin)} \\ \mathbf{R}_0 = \{\, \vec{v}_i,\ -\vec{v}_i \mid 1 \leq i \leq n \,\} & \textit{(axes)} \end{cases}$$

where $\vec{v}_i$ is the basis vector where all the components equal 0, except at position i where it equals 1. Then, step k constructs $[\mathbf{P}_k, \mathbf{R}_k]$ from $[\mathbf{P}_{k-1}, \mathbf{R}_{k-1}]$ by adding the $k-$th constraint in $\langle \mathbf{M}, \vec{C} \rangle$, which we denote as $\vec{M}_k \cdot \vec{V} \geq C_k$. Adding this constraint consists in keeping the generators from $[\mathbf{P}_{k-1}, \mathbf{R}_{k-1}]$ that satisfy the constraint, removing those that do not, and creating new generators by combining them so that they *saturate* the constraint, i.e., lie on or are parallel to it. More precisely:

- If $\vec{P} \in \mathbf{P}_{k-1}$ s.t. $\vec{M}_k \cdot \vec{P} \geq C_k$, keep $\vec{P}$ in $\mathbf{P}_k$ ($\vec{P}$ satisfies the constraint).
  If $\vec{M}_k \cdot \vec{P} < C_k$, discard it.
- If $\vec{R} \in \mathbf{R}_{k-1}$ s.t. $\vec{M}_k \cdot \vec{R} \geq 0$, keep $\vec{R}$ in $\mathbf{R}_k$ ($\vec{R}$ satisfies the constraint).
  If $\vec{M}_k \cdot \vec{R} < 0$, discard it.
  Indeed, a ray satisfies a constraint if, given any point in the polyhedron, we stay in the polyhedron by following the ray.

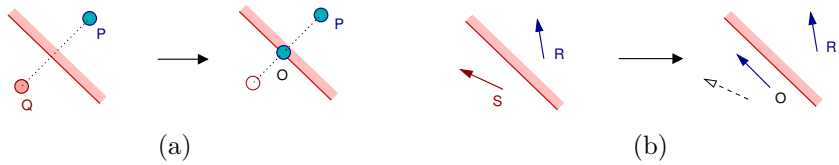


Figure 5.4: Combining two generators to build a new generator $\vec{O}$ in Chernikova's algorithm: (a) combining two vertices, and (b) combining two rays.

- For $\vec{P}, \vec{Q} \in \mathbf{P}_{k-1}$ s.t. $\vec{M}_k \cdot \vec{P} > C_k$ and $\vec{M}_k \cdot \vec{Q} < C_k$, add to $\mathbf{P}_k$:

$$\vec{O} \stackrel{\text{def}}{=} \frac{(C_k - \vec{M}_k \cdot \vec{Q})\vec{P} - (C_k - \vec{M}_k \cdot \vec{P})\vec{Q}}{\vec{M}_k \cdot \vec{P} - \vec{M}_k \cdot \vec{Q}}$$

  ($\vec{P}$ satisfies the constraint strictly and $\vec{Q}$ does not)
  which corresponds to the point at the intersection of the segment $[\vec{P}, \vec{Q}]$ and the hyper-plane supporting the constraint. This is illustrated in Fig. 5.4.(a).

- For $\vec{R}, \vec{S} \in \mathbf{R}_{k-1}$ s.t. $\vec{M}_k \cdot \vec{R} > 0$ and $\vec{M}_k \cdot \vec{S} < 0$, add to $\mathbf{R}_k$:

$$\vec{O} \stackrel{\text{def}}{=} (\vec{M}_k \cdot \vec{R})\vec{S} - (\vec{M}_k \cdot \vec{S})\vec{R}$$

  ($\vec{R}$ satisfies the constraint strictly and $\vec{S}$ does not)
  which corresponds to rotating $\vec{S}$ towards $\vec{R}$ until it is parallel to the constraint. This is illustrated in Fig. 5.4.(b).

- For $\vec{P} \in \mathbf{P}_{k-1}$, $\vec{R} \in \mathbf{R}_{k-1}$ s.t. $\vec{M}_k \cdot \vec{P} > C_k$ and $\vec{M}_k \cdot \vec{R} < 0$, add to $\mathbf{P}_k$:

$$\vec{O} \stackrel{\text{def}}{=} \vec{P} + \frac{C_k - \vec{M}_k \cdot \vec{P}}{\vec{M}_k \cdot \vec{R}}\vec{R}$$

  ($\vec{P}$ satisfies the constraint strictly and $\vec{R}$ does not)
  i.e., move the vertex $\vec{P}$ in the direction of the ray $\vec{R}$ until it touches the hyper-plane defining the constraint.
  The case $\vec{M}_k \cdot \vec{P} < C_k$ and $\vec{M}_k \cdot \vec{R} > 0$ is similar.

One major improvement, mentioned by LeVerge [1992], consists in treating specially equalities: greater efficiency can be obtained by incorporating Gaussian elimination for equalities rather than treating them as pairs of inequalities. Another major improvement is minimization: modern versions of Chernikova's algorithm minimize their representations on the fly, hence solving two difficult problems at once. The key to effective redundancy removal is to maintain the relationship between the constraint and the generator representations we build, by creating a saturation matrix remembering which generators saturate which constraints. A classic result states that generators with a non-maximal set of saturated constraints are redundant.

We mentioned that Chernikova's method can be costly. Indeed, it can have an exponential cost. Unfortunately, this worse-case scenario is unavoidable as, for some polyhedra, one (minimal) representation is exponentially larger than the other. Consider, for instance, the case of an axis-aligned hyper-cube. It has a constraint representation that is linear in $|\mathbb{V}|$, for instance: $\bigwedge_{i=1}^{n} \; 0 \leq V_i \leq 1$. However, the minimal generator representation is exponential in $|\mathbb{V}|$: $\{\, \vec{V} \in \mathbb{P} \mid \forall i \in [1,n] \colon V_i \in \{0,1\} \,\}$.

5.3.3 Abstract Operators

We are now ready to present the operators on polyhedra, assuming both constraint and generator representations are available for all arguments.

Ordering. The ordering $X^\sharp \sqsubseteq^\sharp Y^\sharp$ is semantically equivalent to set inclusion, $\gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$, and is implemented by checking that each generator of $X^\sharp$ satisfies every constraint of $Y^\sharp$:

$$X^\sharp \sqsubseteq^\sharp Y^\sharp \overset{\text{def}}{\iff} \begin{cases} \forall \vec{P} \in \mathbf{P}_{X^\sharp} \colon \mathbf{M}_{Y^\sharp} \times \vec{P} \geq \vec{C}_{Y^\sharp} \\ \forall \vec{R} \in \mathbf{R}_{X^\sharp} \colon \mathbf{M}_{Y^\sharp} \times \vec{R} \;\geq\; \vec{0} \end{cases}$$

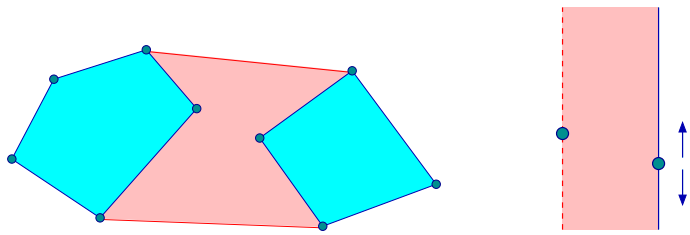
We solve the problem of semantic equality checking without a normal form through checking the double inclusion: $\gamma(X^\sharp) = \gamma(Y^\sharp) \iff (X^\sharp \sqsubseteq^\sharp Y^\sharp) \wedge (Y^\sharp \sqsubseteq^\sharp X^\sharp)$. Technically, $\sqsubseteq^\sharp$ is a pre-order as it does not satisfy the antisymmetry condition but, by identifying elements in $\mathcal{D}^\sharp$ with the same concretization, we obtain a partial order. We even get a lattice $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp)$, although it is not complete (a disc, which is not a polyhedron, can be constructed as the join of an infinite family of polyhedra, as well as the meet of an infinite family of polyhedra).

Intersection. The abstract intersection $\cap^\sharp \overset{\text{def}}{=} \sqcap^\sharp$ is simply joining constraint sets, and it is an exact operator:

$$X^\sharp \sqcap^\sharp Y^\sharp \overset{\text{def}}{=} \left\langle \begin{bmatrix} \mathbf{M}_{X^\sharp} \\ \mathbf{M}_{Y^\sharp} \end{bmatrix}, \begin{bmatrix} \vec{C}_{X^\sharp} \\ \vec{C}_{Y^\sharp} \end{bmatrix} \right\rangle$$

Join. The abstract union $\cup^\sharp \overset{\text{def}}{=} \sqcup^\sharp$ necessarily conveys some approximation as the set union of two polyhedra may not be a polyhedron. Although there is no abstraction function α, there always exists a smallest polyhedron that contains two polyhedra.

Given two *bounded* polyhedra $P_1, P_2 \subseteq \mathbb{P}$, this join is the convex hull P, which is the set of points obtained by a convex combination of points in P_1 and P_2: $P \overset{\text{def}}{=} \{\, \lambda \vec{p_1} + (1-\lambda)\vec{p_2} \mid \vec{p_1} \in P_1, \vec{p_2} \in P_2, \lambda \in [0,1] \,\}$. As every point $\vec{p_1} \in P_1$ is a convex combination of P_1's generators and $\vec{p_2} \in P_2$ is a convex combination of P_2's generators, we note that a point $\vec{p} \in P$ in their convex hull is a convex combination of the generators of P_1 and P_2. Hence, the convex hull can be simply obtained by joining the generators of P_1 and P_2. This is illustrated in Fig. 5.5.(a).



(a) (b)

Figure 5.5: Abstract join of two polyhedra: (a) join of two bounded polyhedra, and (b) join of a point and an unbounded line.

This generalizes to possibly unbounded polyhedra, by joining rays as well as vertices:

$$X^\sharp \sqcup^\sharp Y^\sharp \stackrel{\text{def}}{=} \left[\left[\mathbf{P}_{X^\sharp} \quad \mathbf{P}_{Y^\sharp}\right], \left[\mathbf{R}_{X^\sharp} \quad \mathbf{R}_{Y^\sharp}\right]\right]$$

In the presence of rays, however, the result is not exactly the convex hull, but rather the topological closure of the convex hull. This is illustrated in Fig. 5.5.(b): the convex hull of a point $(0,0)$ and a line $\{\,(1,y) \mid y \in \mathbb{R}\,\}$ is not representable as a polyhedron as it is not closed: it misses the points in $\{\,(0,y) \mid y \neq 0\,\}$. The join $\sqcup^\sharp$ adds these closure points.

Note that the join, as well as the intersection, can generate redundant constraints or generators from minimal representations, hence, it can be useful to apply a round of Chernikova's algorithm, even when the correct representation is available, in order to ensure that the representations stays minimal.

Abstract conditions. Similarly to the case of the affine equalities domain, we handle precisely assignments and tests that can be exactly represented in our domain, and revert, respectively, to non-deterministic assignments and the identity to abstract other assignments and tests.

We can abstract exactly affine inequality tests by adding the test directly to the constraint representation:

$$\mathbb{C}^\sharp[\![\, \textstyle\sum_i \alpha_i V_i \geq \beta \,]\!] X^\sharp \stackrel{\text{def}}{=} \left\langle \begin{bmatrix} \mathbf{M}_{X^\sharp} \\ \alpha_1 \cdots \alpha_n \end{bmatrix}, \begin{bmatrix} \vec{C}_{X^\sharp} \\ \beta \end{bmatrix} \right\rangle$$

Abstract assignments. We can abstract exactly the non-deterministic assignment $V_j \leftarrow [-\infty, +\infty]$ using the generator representation, by simply adding two opposite rays in the two basic vector directions $\vec{v_j}$ and $-\vec{v_j}$. Indeed, assigning a random value r to V_j can be viewed as adding some positive or negative value $r - c$ to the current value c of V_j. Hence, we define:

$$\mathbb{S}^\sharp[\![\, V_j \leftarrow [-\infty, +\infty] \,]\!] X^\sharp \stackrel{\text{def}}{=} \left[\, \mathbf{P}_{X^\sharp}, \left[\mathbf{R}_{X^\sharp} \quad \vec{v}_j \quad (-\vec{v}_j)\right] \,\right]$$

Finally, we can also abstract exactly affine assignments $\mathbb{S}^\sharp[\![V_j \leftarrow \sum_i \alpha_i V_i + \beta]\!] X^\sharp$ using the constraint representation. We use the same technique as we did for the affine equalities domain, with two cases depending on α_j:

- In case $\alpha_j \neq 0$, the assignment is invertible and we simply replace, in every constraint, V_j with $(V_j - \sum_{i \neq j} \alpha_i V_i - \beta)/\alpha_j$, which expresses the old value of V_j as a function of the new value.
- In case $\alpha_j = 0$, the assignment is non-invertible and we revert to forgetting the value of V_j, which is not used to express the new value of V_j anyway, and add an equality constraint, modelled as a pair of inequalities:

$$\begin{aligned}&\mathbb{S}^\sharp[\![V_j \leftarrow \textstyle\sum_i \alpha_i V_i + \beta]\!] \stackrel{\text{def}}{=} \\ &\qquad \mathbb{C}^\sharp[\![V_j = \textstyle\sum_i \alpha_i V_i + \beta]\!] \circ \mathbb{S}^\sharp[\![V_j \leftarrow [-\infty, +\infty]]\!]\end{aligned}$$

An alternate view of an affine assignment $\mathbb{S}^\sharp[\![V_j \leftarrow \sum_i \alpha_i V_i + \beta]\!] X^\sharp$ is as an affine transformation on the set of points in $\gamma(X^\sharp)$. Given a polyhedron in generator representation, as any such a point is already a linear expression of the generators, it is possible to obtain the polyhedron after the assignment by applying the transformation to the generators only. More precisely, we apply the affine transformation to the vertices, while we apply the associated linear transformation $V_j \leftarrow \sum_i \alpha_i V_i$ to the rays.

5.3.4 Convergence Acceleration

We need a widening as the polyhedra domain has infinite strictly increasing chains. We use the same idea as for the interval domain except that, instead of putting unstable bounds to infinity, i.e., removing bounds, we remove unstable constraints.

Naive widening. A first idea for the widening is thus: $X^\sharp \triangledown Y^\sharp \stackrel{\text{def}}{=} \{ c \in X^\sharp \mid Y^\sharp \sqsubseteq \{c\} \}$, viewing the abstract element $X^\sharp$ as sets of constraints, and $Y^\sharp \sqsubseteq \{c\}$ meaning that the polyhedron $Y^\sharp$ is included in the half-space defined by the constraint c from the constraint representation of $X^\sharp$. This widening indeed satisfies Def. 2.17, as it returns an upper bound and iterations with widening necessarily terminate as the set of constraints decreases. However, it is not entirely satisfactory precision-wise, as shown by the following example.

Example 5.10 (Representation-dependent widening)**.** *Consider computing $X^\sharp \triangledown Y^\sharp$ where $X^\sharp$ and $Y^\sharp$ are defined by constraint sets as follows: $X^\sharp \stackrel{\text{def}}{=} \{x \geq 1, y \geq 1, y \leq 1\}$ and $Y^\sharp \stackrel{\text{def}}{=} \{x \geq y, y \geq 1, y \leq 2\}$. Then, our naive widening definition outputs $X^\sharp \triangledown Y^\sharp = \{x \geq 1, y \geq 1\}$.*

Note now that another constraint set that represents the exact same half-line as $X^\sharp$ is $X^{\sharp\prime} \stackrel{\text{def}}{=} \{x \geq y, y \geq 1, y \leq 1\}$. With this representation, the widening becomes $X^{\sharp\prime} \triangledown Y^\sharp = \{x \geq y, y \geq 1\}$. Hence, the widening is not fully semantics: it does not depend only on the polyhedron as a set of points, but also on the set of constraints chosen to represent this set of points.

It is also interesting to note that $X^{\sharp\prime} \triangledown Y^\sharp$ is more precise than $X^\sharp \triangledown Y^\sharp$ as the constraint $x \geq y$ gives a more precise information than $x \geq 1$. Note that the constraint $x \geq y$ is also satisfied in $X^\sharp$, but it is not kept in $X^\sharp \triangledown Y^\sharp$ because, unlike for $X^{\sharp\prime}$ in $X^{\sharp\prime} \triangledown Y^\sharp$, the constraint does not appear syntactically in $X^\sharp$. ♦

Semantic widening. In order to solve this problem, Cousot and Halbwachs [1978] proposed a refined widening that is able to take into account both the constraints in the left and in the right argument. More precisely, $X^\sharp \triangledown Y^\sharp$ not only keeps stable constraints from $X^\sharp$, but also keeps constraints from $Y^\sharp$, provided that they can be swapped with a constraint from $X^\sharp$ without changing $\gamma(X^\sharp)$:

$$\begin{aligned} X^\sharp \triangledown Y^\sharp \stackrel{\text{def}}{=} \; & \{\, c \in X^\sharp \mid Y^\sharp \sqsubseteq^\sharp \{c\} \,\} \cup \\ & \{\, c \in Y^\sharp \mid \exists c' \in X^\sharp \colon X^\sharp =^\sharp (X^\sharp \setminus \{c'\}) \cup \{c\} \,\} \end{aligned}$$

where $X^\sharp =^\sharp Y^\sharp$ means $(X^\sharp \sqsubseteq^\sharp Y^\sharp) \wedge (Y^\sharp \sqsubseteq^\sharp X^\sharp)$, i.e., $\gamma(X^\sharp) = \gamma(Y^\sharp)$.

It is easy to see that this widening still outputs an upper bound of $X^\sharp$ and $Y^\sharp$, as we only keep constraints satisfied by both $X^\sharp$ and $Y^\sharp$. This operator also enforces termination as, although the set of constraints is not strictly decreasing, any new constraint added to the iterates is traded for an equivalent constraint (assuming, as additional technical condition, that the arguments are minimal, so that we do not trade one constraint for several redundant ones). Less obviously, this widening can be proved to be semantic, i.e., independent from the chosen representation [Bagnara et al., 2005a]. Intuitively, the widening is more precise as it chooses, among the possible equivalent constraint representations of the first argument, the one that maximizes the number of constraints that are kept, based on the second argument.

Advanced widenings. Similarly to the case of the interval domain, we can design more complex, more precise widenings. We saw in Sect. 4.7.3 a widening with thresholds that allows gradually relaxing bounds instead of setting them to infinity immediately. For polyhedra, a widening with thresholds is parameterized by a finite set C of constraints. Then, $X^\sharp \triangledown Y^\sharp$ keeps the constraints from $X^\sharp$ (or equivalent ones from $Y^\sharp$) stable in $Y^\sharp$, as before, but also includes all the threshold constraints $c \in C$ satisfied by both $X^\sharp$ and $Y^\sharp$. This gives the iterates the opportunity to check whether the constraints in C are useful in establishing an inductive loop invariant, before discarding them at the next iterate if they are not. More advanced widenings for polyhedra have been proposed by Bagnara et al. [2005a].

Decreasing iterations. The polyhedra domain also has infinite decreasing chains, but it does not feature any standard narrowing *per se*. As mentioned in Sect. 4.7, this is not a real problem: we can replace the application of a narrowing $\triangle$ until stabilization with a limited number of applications of the intersection $\sqcap^\sharp$. After a selected number of iterations has been reached, we keep the result, without waiting further for stabilization.

5.3.5 Polyhedral Analysis Example

Example 5.11 (Polyhedra analysis)**.** *Consider the following program, which is a slightly more complex version of Ex. 5.2 requiring a polyhedral relational inductive loop invariant:*

```
X ← 2; I ← 0;
while I < 10 do
    if [0,1] = 0 then X ← X + 2
    else X ← X − 3 endif;
    I ← I + 1
done
```

This program either adds 2 to X or subtract 3 at each loop iteration. At the loop head, increasing iterations with widening give the following sequence:

$$\begin{array}{rcl}
\mathcal{X}_1^\sharp & = & \{X = 2, I = 0\} \\
\mathcal{X}_2^\sharp & = & \{X = 2, I = 0\} \triangledown (\{X = 2, I = 0\} \cup^\sharp \{X \in [-1, 4], I = 1\}) \\
& = & \{X = 2, I = 0\} \triangledown \{\, I \in [0, 1],\ 2 - 3I \leq X \leq 2I + 2 \,\} \\
& = & \{I \geq 0, 2 - 3I \leq X \leq 2I + 2\}
\end{array}$$

Then, a step of decreasing iteration allows, similarly to the interval case (see Ex. 4.5), retrieving a finite upper bound for I:

$$\begin{array}{rcl}
\mathcal{X}_3^\sharp & = & \{X = 2, I = 0\} \cup^\sharp \{\, I \in [1, 10],\ 2 - 3I \leq X \leq 2I + 2 \,\} \\
& = & \{I \in [0, 10],\ 2 - 3I \leq X \leq 2I + 2\}
\end{array}$$

When exiting the loop, applying $\mathbb{C}^\sharp[\![\, I \geq 10 \,]\!]$ allows finding that $I = 10$ when the program stops which, combined with the relation $2 - 3I \leq X \leq 2I + 2$ between X and I, allows finding that $X \in [-28, 22]$ as well. It is interesting to note that, as before, the relational information is synthesized from non-relational ones by the join used to merge two loop iterations at the loop head, while the role of the widening is rather to filter out those relations that do not appear to be inductive. Also, as before, the number of abstract loop iterations (3) is independent from the number of concrete loop iterations (here 10), and is significantly smaller. ♦

5.3.6 Constraint-Only Implementation

Much of the cost of the polyhedra domain comes from the need to convert from one representation to the other. We saw, in particular, the example of the hyper-cube, where the generator representation is exponentially larger than the constraint representation, justifying the cost of the conversion algorithm by the cost of its output. Unfortunately, axis-aligned hyper-cubes are frequent in program analysis: they correspond to an abstract information representable in the interval domain, i.e., bounds for each variables and no relation. There is thus much incentive to abandon the double description method and use solely the constraint representation, as advocated for instance by Simon and King [2005].

In order to present a polyhedra domain based solely on the constraint representation, we must provide alternate algorithms for those abstract operators that currently use

generators: inclusion checking, minimization, non-deterministic assignment, and join — widening uses generators, but only indirectly, when it checks for inclusion or equality. The first two operators can be implemented thanks to *linear programming*, while the two other operators use *Fourier-Motzkin elimination*, both classic methods in affine theory.

Linear programming. A large number of optimisation problems can be stated in the form of linear programming: given a polyhedron $\langle \mathbf{M}, \vec{C} \rangle$ in constraint form and a vector $\vec{v}$, find the optimum $LP(\langle \mathbf{M}, \vec{C} \rangle, \vec{v})$ in the polyhedron of $\vec{p} \cdot \vec{v}$:

$$LP(\langle \mathbf{M}, \vec{C} \rangle, \vec{v}) \stackrel{\text{def}}{=} \min \{ \vec{p} \cdot \vec{v} \mid \mathbf{M} \times \vec{p} \geq \vec{C} \} \tag{5.8}$$

The function to minimize, $\lambda \vec{p}. \vec{p} \cdot \vec{v}$ is called the *objective function*. This problem has been extensively studied and very efficient algorithms have been devised to solve it, such as the pervasive Simplex method. We refer the reader to classic textbooks, such as [Schrijver, 1986], for more information on the subject.

Viewing polyhedra $X^\sharp$, $Y^\sharp$ as sets of constraints, we can exploit linear programming to test the inclusion. Note first that if $LP(X^\sharp, \vec{\alpha}) \geq \beta$, then all the points $\vec{V} \in \gamma(X^\sharp)$ satisfy the constraint $\vec{\alpha} \cdot \vec{V} \geq \beta$. Then, by checking $X^\sharp$ against every constraint in $Y^\sharp$, we can ensure that $X^\sharp \sqsubseteq^\sharp Y^\sharp$, i.e., that $\gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$.

$$X^\sharp \sqsubseteq^\sharp Y^\sharp \iff \forall (\vec{\alpha} \cdot \vec{V} \geq \beta) \in Y^\sharp \colon LP(X^\sharp, \vec{\alpha}) \geq \beta$$

Additionally, the constraint $c \stackrel{\text{def}}{=} (\vec{\alpha} \cdot \vec{V} \geq \beta) \in X^\sharp$ is redundant and can be safely removed from $X^\sharp$ if and only if $LP(X^\sharp \setminus \{c\}, \vec{\alpha}) \geq \beta$. Hence, a simple algorithm to remove redundant constraints is to consider each constraint in turn, and compute a linear programming to see if it can be removed, before considering the next one. Note that this process must be done sequentially and not in parallel: given two constraints $c_1, c_2 \in X^\sharp$, it is possible that any one of c_1 or $_2$ can be removed but not both, so that after noticing that c_1 can be removed, the test on c_2 should be performed against $X^\sharp \setminus \{ c_1 \}$ and not $X^\sharp$ — the later would deduce that c_2 can be removed as well, which is no longer true after having removed c_1.

Although linear programming is consdiered to be efficient, our abstract domain will perform many calls to this procedure. It is thus worth trying cheaper solutions before resorting to general linear programming, such as testing a constraint against the bounding box of the polyhedron. We refer the reader to the work of Simon and King [2005] for additional methods to improve the efficiency of constraint-based operators.

Fourier-Motzkin elimination. Another key operation in logic is quantifier elimination: given a formula $\exists V_j \colon P(V_j)$, it finds a closed form of P where V_j no longer appears. Geometrically, this corresponds to a projection, as we remove one coordinate. Semantically, this corresponds to forgetting the value of a variable, i.e., to a non-deterministic assignment $V_j \leftarrow [-\infty, +\infty]$. The key algorithm to achieve quantifier elimination is *Fourier-Motzkin elimination*. This algorithm is similar to Gauss elimination in that it combines constraints

linearly in order to eliminate some coefficients. However, as we manipulate inequalities and not equalities, we must take care to always multiply by a positive value. Given a set of constraints $X^\sharp$ and a variable V_j, Fourier-Motzkin elimination $FM(X^\sharp, V_j)$ constructs a new constraint system where V_j does not appear. Naturally, it first keeps unchanged the constraints from $X^\sharp$ where V_j does not already appear, but then, it also enriches this set with all possible combinations of pairs of constraints from $X^\sharp$ where V_j has a positive coefficient in one constraint and a negative one in the other constraint, so that multiplying them by some positive value and adding them make V_j disappear, More precisely:

$$\begin{array}{rcl} FM(X^\sharp, V_j) & \stackrel{\text{def}}{=} & \{\, (\sum_i \alpha_i V_i \geq \beta) \in X^\sharp \mid \alpha_j = 0 \,\} \cup \\ & & \{\, (-\alpha_j^-)c^+ + \alpha_j^+ c^- \mid \\ & & \quad c^+ = (\sum_i \alpha_i^+ V_i \geq \beta^+) \in X^\sharp,\ \alpha_j^+ > 0, \\ & & \quad c^- = (\sum_i \alpha_i^- V_i \geq \beta^-) \in X^\sharp,\ \alpha_j^- < 0 \,\} \end{array}$$

This provides an exact abstraction $\mathbb{S}^\sharp[\![\, V_j \leftarrow [-\infty, +\infty] \,]\!] X^\sharp$.

Join. The join is quite similar to the join we defined in the affine equalities domain, in (5.3), which was also a relational domain based only on constraints. A point $\vec{V} \in \mathbb{P}$ is in the convex hull of polyhedra $\langle \mathbf{M^1}, \vec{C^1} \rangle$ and $\langle \mathbf{M^2}, \vec{C^2} \rangle$ if it is a convex combination of points $\vec{V^1}$ and $\vec{V^2}$ in these polyhedra. We express this as the (non-affine) equation system:

$$\begin{cases} \vec{V} = \lambda_1 \vec{V^1} + \lambda_2 \vec{V^2} \\ \mathbf{M^1} \times \vec{V^1} \geq \vec{C^1} \\ \mathbf{M^2} \times \vec{V^2} \geq \vec{C^2} \\ \lambda_1 + \lambda_2 = 1 \\ \lambda_1, \lambda_2 \geq 0 \end{cases}$$

which can be rewritten, similarly to (5.3), after the change of variables $\vec{W^1} \stackrel{\text{def}}{=} \lambda_1 \vec{V^1}$ and $\vec{W^2} \stackrel{\text{def}}{=} \lambda_2 \vec{V^2}$, into the following affine system:

$$\begin{cases} \vec{V} = \vec{W^1} + \vec{W^2} \\ \mathbf{M^1} \times \vec{W^1} \geq \lambda_1 \vec{C^1} \\ \mathbf{M^2} \times \vec{W^2} \geq \lambda_2 \vec{C^2} \\ \lambda_1 + \lambda_2 = 1 \\ \lambda_1, \lambda_2 \geq 0 \end{cases} \tag{5.9}$$

We can now eliminate $\vec{W^1}$, $\vec{W^2}$, λ_1, and λ_2 from this system using Fourier-Motzkin elimination. Benoy et al. [2005] prove that this indeed gives the optimal abstract join, even in the more complex case of unbounded polyhedra. Note that the join performs a large number of Fourier-Motzkin eliminations, generating many redundant constraints in the process. Hence, it is beneficial to apply minimization regularly during this computation. Additional insights on the structure of this problem also allows removing redundant constraints more easily, such as applying Kohler's rule, as advocated by Simon and King [2005].

5.3.7 Conversion with Intervals

When interested in inferring variable bounds, the interval and the polyhedra domains stand at opposite ends of the cost versus precision spectrum, and it might be difficult to know which one to choose for a given analysis. We can, however, achieve a better flexibility if we can switch dynamically from one domain to the other. For this, it is sufficient to design operators able to convert from one abstract representation to the other. To be sound, when the abstract element cannot be represented in the other domain, we return an over-approximation. We take care to output the smallest over-approximation, i.e., our operators are optimal.

Conversion from intervals. Any element $I^\sharp$ in the interval domain can always be represented exactly as a polyhedron: given an abstract element $I^\sharp$ that associates to each variable V_i an interval $I^\sharp(V_i) = [a_i, b_i]$, we associate a pair of constraints $V_i \geq a_i$ and $V_i \leq b_i$. We denote as $Poly(I^\sharp)$ this polyhedron.

Conversion to intervals. Not all polyhedra can be exactly represented in the interval domain. An efficient way to find the optimal interval element, i.e., the *bounding box*, is to use the generator representation $X^\sharp = [\mathbf{P}, \mathbf{R}]$ of the polyhedron:

- first, compute in the interval domain the join $\sqcup^\sharp_{P \in \mathbf{P}} \vec{P}$ of every vertex $\vec{P}$, viewed as an interval element representing a single point;
- then, for every ray $\vec{R} \in \mathbf{R}$ and for every variable V_i, set V_i's upper bound to $+\infty$ if the $i-$th coordinate of $\vec{R}$ is strictly positive, $R_i > 0$, and set its lower bound to $-\infty$ if $R_i < 0$.

We denote by $Int(X^\sharp)$ this interval element.

Alternatively, in case the generator representation is not available, then the optimal interval element can be constructed by solving a linear programming problem for the upper bound and another one for the lower bound of every variable.

Fallback operators. Another interesting application of abstract domain conversion is to design better fallback operators. Recall that the interval domain features assignment and test operators that can handle arbitrary expressions, even non-affine ones, while the polyhedra domain reverts to coarse abstractions for non-affine expressions. Hence, a more precise fallback solution would exploit the interval operators. We would use, to model an assignment $V \leftarrow e$:

$$\mathbb{S}^\sharp[\![V \leftarrow [-\infty, +\infty]]\!]_{Poly} X^\sharp \cap^\sharp Poly(\mathbb{S}^\sharp[\![V \leftarrow e]\!]_{Int}(Int(X^\sharp)))$$

Note that, although the interval assignment, denoted here as $\mathbb{S}^\sharp[\![V \leftarrow e]\!]_{Int}$, provides a precise value for V, it looses all the relations in the argument. Hence, we combine it, using $\cap^\sharp$, with the fall-back assignment we used in Sect. 5.3.3 that does not provide any

information on the value of V but imports from $X^\sharp$ relations that are still valid after the assignment.

A condition c would be similarly modeled as:

$$X^\sharp \cap^\sharp Poly(\mathbb{C}^\sharp[\![\, c \,]\!]_{Int}(Int(X^\sharp)))$$

and we import all the relations from $X^\sharp$ as we know they are still valid after the test.

5.3.8 Handling Strict Inequalities

While any non-strict affine inequality test can be exactly modeled, this is not the case for strict inequalities. Hence, $\mathbb{C}^\sharp[\![\, \sum_i \alpha_i V_i + \beta > 0 \,]\!]$ must be relaxed into $\mathbb{C}^\sharp[\![\, \sum_i \alpha_i V_i + \beta \geq 0 \,]\!]$, which is a sound, but non-exact approximation.

Alternatively, it is possible to increase the expressiveness of polyhedra to include both strict and non-strict inequalities. A classic technique, reviewed for instance by Bagnara et al. [2002], consists in adding a synthetic variable V_ϵ, used to encode strictness. Then, we can encode any polyhedron mixing strict and non-strict constraints over $\mathbb{V}$ as a polyhedron with only non-strict constraints over $\mathbb{V}_\epsilon \stackrel{\text{def}}{=} \mathbb{V} \cup \{V_\epsilon\}$. More precisely:

$$\begin{array}{lll} \sum_i \alpha_i V_i + \beta \geq 0 & \text{is represented as} & \sum_i \alpha_i V_i + 0 V_\epsilon + \beta \geq 0 \\ \sum_i \alpha_i V_i + \beta > 0 & \text{is represented as} & \sum_i \alpha_i V_i - c V_\epsilon + \beta \geq 0 \end{array}$$

where $c > 0$ is an arbitrary positive constant. To tie a constraint system over $\mathbb{V}_\epsilon$ to a set of points over $\mathbb{V}$, we need to slightly change the concretization into $\gamma_>$ as follows:

$$\gamma_>(X^\sharp) \stackrel{\text{def}}{=} \{\, (V_1, \ldots, V_n) \in \mathbb{P} \mid \exists V_\epsilon > 0 \colon (V_1, \ldots, V_n, V_\epsilon) \in \gamma(X^\sharp) \,\}$$

where γ is the regular concretization (5.6).

For the most part, the algorithms we designed for regular polyhedra work correctly, that is, applying them while considering V_ϵ as a regular variable also gives sound results with respect to our new concretization $\gamma_>$. There are however small technical difficulties concerning minimization and more advanced widenings, which require enriching the generator representation to remember which vertices actually belong to the polyhedron, and which belong only in the closure of the polyhedron, which are now distinct objects. We refer the reader to the work of Bagnara et al. [2002] for technical details about these changes.

5.3.9 Handling Integers

As in the affine equalities domain, we have assumed that $\mathbb{I} = \mathbb{Q}$ or $\mathbb{R}$ in order to apply classic linear algebra and safely manipulate constraints and vectors. In order to analyze programs with integer-valued variables, we have the same two choices as for affine equalities in Sect. 5.2.5.

Modeling integers as rationals. The first, and simplest choice is to keep using the same algorithms, using as coefficients exact rationals, but change our concretization to convey the fact that the program variables have only integer values. We reuse $\gamma_{\mathbb{Z}}$ from (5.4):

$$\gamma_{\mathbb{Z}}(X^\sharp) \stackrel{\text{def}}{=} \gamma(X^\sharp) \cap \mathbb{Z}^{|\mathbb{V}|}$$

where γ is the regular polyhedra concretization (5.6). As before, we maintain soundness, but we lose exactness and optimality of operators, which hold with respect to γ but no longer to $\gamma_{\mathbb{Z}}$. For instance, Ex. 5.9 describing the non-exactness of the assignment $V_0 \leftarrow 0$ on the set of points defined by the affine equality $2V_0 = V_1$, still holds in the polyhedra domain — it gives a non-optimal result.

Modeling integer sets using relational polyhedra results in a loss of precision. However, there are some situations where it is easy to exploit the "integerness" information to gain back a little precision. Consider, for instance, a constraint such as $2V_0 + 4V_1 \geq 3$. As we know that both V_0 and V_1 are integer, then $2V_0 + 4V_1$ is necessarily even, and we can strengthen the constraint into $2V_0 + 4V_1 \geq 4$. More generally, assuming that we have reduced the constraints to the form $\sum_i \alpha_i V_i \geq \beta$ where all the α_i are integers, we can increase β to the next multiple of the greatest common denominator $\gcd_i \alpha_i$. Such strengthening is not sufficient to recover the optimality or exactness for all cases, but it does improve the precision in practice, at a very low cost. Such methods are implemented in libraries, such as Apron [Jeannet and Miné, 2009].

If even more precision needs to be achieved, more heavyweight methods, such as integer linear programming, could be employed, but with a greater impact on performance: integer linear programming is NP-hard, whereas regular linear programming is polynomial. Classic techniques, such as Gomory's cut, described for instance by Schrijver [1986], can generate additional, non-trivial constraints that exploit the "integerness" property and shrink the polyhedron closer to the integer point lattice, at the cost of increasing the size of the constraint representation.

Improving the expressiveness. Some operations, such as projection, can no longer be exact when switching to integers, as their result is not expressible as a polyhedron. Thus, a second choice consists in increasing the expressiveness of the domain, and represent exactly the projection. Note, however, that, if we are able to exactly model projection, we can also model the join exactly (at least for bounded polyhedra) even though, in the rational world, projection was exact but not join. Indeed, given polyhedra P_1 and P_2, viewed as logic formulas, and a fresh variable V, consider the convex hull P of $(V = 0) \wedge P_1$ and $(V = 1) \wedge P_2$. Then, $\gamma_{\mathbb{Z}}(P)$ is exactly the set of integer points that satisfy the formula $((V = 0) \wedge P_1) \vee ((V = 1) \wedge P_2)$. Its projection $\exists V\colon P$, when represented exactly, gives exactly $P_1 \vee P_2$. The expressiveness of our domain then escalates quickly, so that it can represent exactly formulas in Presburger arithmetic.

While Presburger arithmetic is a decidable theory, its cost is super-exponential, making it rather impractical. Nevertheless, effective abstract operators have been designed for it, such as the Omega test by Pugh [1992] or automata-based approaches by Bartzis and

Bultan [2003], including a widening operator [Bartzis and Bultan, 2004].

5.4 The Zone and Octagon Domains

For many programs, the interval domain is insufficient to provide precise bounds for the variables, which motivated our introduction of the polyhedra domain. Unfortunately, the polyhedra domain is much more costly: it has been observed to have exponential cost in practice [Nguyen Que, 2010], while interval domain operations are linear, at worst, in the number of variables. This motivated the introduction of so-called *weakly relational* abstract domains, that is, relational domains in-between, both in expressiveness and in cost, between intervals and polyhedra. They allow trading precision for efficiency.

The first of these domains we present here is the so-called *zone domain*, introduced by Miné [2001], which can represent only very restricted forms of affine invariants, bounds of variable differences, but has a quadratic cost on its memory representation (in the number of variables) and cubic worst-case time cost. Its algorithms are based on radically different principles than polyhedra: we eschew the double description method, Chernikova's algorithm, linear programming, and Fourier-Motzkin elimination in favor of graph-based algorithms modeling constraint propagation.

5.4.1 Representation

In this domain, we revert to $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$, i.e., our domain can natively represent sets of integer-valued memory states, as well as rational and real ones, without any change in algorithms nor loss of precision. The domain infers invariants of the form:

$$\begin{array}{l} \bigwedge_{i,j}\ V_j - V_i \leq m_{ij} \\ \bigwedge_i\ a_i \leq V_i \leq b_i \end{array}$$

Note that the simple loop example from Ex. 5.2: "$I \leftarrow 1$; $X \leftarrow 0$; **while** $I \leq 1000$ **do** $I \leftarrow I + 1$; $X \leftarrow X + 1$ **done**" only features invariants of this form, e.g., $(I = X + 1) \wedge (I \in [0, 1000])$ for the loop invariant. It can be analyzed precisely in the zone domain, without resorting to the, more costly, polyhedra domain.

Potential graphs. We first focus on constraints of the form $V_j - V_i \leq m_{ij}$ only. They are called *potential constraints*, because any solution of a conjunction of such constraints is defined up to a constant. Given a solution $(v_1, \ldots, v_n)$, adding the same value c to each coordinate gives us another solution: $(v_1 + c, \ldots, v_n + c)$.

A conjunction of potential constraints $\bigwedge_{ij} V_j - V_i \leq m_{ij}$ can be represented as an oriented weighted graph, with one node for each variable $V_i \in \mathbb{V}$, and an arc $V_i \xrightarrow{m_{ij}} V_j$ from V_i to V_j with weight $m_{ij} \in \mathbb{I}$ for every constraint $V_j - V_i \leq m_{ij}$ in the conjunction. If there is no constraint on $V_j - V_i$, there is no arc from V_i to V_j. Note that if two constraints $V_j - V_i \leq m_{ij}$ and $V_j - V_i \leq m'_{ij}$ exist, it is sufficient to keep the strongest one: $V_j - V_i \leq \min(m_{ij}, m'_{ij})$, hence, in the following, we assume that there is at most one arc from a given V_i to a given V_j.

Example 5.12 (Potential graph)**.** *Figure 5.6.(a) gives an example potential graph, and the potential constraints it represents in Fig. 5.6.(c).* ♦

Difference bound matrices. An equivalent representation for potential constraints is using matrices. A *difference bound matrix*, or DBM, $\mathbf{m}$ is a $n \times n$ square matrix, where $n \stackrel{\text{def}}{=} |\mathbb{V}|$, with elements in $\mathbb{I} \cup \{+\infty\}$, where:

- $m_{ij} \in \mathbb{I}$ denotes a constraint $V_j - V_i \leq m_{ij}$;
- $m_{ij} = +\infty$ denotes the absence of any constraint of the form $V_j - V_i \leq c$, that is, there is no upper bound on $V_j - V_i$ or, equivalently, the upper bound is $+\infty$.

Such a DBM $\mathbf{m}$ is actually the adjacency matrix of the potential graph representing the same set of constraints. We will use matrices and graphs interchangeably: while graphs provide additional insights on some operations, some others are better described on matrices. Additionally, when the constraint sets are dense, which will actually often be the case, a matrix representation may be more compact in memory, and allows efficient access.

A DBM $\mathbf{m}$ represents the following set of points in $\mathbb{P}$:

$$\gamma(\mathbf{m}) \stackrel{\text{def}}{=} \{\, (v_1, \ldots, v_n) \in \mathbb{P} \mid \forall i, j \in [1, n] \colon v_j - v_i \leq m_{ij} \,\} \tag{5.10}$$

Example 5.13 (DBM)**.** *Figure 5.6.(b) gives the DBM equivalent to the potential graph in Fig 5.6.(a) and the potential constraints it represents in Fig. 5.6.(c). The concretization of Fig. 5.6.(a)–(c) is given in Fig. 5.6.(d).* ♦

Representing zones. In order to be at least as expressive as intervals, we add to potential constraints unary constraints, able to represent variable bounds: $V_i \in [a_i, b_i]$. To seamlessly introduce these constraints into potential graphs and DBM, we view them as potential constraints using a special variable V_0, which is assumed to be a constant set to zero. Our abstract elements are now $(n+1) \times (n+1)$ matrices for n actual program variables. More precisely:

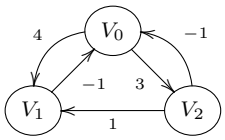
- $V_i \leq b_i$ is represented as $V_i - V_0 \leq b_i$, i.e., $m_{0i} = b_i$;
- $V_i \geq a_i$ is represented as $V_0 - V_i \leq -a_i$, i.e., $m_{i0} = -a_i$.

The concretization is updated to reflect this new meaning:

$$\gamma(\mathbf{m}) \stackrel{\text{def}}{=} \{\, (v_1, \ldots, v_n) \in \mathbb{P} \mid \forall i, j \in [0, n] \colon v_j - v_i \leq m_{ij} \wedge v_0 = 0 \,\} \tag{5.11}$$

The shapes defined by such constraints are called *zones*. From now on, we will use this zone concretization and will not consider the potential constraint concretization from (5.10) anymore.

Example 5.14 (Zones)**.** *Figure 5.6.(e) gives the zone constraints represented by the potential graph in Fig. 5.6.(a) and the DBM in Fig. 5.6.(b). We simply replace V_0 with 0 in Fig. 5.6.(c). Figure 5.6.(f) gives the set of points enclosed by this zone, which is the prism from Fig. 5.6.(d) intersected with the plane $V_0 = 0$.* ♦



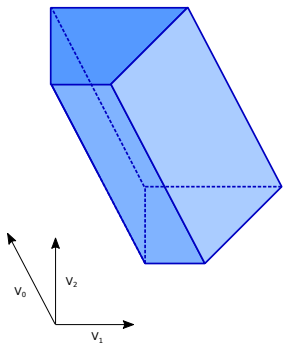
(a)

	V_0	V_1	V_2
V_0	$+\infty$	4	3
V_1	-1	$+\infty$	$+\infty$
V_2	-1	1	$+\infty$

(b)

$$\begin{cases} V_1 - V_0 \leq 4 \\ V_0 - V_1 \leq -1 \\ V_2 - V_0 \leq 3 \\ V_0 - V_2 \leq -1 \\ V_1 - V_2 \leq 1 \end{cases}$$

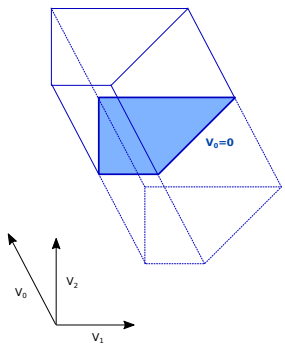
(c)



(d)

$$\begin{cases} V_1 \leq 4 \\ V_1 \geq 1 \\ V_2 \leq 3 \\ V_2 \geq 1 \\ V_1 - V_2 \leq 1 \end{cases}$$

(e)



(f)

Figure 5.6: A potential graph (a) and its equivalent difference bound matrix (b); their interpretation as a set of potential constraints over variables $\{V_0, V_1, V_2\}$ (c) and the associated set of points by γ (5.10) in (d); their interpretation as zone constraints over $\{V_1, V_2\}$ (e) and the associated set of points by γ (5.11) in (f).

5.4.2 Ordering and Closure

Lattice structure. The set $\mathbb{I} \cup \{+\infty\}$ is totally ordered with the regular arithmetic order $\leq$. The point-wise extension of this order on DBM with coefficients in $\mathbb{I} \cup \{+\infty\}$ gives a partial order, and we have actually a lattice structure, with max as join and min as meet. We naturally have a greatest element $\top^\sharp$, where all coefficients equal $+\infty$, but no least element, as $\mathbb{I} \cup \{+\infty\}$ has no least element. Hence, we add a least element $\bot^\sharp$. More precisely, we have the following lattice $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot^\sharp, \top^\sharp)$:

$$
\begin{array}{rcl}
\mathcal{D}^\sharp & \stackrel{\text{def}}{=} & (\mathbb{I} \cup \{+\infty\})^{(n+1)\times(n+1)} \cup \{\bot^\sharp\} \\
\forall i,j\colon \left[\top^\sharp\right]_{ij} & \stackrel{\text{def}}{=} & +\infty \\
\mathbf{m} \sqsubseteq^\sharp \mathbf{n} & \stackrel{\text{def}}{\iff} & \forall i,j\colon m_{ij} \leq n_{ij} \\
\forall i,j\colon \left[\mathbf{m} \sqcup^\sharp \mathbf{n}\right]_{ij} & \stackrel{\text{def}}{=} & \max(m_{ij}, n_{ij}) \\
\forall i,j\colon \left[\mathbf{m} \sqcap^\sharp \mathbf{n}\right]_{ij} & \stackrel{\text{def}}{=} & \min(m_{ij}, n_{ij})
\end{array}
$$

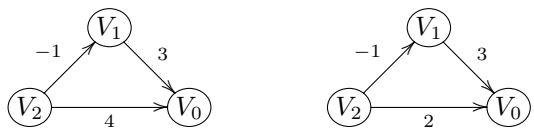
Naturally, γ (5.11) is monotonic, i.e., if $\mathbf{m} \sqsubseteq^\sharp \mathbf{n}$, then $\gamma(\mathbf{m}) \subseteq \gamma(\mathbf{n})$.

As for the case of intervals, the lattice is complete when min and max exist for arbitrary (possibly infinite) families of numbers in $\mathbb{I}$: this is the case for integers and reals, but not rationals.

Normal form. We note that γ is not one-to-one: we can have two different matrices, $\mathbf{m} \neq \mathbf{n}$, representing the same set: $\gamma(\mathbf{m}) = \gamma(\mathbf{n})$. Likewise, $\gamma(\mathbf{m}) \subseteq \gamma(\mathbf{n})$ does not necessarily imply $\mathbf{m} \sqsubseteq^\sharp \mathbf{n}$. We need a notion of normal form in order to effectively test, in the abstract, for equality and inclusion.

Example 5.15 (Local propagation)**.** *Consider the following two equation systems, with associated potential graph:*

$$
\begin{cases} V_0 - V_1 \leq 3 \\ V_1 - V_2 \leq -1 \\ V_0 - V_2 \leq 4 \end{cases}
\qquad\qquad
\begin{cases} V_0 - V_1 \leq 3 \\ V_1 - V_2 \leq -1 \\ V_0 - V_2 \leq 2 \end{cases}
$$



(a) *(b)*

The graphs are not equal, and yet, they represent the same set of points through γ. Actually, we can see that (b) is smaller, for $\sqsubseteq^\sharp$, than (a), giving a tighter representation. This is not necessarily the case, and there are examples of graphs that are not even comparable and yet represent the same set.

On our example, we note that (b) can be constructed from (a) by summing the constraints $V_0 - V_1 \leq 3$ and $V_1 - V_2 \leq -1$ that already exist in (a), to get $V_0 - V_2 \leq 2$ that

only exists in (b). This transformation has strengthened the constraints without changing the concretization. On the potential graph, we have replaced the weight on the arc from V_2 to V_0, originally 4, with the sum of weights along the path $V_2 \xrightarrow{-1} V_1 \xrightarrow{3} V_0$, i.e., 2. ♦

The preceding example suggests a natural operation on systems of zone constraints: constraint propagation, by adding two or more constraints such that the sum simplifies into a zone constraint. This also corresponds to adding the weight of edges along a path in the graph. To construct our normal form, we will propagate the constraints until a fixpoint is reached. From a logical point of view, this operation is a *saturation*. From a graph point of view, we construct the *shortest-path closure*.

Formally, given a DBM $\mathbf{m}$, its shortest-path closure is denoted as $\mathbf{m}^*$ and defined as:

$$\begin{array}{rl} \forall i \neq j\colon & m^*_{ij} \stackrel{\text{def}}{=} \min_{\forall N\colon \langle i=i_1,\ldots,i_N=j\rangle} \sum_{k=1}^{N-1} m_{i_k\, i_{k+1}} \\ \forall i\colon & m^*_{ii} \stackrel{\text{def}}{=} 0 \end{array} \tag{5.12}$$

Note that the minimum is defined if and only if the graph does not contain any cycle with a strictly negative total weight; otherwise, we can constructs paths of increasing lengths with arbitrary negative total weight by simply repeating the cycle. When no such cycle exists, only simple paths need to be considered in the sum, and the result is a matrix with elements in $\mathbb{I} \cup \{+\infty\}$. Interestingly, a cycle with strictly negative total weight corresponds to a set of constraints the sum of which is a constraint of the form $0 \leq c$ for $c < 0$, as all the left-hand sides cancel each others, i.e., an unsatisfiable constraint. We actually have an equivalence. When there is no such cycle, then, the shortest-path closure, $\mathbf{m}^*$, is the smallest matrix, for $\sqsubseteq^\sharp$, which represents $\gamma(\mathbf{m})$. A geometric intuition is that every constraint in $\mathbf{m}^*$ saturates the set $\gamma(\mathbf{m})$, i.e., for every m_{ij}, there exists a point in $(v_1, \ldots, v_n) \in \gamma(\mathbf{m})$ such that $v_j - v_i = m_{ij}$. The following theorem summarizes these findings and applies them to implement equality and inclusion tests:

Theorem 5.1 (Normal form, equality and inclusion checking)**.**

1. $\gamma(\mathbf{m}) = \emptyset \iff$ *the graph of* $\mathbf{m}$ *has a cycle with strictly negative weight.*
2. *If* $\gamma(\mathbf{m}) \neq \emptyset$*, the shortest-path* $\mathbf{m}^*$ *is a normal form:*

$$\mathbf{m}^* = \min_{\sqsubseteq^\sharp} \{\, \mathbf{n} \mid \gamma(\mathbf{m}) = \gamma(\mathbf{n}) \,\} \tag{5.13}$$

3. *If* $\gamma(\mathbf{m}), \gamma(\mathbf{n}) \neq \emptyset$*, then* $\gamma(\mathbf{m}) = \gamma(\mathbf{n}) \iff \mathbf{m}^* = \mathbf{n}^*$.
4. *If* $\gamma(\mathbf{m}), \gamma(\mathbf{n}) \neq \emptyset$*, then* $\gamma(\mathbf{m}) \subseteq \gamma(\mathbf{n}) \iff \mathbf{m}^* \sqsubseteq^\sharp \mathbf{n}$. ■

Hence, to check for equality, it is sufficient to compare syntactically the normal forms. To check inclusion, we note that $\mathbf{m}^* \sqsubseteq^\sharp \mathbf{n} \iff \mathbf{m}^* \sqsubseteq^\sharp \mathbf{n}^*$ because $\mathbf{n}^* \sqsubseteq^\sharp \mathbf{n}$, so that we do not need the normal form of the second argument.

As we will see, the normal form also plays an important role to ensure the precision of several abstract operators. Intuitively, normalizing a matrix makes implicit constraints

explicit, i.e., an upper bound on $V_j - V_i$ in $\gamma(\mathbf{m})$ that could only be discovered by adding several constraints from $\mathbf{m}$ will now be explicit in m_{ij}. Another intuition is that $\mathbf{m}^*$ is solving linear programming problems [Schrijver, 1986] over a zone $\gamma(\mathbf{m})$ for all objective functions of the form $V_j - V_i$ as m^*_{ij} is indeed, by Thm. 5.1.2, $\max\{\, v_j - v_i \mid (v_1, \ldots, v_n) \in \gamma(\mathbf{m})\,\}$.

Closure algorithm. To construct the normal form, we need a so-called *all pairs shortest path closure* algorithm. Several algorithms exist. We employ here the Floyd–Warshall algorithm, a classic algorithm described for instance in Cormen et al. [2001], which is simple, has a very deterministic cubic time complexity, and can be extended to handle more complex constraints, as we will see in Sect. 5.4.5. As we will see shortly, the algorithm is also useful in the case where the graph has negative cycles and no shortest path closure exists. The algorithm can be described as computing $\mathbf{m}^{n+1}$ through the following sequence:

$$\begin{aligned} &\mathbf{m}^0 \stackrel{\text{def}}{=} \mathbf{m} \\ &\forall i, j, k\colon m^{k+1}_{ij} \stackrel{\text{def}}{=} \min(m^k_{ij}, m^k_{ik} + m^k_{kj}) \end{aligned} \tag{5.14}$$

i.e., for each node V_k in turn, it checks in parallel for all pairs (V_i, V_j) whether it is faster to go through V_k when going from V_i to V_j. Note that each V_k needs only be considered once, hence the cubic cost. This also corresponds to applying, may times, the local constraint propagation we saw in Ex. 5.15. We then have the following properties:

Theorem 5.2 (Floyd-Warshall algorithm)**.**

1. *If $\gamma(\mathbf{m}) \neq \emptyset$, then $\mathbf{m}^* = \mathbf{m}^{n+1}$ as computed in (5.14).*
   (up to setting $\forall i\colon m^*_{ii} = 0$)

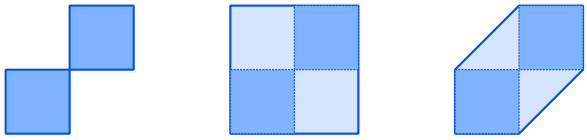
2. $\gamma(\mathbf{m}) = \emptyset \iff \exists i\colon m^{n+1}_{ii} < 0.$ ■

By applying the Floyd–Warshall algorithm to an arbitrary DBM we can check whether it represents an empty set by looking at the diagonal — i.e., constraints of the form $V_i - V_i \leq m_{ii}$ — in which case we return $\bot^\sharp$; otherwise, we return the result of the algorithm, after resetting the diagonal to 0. In all cases, we obtain the normal form.

Galois connection. In order to construct a Galois connection, we can define the following abstraction function α:

$$\begin{array}{rcll} \alpha(S) & \stackrel{\text{def}}{=} & \bot^\sharp & \text{if } S = \emptyset \\ [\alpha(S)]_{ij} & \stackrel{\text{def}}{=} & \max\{\, v_j - v_i \mid (v_1, \ldots, v_n) \in S \wedge v_0 = 0\,\} & \text{if } S \neq \emptyset \end{array}$$

Naturally, α is only guaranteed to exist if $\mathbb{I} \in \{\mathbb{Z}, \mathbb{R}\}$, but not on rationals, as some rational sets have no rational maximum. In all cases — even when $\mathbb{I} = \mathbb{Q}$ — $\alpha(\gamma(\mathbf{m}))$ exists, and $\mathbf{m}^* = \alpha(\gamma(\mathbf{m}))$, again as a consequence of Thm. 5.1.2.



(a) (b) (c)

Figure 5.7: Two zones reduced to a non-relational box (a); their least upper bound $\sqcup^\sharp$, which is no more precise than the interval join (b); and their abstract union $\cup^\sharp$, which contains relational information and is more precise (c).

5.4.3 Abstract Operators

Union and intersection. As $\mathcal{D}^\sharp$ forms a lattice, we are tempted to implement union and intersection as joins and meets: $\cup^\sharp \stackrel{\text{def}}{=} \sqcup^\sharp$ and $\cap^\sharp \stackrel{\text{def}}{=} \sqcap^\sharp$. This works well with $\cap^\sharp$, and we obtain the exact intersection because, as convex polyhedra, zones are closed under intersection, and $\sqcap^\sharp$ is indeed a conjunction of constraints, keeping only the most strict bound for each expression $V_j - V_i$. For unions, however, we obtain a sound but not optimal abstraction, as shown in the following example:

Example 5.16 (Non-optimal join and abstract union)**.** *Consider the zones $X^\sharp$ and $Y^\sharp$ defined by the constraints:*

$$\begin{array}{lcl} X^\sharp & \stackrel{\text{def}}{=} & (0 \leq V_1 \leq 1) \wedge (0 \leq V_2 \leq 1) \\ Y^\sharp & \stackrel{\text{def}}{=} & (1 \leq V_1 \leq 2) \wedge (1 \leq V_2 \leq 2) \end{array}$$

illustrated in Fig. 5.7.(a). Their least upper bound is $X^\sharp \sqcup^\sharp Y^\sharp = (0 \leq V_1 \leq 2) \wedge (0 \leq V_2 \leq 2)$, and is illustrated in Fig. 5.7.(b). Note that $X^\sharp$ and $Y^\sharp$ are boxes, exactly representable in the interval domain. Then, $X^\sharp \sqcup^\sharp Y^\sharp$ is also a box: the result is no more precise in the zone domain than it would be in the interval domain. The smallest zone, for $\subseteq$, that contains $\gamma(X^\sharp)$ and $\gamma(Y^\sharp)$ is depicted in Fig. 5.7.(c). It additionally features the zone constraint $-1 \leq V_1 - V_2 \leq 1$. ♦

To achieve the optimal abstract union, we must compute the least upper bound on the normal forms:

$$\mathbf{m} \cup^\sharp \mathbf{n} \stackrel{\text{def}}{=} \mathbf{m}^* \sqcup^\sharp \mathbf{n}^*$$

Considering again Ex. 5.16, the closure exposes the constraint $-1 \leq V_1 - V_2 \leq 1$ in both $X^\sharp$ and $Y^\sharp$, hence, their join $\sqcup^\sharp$ after closure will also feature the constraint $-1 \leq V_1 - V_2 \leq 1$.

In fact, we have the following stronger property:

$$\mathbf{m}^* \sqcup^\sharp \mathbf{n}^* = \min_{\sqsubseteq^\sharp} \{ \mathbf{o} \mid \gamma(\mathbf{m}) \cup \gamma(\mathbf{n}) \subseteq \gamma(\mathbf{o}) \}$$

which proves that, not only is $\gamma(\mathbf{m}^* \sqcup^\sharp \mathbf{n}^*)$ the tightest approximation of the join $\gamma(\mathbf{m}) \cup \gamma(\mathbf{n})$, but also that $\mathbf{m}^* \sqcup^\sharp \mathbf{n}^*$ is already in normal form.

Dually, while the intersection $\cap^\sharp \stackrel{\text{def}}{=} \sqcap^\sharp$ does not require normalized arguments, the result may not be normalized, even if both arguments are.

Conditions. We can model easily and exactly tests of the form $V_j - V_i \leq c$ (and respectively $V_i \leq c$, $V_i \geq c$): it is sufficient to update the matrix element at position (i, j) — respectively $(0, i)$ and $(i, 0)$. For instance:

$$\forall k, l\colon \left[\mathbb{C}^\sharp[\![\, V_j - V_i \leq c \,]\!]\mathbf{m}\right]_{kl} \stackrel{\text{def}}{=} \begin{cases} \min(\mathbf{m}_{kl}, c) & \text{if } (i,j) = (k,l) \\ \mathbf{m}_{kl} & \text{otherwise} \end{cases}$$

For other tests, we state, as usual $\mathbb{C}^\sharp[\![\, c \,]\!]\mathbf{m} = \mathbf{m}$.

Assignments. We can exactly model assignments of the form $V_j \leftarrow c$ and $V_j \leftarrow V_i + c$. For other assignments, we revert to a non-deterministic assignment $V_j \leftarrow [-\infty, +\infty]$, which can also be handled exactly.

Let us first consider $V_j \leftarrow [-\infty, +\infty]$. A natural idea is to remove all the constraints involving V_j, i.e., we put all the elements at line j and all the elements at column j to $+\infty$. Similarly to the join, however, this operation is only guaranteed to be exact if it is applied on a matrix in normal form, hence, we define:

$$\forall k, l\colon \left[\mathbb{S}^\sharp[\![\, V_j \leftarrow [-\infty, +\infty] \,]\!]\mathbf{m}\right]_{kl} \stackrel{\text{def}}{=} \begin{cases} +\infty & \text{if } k = j \text{ or } l = j \\ m^*_{kl} & \text{otherwise} \end{cases}$$

On a non-normalized matrix, we may lose precision as we discard implicit constraints between two variables that are distinct from V_j, and yet related to it through constraints. For instance, removing V_1 in $(V_1 - V_0 \leq 1) \wedge (V_2 - V_1 \leq 1)$ would give $\top^\sharp$ while, when using the normalization, we get $V_2 - V_0 \leq 2$ instead, which is more precise. Indeed, the normalization makes explicit in the matrix all possible combinations of constraints using V_1 before it is discarded.

Assignments of the form $V_j \leftarrow c$ and $V_j \leftarrow V_i + c$ for $i \neq j$ are non-invertible. We can model them, as in the affine domains, as: $\mathbb{S}^\sharp[\![\, V_j \leftarrow e \,]\!] \stackrel{\text{def}}{=} \mathbb{C}^\sharp[\![\, V_j = e \,]\!] \circ \mathbb{S}^\sharp[\![\, V_j \leftarrow [-\infty, +\infty] \,]\!]$. As both abstractions are exact, their composition is also an exact abstraction (Thm. 2.6). An assignment of the form $V_j \leftarrow V_j + c$ is invertible. It is handled exactly by adding c to all the elements in the $j-$th column, and subtracting c to the $j-$th line:

$$\forall k, l\colon \left[\mathbb{S}^\sharp[\![\, V_j \leftarrow V_j + c \,]\!]\mathbf{m}\right]_{kl} \stackrel{\text{def}}{=} \begin{cases} m_{kl} - c & \text{if } k = j \text{ and } l \neq j \\ m_{kl} + c & \text{if } l = j \text{ and } \neq j \\ m_{kl} & \text{otherwise} \end{cases}$$

Conversion with intervals. As explained in Sect. 5.3.7, it can be useful to convert between one abstract domain and another, for instance to switch abstract domains dynamically during the analysis, or benefit from abstract operators available in the interval domain. We can convert easily an interval element into a zone, as zones can encode directly variable bounds; the conversion is exact. Given a zone described as a DBM $\mathbf{m}$, it is also easy to extract an interval abstract element that over-approximates it: the bounds of each variables can be read at line 0 and at column 0 of the matrix. Moreover, if $\mathbf{m}$ is normalized,

as a consequence of Thm. 5.1.2, the bounds we extract are the tightest possible, i.e., our conversion operator gives the best abstraction.

Conversion with polyhedra. Likewise, we can convert between zones and polyhedra. Given a DBM $\mathbf{m}$ representing a zone, we can easily obtain an exact representation as a polyhedron by interpreting each matrix element m_{ij} as a constraint $V_j - V_i \leq m_{ij}$. To convert from a polyhedron to a zone in a sound way, we use the same technique as in the conversion from polyhedra to intervals from Sect. 5.3.7. Starting from the generator representation $[\mathbf{P}, \mathbf{R}]$:

- First, we compute, in the zone domain, the join $\sqcup^{\sharp}_{\vec{P} \in \mathbf{P}} \vec{P}$ of every vertex $\vec{P}$ in the polyhedron, each one being viewed as zone representing a single point.
- Then, for every ray $\vec{R} \in \mathbf{R}$, and for every pair of coefficients R_i, R_j of $\vec{R}$ such that $i \neq j$ and $R_j - R_i > 0$, we set m_{ij} to $+\infty$. Indeed, such a ray allows increasing the value of $V_j - V_i$ arbitrarily, so that its bound no longer applies. The case $R_j - R_i < 0$ is equivalent to $R_i - R_j > 0$, and thus handled by this case as well. Moreover, the cases $i = 0$ and $j = 0$ correspond to removing, respectively, the upper bound of V_j or the lower bound of V_i when, respectively, $R_j > 0$ or $R_i < 0$.

Alternatively, linear programming can be employed to find an upper bound for every expression $V_j - V_i$, i.e., a value for m_{ij}.

Applications include switching dynamically between the polyhedra and the zone domains, but also improving the precision of assignments and tests that cannot be handled precisely by the zone domain. Consider, for instance, an affine assignment $V \leftarrow e$. We can define:

$$\mathbb{S}^{\sharp}[\![V \leftarrow e]\!]_{Zone} \stackrel{\text{def}}{=} Zone \circ \mathbb{S}^{\sharp}[\![V \leftarrow e]\!]_{Poly} \circ Poly$$

i.e., convert to a polyhedron (*Poly*), apply the polyhedron assignment ($\mathbb{S}^{\sharp}[\![V \leftarrow e]\!]_{Poly}$), and convert back to a zone (*Zone*). The two first operations are exact abstractions, and the third one is a best abstraction. Hence, the combination gives back a best abstraction for affine assignments in zones. The same holds for affine tests.

Note, however, that a conversion from a zone to a polyhedron and back requires either at least one application of Chernikova's algorithm, as we generate a constraint representation but expect back a generator representation, or many linear programming problems to extract the difference bounds from the constraint representation. Both are very costly.

Approximate affine operators. We now propose a more practical abstract assignment operator for arbitrary affine expressions on zones: it trades precision for efficiency as it is no longer a best abstraction, but it has a quadratic cost instead of the possibly exponential cost of the assignment based on polyhedra.

Consider an affine expression $e \stackrel{\text{def}}{=} \sum_i \alpha_i V_i + \beta$, and the assignment $V_j \leftarrow e$. The idea is to exploit the abstract evaluation of expressions in the interval domain, which is the generic way to handle assignments in a non-relational domain (Fig. 4.1) but, unlike an

interval assignment, we will not only infer the bounds of the assigned variable V_j, but also of expressions $V_j - V_i$ for all variables $V_i \in \mathbb{V}$. More precisely, we define:

$$\forall k, l\colon \left[\mathbb{S}^\sharp[\![V_j \leftarrow e]\!]_{Zone}\mathbf{m}\right]_{kl} \stackrel{\text{def}}{=} \begin{cases} \max(\mathbb{E}^\sharp[\![e]\!]_{Int} Int(\mathbf{m})) & \text{if } k = 0 \text{ and } l = j \\ -\min(\mathbb{E}^\sharp[\![e]\!]_{Int} Int(\mathbf{m})) & \text{if } k = j \text{ and } l = 0 \\ \max(\mathbb{E}^\sharp[\![e - V_k]\!]_{Int} Int(\mathbf{m})) & \text{if } k \neq 0, j \text{ and } l = j \\ -\min(\mathbb{E}^\sharp[\![e + V_l]\!]_{Int} Int(\mathbf{m})) & \text{if } k = j \text{ and } l \neq 0, j \\ m_{ij} & \text{otherwise} \end{cases} \tag{5.15}$$

where $\mathbb{E}^\sharp[\![e]\!]_{Int}$ is the evaluation of e in the interval domain (Figs. 4.1, 4.6) and $Int(\mathbf{m})$ is the conversion from zones to intervals described above. Moreover, we assume that an expression such as $e - V_k$ is simplified symbolically before being fed to $\mathbb{E}^\sharp[\![e]\!]_{Int}$, i.e., $e - V_k$ is $\sum_{i \neq k} \alpha_i V_i + (\alpha_k - 1)V_k + \beta$. For instance, given the assignment $X \leftarrow Y + Z$, then our operator will use the bound of Z to derive a bound in $X - Y$. Although our operator cannot exploit relational information, as expressions are evaluated in the interval domain, it is nevertheless able to infer new relations, which is more than what the interval domain can do.

Example 5.17 (Approximate zone assignments)**.** *Figure 5.8 presents three different abstractions of the assignment $X \leftarrow Y - Z$ on the zone $(Y \in [0, 10]) \wedge (Z \in [0, 10]) \wedge (Y - Z \in [0, 10])$ (Fig. 5.8.(a)). In the results, boldface numbers correspond to non-optimal bounds:*

1. *Fig. 5.8.(b) presents the result obtained by only evaluating the bounds of X in the interval domain. Bounds on $X - Y$ and $X - Z$ are then retrieved by normalization, combining the novel bounds for X with the existing bounds for Y and Z. This is the least precise operator, as four bounds are non-optimal.*

2. *Fig. 5.8.(c) presents the result using the operator we defined in (5.15). The result is much more precise as we infer optimal bounds for $X - Y$ using the bound of Z, which are then combined by the normalization with the bounds of $Y - Z$ to get optimal bounds for $X - Z$. Only one bound is not optimal: the lower bound of X.*

3. *Fig. 5.8.(d) presents the result using the polyhedra domain. It is the most precise as it is guaranteed to be optimal, but also the most costly.* ♦

A similar operator can be designed for affine tests: given a test such as $e \leq 0$, we will derive bounds for each $V_j - V_i$ by evaluating, in the interval domain, $V_j - V_i + e$ after symbolic simplification.

5.4.4 Convergence Acceleration

As for intervals and polyhedra, the zone domain has both infinite increasing chains and infinite decreasing chains.

(a)

$$\begin{cases} 0 \leq Y \leq 10 \\ 0 \leq Z \leq 10 \\ 0 \leq Y - Z \leq 10 \end{cases}$$

$$\Downarrow \; X \leftarrow Y - Z$$

$$\begin{cases} \mathbf{-10} \leq X \leq 10 \\ \mathbf{-20} \leq X - Y \leq \mathbf{10} \\ \mathbf{-20} \leq X - Z \leq 10 \\ 0 \leq Y \leq 10 \\ 0 \leq Z \leq 10 \\ 0 \leq Y - Z \leq 10 \end{cases} \qquad \begin{cases} \mathbf{-10} \leq X \leq 10 \\ -10 \leq X - Y \leq 0 \\ -10 \leq X - Z \leq 10 \\ 0 \leq Y \leq 10 \\ 0 \leq Z \leq 10 \\ 0 \leq Y - Z \leq 10 \end{cases} \qquad \begin{cases} 0 \leq X \leq 10 \\ -10 \leq X - Y \leq 0 \\ -10 \leq X - Z \leq 10 \\ 0 \leq Y \leq 10 \\ 0 \leq Z \leq 10 \\ 0 \leq Y - Z \leq 10 \end{cases}$$

(b) (c) (d)

Figure 5.8: Three operators to model the assignment $X \leftarrow Y - Z$ on the zone (a): (b), non-relational operator based on the interval domain; (c), approximate operator from (5.15); and (d), optimal operator based on the polyhedra domain. Boldface numbers correspond to non-optimal bounds.

Widening. Similarly to the interval domain, the zone domain manipulates bounds, hence, we can design similar widenings. The standard widening $\triangledown$ simply puts unstable bounds to infinity:

$$\forall i, j\colon [\mathbf{m} \triangledown \mathbf{n}]_{ij} \stackrel{\text{def}}{=} \begin{cases} m_{ij} & \text{if } n_{ij} \leq m_{ij} \\ +\infty & \text{otherwise} \end{cases} \tag{5.16}$$

We can also define a widening with thresholds, $\triangledown_T$, similarly to the interval one (Sect. 4.7.3). Given a finite set of thresholds T containing $+\infty$, unstable bounds are set to the next value in T, until they become stable or $+\infty$ is reached:

$$\forall i, j\colon [\mathbf{m} \triangledown_T \mathbf{n}]_{ij} \stackrel{\text{def}}{=} \begin{cases} m_{ij} & \text{if } n_{ij} \leq m_{ij} \\ \min \{ x \in T \mid x \geq n_{ij} \} & \text{if } n_{ij} > m_{ij} \end{cases} \tag{5.17}$$

Interaction between normalization and widening. Recall that the zone union was defined as $\mathbf{m}^* \sqcup^\sharp \mathbf{n}^*$, i.e., the arguments are put in normal form before applying the lattice join $\sqcup^\sharp$, and we saw that this is important to guarantee the best precision possible. As a widening can be seen as a kind of join, relaxed enough to enforce termination, it is tempting to apply the same reasoning, and try and improve the analysis precision by replacing $\mathbf{m} \triangledown \mathbf{n}$ with $\mathbf{m}^* \triangledown \mathbf{n}^*$. The following example shows that this can lead to non-termination: while a sequence of the form $X^{\sharp\, n+1} \stackrel{\text{def}}{=} X^{\sharp\, n} \triangledown Y^{\sharp\, n}$ always converges, a sequence $X^{\sharp\, n+1} \stackrel{\text{def}}{=} (X^{\sharp\, n})^* \triangledown Y^{\sharp\, n}$ or $X^{\sharp\, n+1} \stackrel{\text{def}}{=} (X^{\sharp\, n} \triangledown Y^{\sharp\, n})^*$ may not. Intuitively, the convergence is ensured by putting finite elements to $+\infty$, at which point they stay stable at $+\infty$, while the normalization tightens the bounds, in particular replacing $+\infty$ with finite bounds, threatening termination.

Example 5.18. *Consider the following, rather intricate program:*

$X \leftarrow 0; Y \leftarrow [-1, 1];$
while $[0, 1] = 0$ **do**
  $R \leftarrow [-1, 1];$
  if $X = Y$ **then** $Y \leftarrow X + R$
  else $X \leftarrow Y + R$ **endif**
done

It is comprised of an unbounded loop that either adds a value R in $[-1, 1]$ to X to get a new Y value, or to Y to get a new X value, at each iteration. An analysis using the widening without normalization, i.e., as $X^{\sharp n+1} \stackrel{\text{def}}{=} X^{\sharp n} \triangledown F^{\sharp}(X^{\sharp n})$, stabilizes the loop invariant in three iterations:

iter.	X	Y	$X - Y$
0	0	$[-1, 1]$	$[-1, 1]$
1	$[-\infty, +\infty]$	$[-1, 1]$	$[-1, 1]$
2	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-1, 1]$

and outputs that $X - Y \in [-1, 1]$ without any bound on X nor Y, which is actually the most precise invariant.

However, if we compute $X^{\sharp n+1} \stackrel{\text{def}}{=} (X^{\sharp n} \triangledown F^{\sharp}(X^{\sharp n}))^*$, we get an infinite iteration sequence:

iter.	X	Y	$X - Y$
0	0	$[-1, 1]$	$[-1, 1]$
1	$[-2, 2]$	$[-1, 1]$	$[-1, 1]$
2	$[-2, 2]$	$[-3, 3]$	$[-1, 1]$
...	...	...	...
$2j$	$[-2j, 2j]$	$[-2j - 1, 2j + 1]$	$[-1, 1]$
$2j + 1$	$[-2j - 2, 2j + 2]$	$[-2j - 1, 2j + 1]$	$[-1, 1]$
...	...	...	...

Indeed, at each iteration, either X or Y is set to $[-\infty, +\infty]$ by the widening, but then, the normalization combines the variable which was kept finite at this iteration with the invariant $X - Y \in [-1, 1]$ to deduce a finite bound for the variable that was set to $[-\infty, +\infty]$. ♦

A final remark about widening, related to our inability to use normal forms when iterating, is that the output of the widening $\mathbf{m} \triangledown \mathbf{n}$ does not only depend on the zones $\gamma(\mathbf{m})$ and $\gamma(\mathbf{n})$ represented by the arguments, but also on the DBM chosen to represent them. As for the naive polyhedra widening (Sect. 5.3.4), we have a representation-aware widening. A more semantic widening, that is independent from the DBM chosen to represent the zone arguments, has been proposed by Bagnara et al. [2009], based on an alternate normal form that tries to remove as many constraints as possible (i.e., put bounds to $+\infty$).

Narrowing. In order to stabilize decreasing sequences in finite time, we can design a narrowing operator $\triangle$ on zones. We take some inspiration form the interval narrowing

(4.10): we only refine an upper bound if it is $+\infty$. This guarantees that each bound is refined a most once, and so, there are finitely many refinements:

$$\forall i, j\colon [\mathbf{m} \vartriangle \mathbf{n}]_{ij} \stackrel{\text{def}}{=} \begin{cases} n_{ij} & \text{if } m_{ij} = +\infty \\ m_{ij} & \text{otherwise} \end{cases} \tag{5.18}$$

5.4.5 The Octagon Abstract Domain

The octagon domain, introduced by Miné [2006a], is a slight extension of the zone domain. More precisely, instead of expressing only constraints of the form $V_j - V_i \leq c$, the octagon domain can express constraints with two variables and unit coefficients, i.e., each coefficient can be independently $+1$ or -1. Octagonal constraints have the form: $\pm V_i \pm V_j \leq c$.

The octagon domain uses the same kinds of data-structures and algorithms as zones, i.e., potential graphs, difference bound matrices, and shortest path closure. It has also the same cubic time and quadratic space complexity. As it is more expressive and more symmetric, it is much more used in practice than zones for program analysis; however, as it is slightly more complex, we chose to present the zone domain in details, and only discuss here briefly the additional technical details that are specific to octagons. We will also illustrate our domains with a static analysis example in the octagon domain.

The term *octagon* comes from the fact that, in two dimensions, abstract elements are octagon-shaped, having at most eight corners, although this is no longer true in higher dimensions. The same shapes are called "simple sections" by Balasundaram and Kennedy [1989].

Representation. A set of octagonal constraints $\pm V_j \pm V_i \leq c$ is encoded as a difference bound matrix or, equivalently, a potential graph. We use a variable change to map octagonal constraints to potential constraints. More precisely, for each variable $V_i \in \mathbb{V}$, we consider two versions, V'_{2i-1} and V'_{2i}, which correspond respectively to $+V_i$ and $-V_i$, i.e., V_i may appear in a positive or a negative form. Hence:

$$\begin{array}{rlrlrr}
V_i - V_j \leq c & \text{is encoded as} & V'_{2i-1} - V'_{2j-1} \leq c & \text{and} & V'_{2j} - V'_{2i} \leq c \\
V_i + V_j \leq c & \text{is encoded as} & V'_{2i-1} - V'_{2j} \leq c & \text{and} & V'_{2j-1} - V'_{2i} \leq c \\
-V_i - V_j \leq c & \text{is encoded as} & V'_{2i} - V'_{2j-1} \leq c & \text{and} & V'_{2j} - V'_{2i-1} \leq c \\
V_i \leq c & \text{is encoded as} & V'_{2i-1} - V'_{2i} \leq 2c & & \\
-V_i \leq -c & \text{is encoded as} & V'_{2i} - V'_{2i-1} \leq -2c & &
\end{array}$$

We note that some constraints can be encoded in two ways, and we will assume a *coherence property*: whenever a constraint appears in one form with some bound c, the equivalent one appears with the same bound c. We also note that unary constraints, such as $V_i \leq c$, can be encoded directly as $V_i + V_i \leq 2c$, so that we do not need to add a variable V_0 representing the constant zero, as we did for zones. Thus, an octagon can be represented as a DBM with size $2n$. As usual, we add the $\bot^\sharp$ element to obtain our abstract domain, hence:

$$\mathcal{D}^\sharp \stackrel{\text{def}}{=} (\mathbb{I} \cup \{+\infty\})^{(2n)\times(2n)} \cup \{\bot^\sharp\}$$

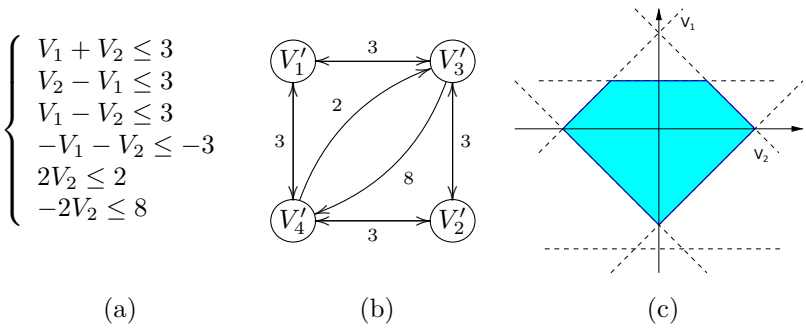


Figure 5.9: A set (a) of octagonal constraints; (b), its representation as a potential graph; and (c) the octagon it represents.

A DBM $\mathbf{m}$ represents the following set $\gamma(\mathbf{m})$:

$$\begin{array}{l} \gamma(\mathbf{m}) \stackrel{\text{def}}{=} \{\, (v_1, \ldots, v_n) \in \mathbb{P} \mid (v_1, -v_1, \ldots, v_n, -v_n) \in \gamma_{DBM}(\mathbf{m}) \,\} \\ \text{where:} \\ \gamma_{DBM}(\mathbf{m}) \stackrel{\text{def}}{=} \{\, (v_1, \ldots, v_{2n}) \in \mathbb{I}^{2n} \mid \forall i, j \in [1, 2n] \colon v_j - v_i \leq m_{ij} \,\} \end{array} \tag{5.19}$$

where we reused the concretization of potential constraints (5.10), here denoted as γ_{DBM} to avoid any ambiguity.

Due do our numbering of variables, variable V_i corresponds to the lines and columns numbered $2i - 1$, for $+V_i$, and $2i$, for $-V_i$. Given $i \in [1, 2n]$, we denote as $\bar{\imath} \stackrel{\text{def}}{=} i - 1$ if i is even, and $\bar{\imath} \stackrel{\text{def}}{=} i + 1$ if i is odd, i.e., if i corresponds to $+V_j$, then $\bar{\imath}$ corresponds to $-V_j$, and if i corresponds to $-V_j$, then $\bar{\imath}$ corresponds to $+V_j$. Hence, for instance, the coherence property, stating that the two potential constraint representations of an octagon constraint are equivalent, is written simply: $\forall i, j \colon \mathbf{m}_{ij} = \mathbf{m}_{\bar{\jmath}\bar{\imath}}$.

Figure 5.9.(a) gives an example of octagonal constraints over $\{V_1, V_2\}$. Then, Fig. 5.9.(b) gives the encoding as a potential graph, using the set of variables $\{V_1', V_2', V_3', V_4'\}$ representing intuitively $\{V_1, -V_1, V_2, -V_2\}$ (we could have used an equivalent DBM representation as well), and Fig. 5.9.(c) gives the corresponding set of points.

Order structure. The very same order structure we used on DBM representing zones can be used on DBM representing octagons. We have a lattice $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot^\sharp, \top^\sharp)$ where $\sqsubseteq^\sharp$, $\sqcup^\sharp$, and $\sqcap^\sharp$ are defined element-wise, and $\top^\sharp$ has all its elements set to $+\infty$. As for zones, if $\mathbb{I} \in \{\mathbb{Z}, \mathbb{R}\}$, then the lattice is complete, and we have a Galois connection $(\mathcal{P}(\mathbb{P}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$ with $\alpha(\emptyset) \stackrel{\text{def}}{=} \bot^\sharp$ and, when $S \neq \emptyset$:

$$[\alpha(S)]_{ij} \stackrel{\text{def}}{=} \max \{\, v_j' - v_i' \mid \exists (v_1, \ldots, v_n) \in S \colon \forall k \colon v_{2k-1}' = v_k = -v_{2k}' \,\}$$

Strong normalization. The DBM normal form $\mathbf{m}^*$ is not a normal form with respect to the octagonal concretization γ (5.19): we can have two different matrices that are closed by shortest path and yet represent the same octagon. Consider, for instance: $(V_1' - V_2' \leq$

$2) \wedge (V_3' - V_4' \leq 2)$, which represents $(2V_1 \leq 2) \wedge (2V_2 \leq 2)$, and $(V_1' - V_2' \leq 2) \wedge (V_3' - V_4' \leq 2) \wedge (V_1' - V_4' \leq 2) \wedge (V_3' - V_2' \leq 2)$, which represents $(2V_1 \leq 2) \wedge (2V_2 \leq 2) \wedge (V_1 + V_2 \leq 2)$. Both systems are in normal form, and yet, both represent the same set. Intuitively, the problem is that the shortest path closure only saturates our constraint system by applying deductions of the form $(V_1' - V_2' \leq a) \wedge (V_2' - V_3' \leq b) \implies (V_1' - V_3' \leq a + b)$. But, because the relation $\forall i\colon V_i' = -V_{\bar{i}}'$ implicitly holds due to our representation encoding, other forms of valid deductions are not applied by the shortest path closure. More precisely, from $V_i' - V_{\bar{i}}' \leq a$ and $V_{\bar{j}}' - V_j' \leq b$, we should be able to deduce that $V_i' - V_j' \leq (a + b)/2$. Indeed, $V_i' - V_{\bar{i}}' \leq a$ and $V_{\bar{j}}' - V_j' \leq b$ both represent unary constraints over $\mathbb{V}$, and two unary constraints can be added to derive an octagonal constraint.

Bagnara et al. [2008a] prove that, in order to saturate an octagon with respect to both kinds of constraints, it is sufficient to first saturate with respect to shortest path closure, and then saturate with respect to adding unary constraints. We thus replace the normalization $\mathbf{m}^*$ with the so-called *strong normalization*, denoted as $\mathbf{m}^\bullet$ and defined as:

$$\forall i, j\colon [\mathbf{m}^\bullet]_{ij} \stackrel{\text{def}}{=} \min(m^*_{ij}, (m^*_{i\bar{i}} + m^*_{\bar{j}j})/2) \tag{5.20}$$

where $\mathbf{m}^*$ was defined in (5.12) and can be computed by Floyd–Warshall's algorithm (5.14). Note the division by 2: indeed, a unary constraint is encoded as $V_i + V_i \leq a$ or $-V_i - V_i \leq b$; adding two such constraints gives a constraint of the form $\pm 2V_i \pm 2V_j \leq a + b$, which must be divided by 2 to obtain an octagonal constraint. Similarly to the closure, the strong closure can be computed in cubic time.

Abstract operators. The octagon operators are extremely similar to the zone ones; we simply need to employ the strong normalization instead of the classic one. For instance:

$$\begin{aligned}
&\gamma(\mathbf{m}) = \gamma(\mathbf{n}) \iff \mathbf{m}^\bullet = \mathbf{n}^\bullet \\
&\gamma(\mathbf{m}) \subseteq \gamma(\mathbf{n}) \iff \mathbf{m}^\bullet \sqsubseteq^\sharp \mathbf{n} \\
&\mathbf{m} \cup^\sharp \mathbf{n} \stackrel{\text{def}}{=} \mathbf{m}^\bullet \sqcup^\sharp \mathbf{n}^\bullet \\
&\mathbf{m} \cap^\sharp \mathbf{n} \stackrel{\text{def}}{=} \mathbf{m} \sqcap^\sharp \mathbf{n}
\end{aligned} \tag{5.21}$$

The tests and assignments are similar. We can handle exactly the non-deterministic assignment $V_j \leftarrow [-\infty, +\infty]$, which uses the strong normalization to avoid losing implicit constraints:

$$\forall k, l\colon \left[\mathbb{S}^\sharp [\![V_j \leftarrow [-\infty, +\infty]]\!] \mathbf{m}\right]_{kl} \stackrel{\text{def}}{=} \begin{cases} +\infty & \text{if } k \in \{2j-1, 2j\} \\ & \text{or } l \in \{2j-1, 2j\} \\ m^\bullet_{kl} & \text{otherwise} \end{cases}$$

Exact tests and assignments can be extended to tests of the form $\pm V_i \pm V_j \leq c$ and assignments of the form $V_j \leftarrow \pm V_i + c$. We can also extend the approximate affine assignment $V_j \leftarrow e$ of (5.15) to infer bounds on expressions of the form $\pm V_j \pm V_i$ for every i. Finally, the widening $\triangledown$ (5.16), including more advanced widenings such as widening with thresholds $\triangledown_T$ (5.17), and narrowing $\triangle$ (5.18), remain exactly the same: they operate point-wise on every DBM element.

Integer octagons. While the zone domain worked equally well and in a similar fashion for integers, reals, and rationals, this is not the case for octagons. In particular, we glossed over the division by 2 in (5.20), required when computing $(m^*_{i\bar{i}} + m^*_{\bar{j}j})/2$. While it is sound to replace it with the stricter integer bound $\lfloor (m^*_{i\bar{i}} + m^*_{\bar{j}j})/2 \rfloor$, the result does not provide a normal form. Ensuring the saturation of constraint propagation requires extra propagation steps. We refer to Bagnara et al. [2008a] for a description of a normalization algorithm for integer octagons that still runs in cubic time.

5.4.6 Octagon Analysis Example

We illustrate the use of weakly relational domains with the octagon domain, as it is more used in practice than the zone domain. Consider the following program:

```
Y ← 0;
while 1 = 1 do
    X ← [−128, 128];
    D ← [0, 16];
    S ← Y;
    Y ← X;
    R ← X − S;
    if R ≤ −D then Y ← S − D endif;
    if R ≥ D then Y ← S + D endif
done
```

This program is an infinite loop modeling a reactive program, which transforms a stream of inputs into a stream of outputs, with the help of some internal state. At each loop iteration, a new input is fetched in $[-128, 128]$ into X and a new output is stored into Y. The goal is to have Y follow X as closely as possible, with the extra requirement that two successive values of Y should not differ from more than D in absolute value. To achieve this, a state variable S is used to remember the value of Y from the previous iteration. We first set Y to X and, if $X - S$, stored into R, overflows D, we clamp Y to either $X + D$ or $X - D$. The maximal slope D is itself fetched anew at each iteration from an input in $[0, 16]$. This program is inspired from an actual embedded control-command code.

The goal of our static analysis is to find a tight bound on Y. In fact, Y actually ranges in $[-128, 128]$, the same range as X. Because of the assignments $Y \leftarrow S - D$ and $Y \leftarrow S + D$, this is far from obvious. An analysis must be able to exploit the fact that $R = X - S$, as well as the guards $R \leq -D$ and $R \geq D$ protecting these assignments, to deduce the correct bound. We compare the analysis performed in the interval domain, the octagon domain, and the polyhedron domain.

Interval analysis. An interval analysis without widening will compute the following sequence of intervals for Y at the loop head trying to infer a loop invariant: $[0, 0]$, $[-144, 144]$, $[-160, 160]$, etc. More generally, at iteration n, we get $Y \in [-128-16n, 128+16n]$, and the sequence does not converge naturally. Indeed, the tests $R \leq -D$ and $R \geq D$ guarding the assignments $Y \leftarrow S - D$ and $Y \leftarrow S + D$ do not bring any information using the interval

domain, hence, at each iteration, Y is assumed to be increased by $[-16, 16]$, starting at $[-128, 128]$.

When accelerating the convergence using a widening, we get, after a finite number of iterations, $Y \in [-\infty, +\infty]$, which is very coarse. Note that, as no interval for Y is stable in an interval analysis, it is no use employing different widenings, narrowings, or advanced iteration strategies (4.7): we will always find $Y \in [-\infty, +\infty]$.

Octagon analysis. The octagon domain fares better than the interval one as it is relational. Note, however, that neither assignments $R \leftarrow X - S$, $Y \leftarrow S - D$, nor $Y \leftarrow S + D$ can be exactly modeled. We assume that we are employing the octagon domain version of the approximate assignment operator for affine expressions we defined for zones in (5.15). Moreover, we assume that we employ a widening with thresholds ∇_T (5.17). Then, the bounds found for Y are $[-c, c]$, where $c = \min \{ x \in T \mid x \geq 144 \}$, i.e., we find the tightest bound greater than 144. Note that we cannot find 128 as bound, but only $144 = 128 + 16$ due to the loss of precision caused by constraints, such as $R = X - S$, that cannot be exactly represented. Moreover, we rely on thresholds to skip to a stable value, above 144, instead of going to $+\infty$ directly.

Polyhedra analysis. A polyhedra analysis would find that $Y \in [-128, 128]$ by employing a classic polyhedra widening in very few iterations. No thresholds are needed, and we find directly the tightest bounds. All the tests and assignments used in the analysis feature exact abstractions. While the polyhedra analysis is more precise and simpler to setup (no need for thresholds), it is however more costly in time and memory. True to their words, weakly relational domains, such as the octagon domain, offer a trade-off between cost and precision, and are sufficient in some contexts where relational invariants are mandatory.

5.5 The Template Domain

The last relational domain we wish to present is another weakly relational domain, in-between in terms of cost and precision between intervals and polyhedra. More precisely, the *template domain*, introduced by Sankaranarayanan et al. [2005], infers conjunctions of affine inequalities $\mathbf{M} \times \vec{V} \leq \vec{C}$ but, unlike the polyhedra domain, only the right-hand side $\vec{C}$, i.e., the upper bounds are inferred. The left-hand side $\mathbf{M}$, i.e., the coefficients of the variables in the constraints, are fixed before the analysis is run, and not inferred during the analysis. Concerning the expressive power, we can see the zone and the octagon domains (and even the interval domain) as special cases of the template domain, for specific choices of $\mathbf{M}$. However, unlike those domains, the shape of the left-hand side $\mathbf{M}$ is not fixed by the domain, but can be configured freely by the user before the analysis.

This domain has two unique features. Firstly, its expressiveness is fully parameterized, so that a user can decide on a cost versus precision trade-off within the domain, and also adapt the domain to the requirements of a specific program analysis. Secondly, its algorithmic core is based on linear programming, which is a change from polyhedra based

on the double description method, and from zones and octagons based on shortest path closure.

As we use general linear algebra, we assume, as we did for the affine equalities and inequalities domains, that $\mathbb{I} \in \{\mathbb{Q}, \mathbb{R}\}$.

Representation. We assume that a matrix $\mathbf{M} \in \mathbb{I}^{m \times n}$ is fixed. $n = |\mathbb{V}|$ is the number of variables, while m is arbitrary. Intuitively, the set and number of linear expressions on the left-hand of constraints can be freely chosen. We call $\mathbf{M}$ the *template*.

An abstract element $X^\sharp \in \mathcal{D}^\sharp$ is given by setting the upper bound of each linear expression, which we store as a $m-$dimensional vector $\vec{C}$. Note, however, that we need a way to state that a linear expression in $\mathbf{M}$ is unbounded — which was not necessary for polyhedra — hence, we allow the upper bound to be $+\infty$, and our vectors $\vec{C}$ live in $(\mathbb{I} \cup \{+\infty\})^m$. As usual, we also add a $\bot^\sharp$ element, which is a canonical representation of the empty set, thus:

$$\mathcal{D}^\sharp \stackrel{\text{def}}{=} \{\bot^\sharp\} \cup (\mathbb{I} \cup \{+\infty\})^m$$

and the concretization is naturally:

$$\gamma(\vec{C}) \stackrel{\text{def}}{=} \{\, \vec{V} \in \mathbb{P} \mid \mathbf{M} \times \vec{V} \leq \vec{C} \,\}$$

As stated above, the template domain generalizes the interval, zone, and octagon domains, which become special cases for a fixed template $\mathbf{M}$. More precisely:

- for intervals, $m = 2n$: there is an affine expression V_i and an affine expression $-V_i$ for every variable $V_i \in \mathbb{V}$;
- for zones, $m = n^2 + n$: there is an affine expression $V_j - V_i$ for every $i \neq j$, in addition to the affine expressions representing intervals.

However, from an algorithmic point of view, the domains are implemented quite differently, and are much less efficient than the native interval and zone domains we presented in previous sections. The template domain is useful when $\mathbf{M}$ remains small but has a complex structure, featuring more varied expressions, out of the scope of zones.

Order structure. As for zones, we extend the natural total order on bounds $(\mathbb{I} \cup \{+\infty\}, \leq)$ to vectors, pointwise, to get our partial order $\sqsubseteq^\sharp$ on $\mathcal{D}^\sharp$. We actually get a lattice structure $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot^\sharp, \top^\sharp)$, where $\sqcup^\sharp$ is the point-wise maximum, $\sqcap^\sharp$ is the point-wise minimum, and $\top^\sharp$ maps all affine expressions to $+\infty$. The lattice is complete if $\mathbb{I} = \mathbb{R}$, and we can define an abstraction function. It associates to each affine expression $\vec{M}_i$ at line i in $\mathbf{M}$ the tightest upper bound:

$$\forall i \leq m\colon [\alpha(S)]_i = \max \{\, \vec{M}_i \cdot \vec{V} \mid \vec{V} \in S \,\}$$

when $S \neq \emptyset$, and $\alpha(S) = \bot^\sharp$ otherwise.

Normalization. The concretization γ is not one-to-one: there exist different abstract elements that represent the same polyhedron. Similarly to the zone domain, we define a normal form $\vec{C}^*$ that tries to tighten the constraints as much as possible, until they saturate (i.e., touch) the polyhedron $\gamma(\vec{C})$. We have indeed $\vec{C}^* = \alpha(\gamma(\vec{C}))$.

The normal form can be effectively computed using linear programming LP:

$$\begin{array}{l} \forall i \leq m\colon \left[\vec{C}^*\right]_i \stackrel{\text{def}}{=} LP(\langle\, \mathbf{M},\, \vec{C}\,\rangle, M_i) \\ \text{where:} \\ LP(\langle\, \mathbf{M},\, \vec{C}\,\rangle, \vec{V}) \stackrel{\text{def}}{=} \max\{\, \vec{P}\cdot\vec{V} \mid \mathbf{M}\times\vec{P} \leq \vec{C}\,\} \end{array} \tag{5.22}$$

We have adapted slightly the definition of LP from (5.8) because we manipulate upper bounds instead of lower bounds. Additionally, standard LP algorithms are able to determine whether the set of affine constraints $\mathbf{M}\times\vec{V} \leq \vec{C}$ is satisfiable and, if it is not, we return $\bot^\sharp$.

Abstract operators. Using the normal form, we can decide equality and inclusion exactly:

$$\begin{array}{rcl} \gamma(X^\sharp) = \gamma(Y^\sharp) & \iff & X^{\sharp *} = Y^{\sharp *} \\ \gamma(X^\sharp) \subseteq \gamma(Y^\sharp) & \iff & X^{\sharp *} \sqsubseteq^\sharp Y^\sharp \end{array}$$

which is similar to the case of zones (Sect. 5.4.3).

Also similarly to zones, we can model the union and the intersection by taking, respectively, for each affine expression, the loosest or the strictest of the constraints from both arguments. The intersection $\cap^\sharp \stackrel{\text{def}}{=} \sqcap^\sharp$ is exact. The union $\cup^\sharp$ is not exact and, for it to be optimal, we must use the normal form, i.e., we state $X^\sharp \cup^\sharp Y^\sharp \stackrel{\text{def}}{=} X^{\sharp *} \sqcup^\sharp Y^{\sharp *}$. The result is naturally in normal form.

Modeling a test $\mathbb{C}^\sharp[\![\, e \leq c \,]\!]\vec{C}$ is very easy if e is an affine expression that happens to be precisely in the template $\mathbf{M}$. In that case, we simply replace the corresponding coefficient C_i in $\vec{C}$ with $\min(C_i, c)$. Other affine tests can be modeled by applying linear programming to every linear expression M_i in the template $\mathbf{M}$ on the polyhedron enriched with the new constraint:

$$\forall i\colon \left[\mathbb{C}^\sharp[\![\, \textstyle\sum_j \alpha_j V_j \leq \beta \,]\!]\vec{C}\right]_i \stackrel{\text{def}}{=} LP(\left\langle \begin{bmatrix} \mathbf{M} \\ \alpha_1 \cdots \alpha_n \end{bmatrix}, \begin{bmatrix} \vec{C} \\ \beta \end{bmatrix} \right\rangle, M_i) \tag{5.23}$$

In that case, the operator is not exact, but it is optimal. Note that this is more costly to compute, but we always have the recourse to model the test as the identity, to improve performance at the cost of precision.

The non-deterministic assignment $V_j \leftarrow [-\infty, +\infty]$, which serves also as a fallback operator for non-affine assignments can be modeled by putting to $+\infty$ all the bounds C_i corresponding to a template expression in $\mathbf{M}$ where V_j occurs, i.e, $M_{ij} \neq 0$. As in the zone domain, the operator is exact, provided that the argument is in normal form.

For affine assignments $V_j \leftarrow \sum_i \alpha_i V_i + \beta$, a simple and universal method is to exploit the corresponding operator on polyhedra. A template domain element $\vec{C}$ can be viewed as a polyhedron $\langle \mathbf{M}, \vec{C}\rangle$, on which we apply the polyhedra operator $\mathbb{S}^\sharp[\![\, V \leftarrow e \,]\!]_{Poly}$

(Sect. 5.3.3). The result is a general polyhedron that may not obey the template and may have left-hand side affine expressions that are not in $\mathbf{M}$. However, we can find the smallest template element that contains it by applying linear programming for each row $\vec{M}_i$ of the template: Hence, we state:

$$\forall i \leq m\colon \left[\mathbb{S}^\sharp[\![V \leftarrow e]\!]\vec{C}\right]_i \stackrel{\text{def}}{=} LP(\mathbb{S}^\sharp[\![V \leftarrow e]\!]_{Poly}\langle \mathbf{M}, \vec{C}\rangle, M_i)$$

This technique is similar to that employed in the normalization (5.22) and the general affine test (5.23).

Convergence acceleration. The template domain features infinite increasing and decreasing chains. Note that the domain has a similar structure as the interval and the zone domains: it is composed of a finite number of bounds of fixed affine expressions. We can thus construct, as in those domains, widenings and narrowings independently on each bound. For instance, generalizing the classic zone widenings (5.16) and narrowing (5.18), we define:

$$\forall i \leq m\colon \left[\vec{C} \triangledown \vec{D}\right]_i \stackrel{\text{def}}{=} \begin{cases} C_i & \text{if } C_i \geq D_i \\ +\infty & \text{otherwise} \end{cases}$$

$$\forall i \leq m\colon \left[\vec{C} \vartriangle \vec{D}\right]_i \stackrel{\text{def}}{=} \begin{cases} D_i & \text{if } C_i = +\infty \\ C_i & \text{otherwise} \end{cases}$$

We could naturally generalize the widening with thresholds as well (5.17). Finally, note that the result of the widening should not be put in normal form between two iterations, as it may jeopardize the convergence — this is shown in Ex. 5.18, which was designed for the zone domain but also holds for a template domain where the template $\mathbf{M}$ is that of a zone.

Template generation. The template domain assumes that a matrix $\mathbf{M}$ of template constraints is provided. It is not easy to know, before the analysis, which kinds of constraints will be useful. Obvious candidates are the affine expressions appearing syntactically in the program and in the assertions we wish to prove. Sankaranarayanan et al. [2005] note, however, that an expression appearing syntactically at some program point may only provide a relevant template at that location and not others. They suggest deriving new expressions valid at other points by applying the effect of program instructions on the template, propagating it through the program; this operation is similar to a closure by weakest precondition computation.

It is also possible to rely on a prior, possibly unsound analysis to derive a relevant template. For instance, Seladji [2017] suggests computing a limited number of concrete program runs in order to sample the memory states, and then perform a statistical analysis, called Principal Component Analysis, in order to derive the templates.

5.6 Summary

This chapter presented several relational domains. Although all these domains infer affine invariants, and can thus be seen as semantic restrictions of the most general affine domain, polyhedra, they differ significantly in the algorithms they use and the associated cost — their design exploited a variety of algorithms: Gauss elimination, Chernikova's algorithm, linear programming, Fourier-Motzkin elimination, shortest-path closure. We sum up these domains in the table below, together with the invariants they infer and their cost, roughly in the order of their expressiveness:

domain	invariants	memory cost	time cost	Sect.
affine equalities	$\sum_i \alpha_i V_i = \beta$	$\mathcal{O}(\lvert\mathbb{V}\rvert^2)$	$\mathcal{O}(\lvert\mathbb{V}\rvert^3)$	5.2
zones	$V_j - V_i \leq c$	$\mathcal{O}(\lvert\mathbb{V}\rvert^2)$	$\mathcal{O}(\lvert\mathbb{V}\rvert^3)$	5.4
octagons	$\pm V_j \pm V_i \leq c$	$\mathcal{O}(\lvert\mathbb{V}\rvert^2)$	$\mathcal{O}(\lvert\mathbb{V}\rvert^3)$	5.4.5
template	$\sum_i \alpha_i V_i \geq \beta$	$\mathcal{O}(m)$	polynomial	5.5
polyhedra	$\sum_i \alpha_i V_i \geq \beta$	exponential	exponential	5.3

The theoretical cost of the polyhedra domain is unbounded, and it relies only on the widening to limit the growth of the representation. It has nevertheless been observed to be exponential in practice [Nguyen Que, 2010]. The cost of the template domain depends on the size m of the template. We showed a polynomial time cost to account for the theoretical polynomial cost of linear programming.

We also observed that relational domains do not feature generic assignments and conditions working equally well on all expressions: precise operators are instead restricted to expressions that can be exactly expressed in the domain. In particular, for non-linear expressions, the best we can do is revert to the generic algorithms introduced for non-relational domains — more precisely, using intervals. This limitation, as well as the non-monotonicity of the widening, result in the lack of formal guarantee that a more expressive domain will result in a more precise analysis, although this is often the case in practice. Finally, we note that, except for the graph-based domains (zones and octagons), the relational domains use algorithms that do not generalize easily to integers. Yet, we can safely resort to simple solutions for integers, at the cost of certain exactness and optimality results.

Many relational and non-relational domains we have seen have the simple form of a template: a single, fixed formula, with a finite set of unknown quantities that must be determined — variable bounds $[a, b]$, bounds of affine expressions $\sum_i \alpha_i V_i \geq \beta$, or constants a and b in $a\mathbb{Z} + b$. For those, invariant inference can be viewed as a constraint satisfaction or an optimization problem: i.e., discover the values of these unknowns. However, this is not the case for the affine equalities domain nor the affine inequalities domain. The former domain has a bounded number of unknown quantities, but the shape of the affine expressions is dynamically updated by Gauss elimination during the analysis. The later domain does not have a bounded number of unknown quantities. Hence, in general, static analysis by Abstract Interpretation cannot be reduced to constraint satisfaction.

5.7 Bibliographic Notes

The original presentation of the affine equalities analysis by Karr [1976] predates Abstract Interpretation, but can nevertheless be seen as an abstract domain, without the need for a widening. It has been since generalized into congruence equalities by Granger [1991], and later improved by Müller-Olm and Seidl [2005].

The affine inequalities domain, or polyhedra domain, is introduced by Cousot and Halbwachs [1978]. It is one of the most used relational abstract domains, due to the importance of affine relations in programs. Publicly available implementations include Apron [Jeannet and Miné, 2009] and PPL [Bagnara et al., 2008b]. They are based on the double description method by Chernikova [1968] and LeVerge [1992]. Further works discuss the case of non-closed polyhedra [Bagnara et al., 2002] and develop new widening techniques [Bagnara et al., 2005a]. Nguyen Que [2010] provides an experimental evaluation of the performance of different polyhedra algorithms and implementations. The algorithmic aspects of polyhedra is still an active research subject. For instance, Singh et al. [2017] propose a polyhedra decomposition method that improves the domain scalability without loosing any precision. Additionally, recent works focus on the constraint-only representation, advocated notably by Simon and King [2005]. A recent proposal by Maréchal et al. [2017] consists in leveraging parametric linear programming methods in order to project large sets of variables and obtain directly minimized representations. While standard implementations use arbitrary-precision rationals to ensure exact computations, Chen et al. [2008] discuss the use of floating-point computations in the abstract and provide solutions to ensure the soundness despite rounding errors. Dually, Miné [2004] discusses the use of polyhedra to abstract programs using floating-point arithmetic. Finally, Simon and King [2007] discuss the use of polyhedra to abstract programs that use machine integers.

The idea of weakly relational domains is introduced with the zone domain by Miné [2001], later generalized to the octagon domain by Miné [2006a]. The efficiency of these domains has then been improved, firstly through algorithmic improvements, by Bagnara et al. [2008a] and Bagnara et al. [2009] and, secondly, through implementation improvements, such as leveraging the parallel execution available in GPU by Banterle and Giacobazzi [2007]. Alternate weakly relational domains, also based on transitive closure algorithms, include the less expressive pentagon domain by Logozzo and Fähndrich [2010] and the more expressive "Two variables per inequality" domain by Simon et al. [2002]. Other restrictions of polyhedra that rely on different algorithmic principles have been proposed, including template polyhedra by Sankaranarayanan et al. [2005], zonotopes by Ghorbal et al. [2009], and parallelotopes by Amato and Scozzari [2012].

Fewer abstract domains go beyond the expressiveness of polyhedra. Linear absolute relation analysis by Chen et al. [2011] slightly generalizes polyhedra to add absolute value operators. Abstract domains for polynomial inequalities — so-called semi-algebraic varieties — have been proposed by Rodríguez-Carbonell and Kapur [2007] based on the complete algorithmic of Gröbner bases, and by Bagnara et al. [2005b] based on an incomplete reduction to polyhedra, but they face scalability issues. To address these issues, domains specialized to very specific subsets of non-linear invariants have been proposed;

they do not generalize affine inequalities and are incomparable with polyhedra. These include the ellipsoid domain by Feret [2004] to track quadratic invariants found in digital filters, the arithmetic-geometric domain by Feret [2005] for selected exponential sequences, and a domain dedicated to quaternion computations presented by Bertrane et al. [2010].

Chapter 6

Domain Transformers

The last two chapters have presented many numeric abstract domains, with various degree of expressiveness, cost, and precision. In this chapter, we study domain transformers that allow deriving new domains by combining existing ones.

We start with the observation that, on the semantic level, the set of abstract domains can be viewed as a lattice, which will prompt us to construct the meet of abstract domains, allowing the combination of two — or more — domains into a more precise one. We present a notion of reduced product construction that complements this semantic construction with an algorithmic view and makes it effective.

We have seen that most of our domains are not closed under union. The approximation induced by abstract unions is a major source of precision loss. The second part of the chapter will address this problem by presenting disjunctive completion methods that add to arbitrary domains the ability to represent precisely, to some extend, disjunctions.

6.1 The Lattice of Abstractions

In this section, we study and compare abstract domains focusing only on their expressiveness. That is, we are only interested in the set of concrete properties that can be exactly represented in the abstract. Moreover, we assume the presence of a Galois connection or, more precisely, as an abstract domain is viewed here as a subset of the concrete world, the presence of a Galois embedding (Sect. 2.3.4).

This view is reductive as it ignores the problem of abstract element representation and, although it provides a semantic notion of best abstraction $\alpha \circ F \circ \gamma$ of concrete operators F, it does not provide effective algorithms, nor does it help computing fixpoints in finite time. It is also useless to discuss about important abstract domains, such as polyhedra, which do not feature a Galois connection. Finally, note that expressiveness does not equal precision: an analysis in a more expressive domain may turn out to be less precise in practice, due to imprecise abstract operators, accumulating imprecisions when combining them, or due to widening application — see Sect. 4.7.3 for an example. This view nevertheless provides important insights which will be useful when we will turn to effective constructions.

Abstract domains. We focus on numeric abstract domains over a set $\mathbb{V}$ of variables with values in $\mathbb{I}$, so that our concrete domain is $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E})$ where $\mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{I}$. An abstract domain is a set $\mathcal{D}^\sharp$ connected to $\mathcal{D}$ through a Galois embedding $(\mathcal{D}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$. By Def. 2.13, γ is injective, so that $\mathcal{D}^\sharp$ is isomorphic to $\gamma(\mathcal{D}^\sharp)$, a subset of $\mathcal{D}$. We will thus assume, without loss of generality, that $\mathcal{D}^\sharp \subseteq \mathcal{D}$. This amounts, for instance, to stating that an interval is a convex subset of values, rather than a pair of bounds representing an interval.

Example 6.1 (Interval abstraction revisited)**.** *As an example, the integer interval domain was abstracted from $\mathcal{P}(\mathbb{Z})$ through the abstraction (4.6). We rephrase it as follows, given our view of intervals as sets:*

$$\alpha_{itv}(S) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \textit{if } S = \emptyset \\ \{\, x \in \mathbb{Z} \mid \min S \leq x \leq \max S \,\} & \textit{otherwise} \end{cases}$$

Moore families. Recall from Thm. 2.4.6 that the set of properties in an abstract domain is closed under intersection. Such a set is called a *Moore family*. As observed by Cousot and Cousot [1979a], there is an equivalence between being a Moore family and the existence of a Galois embedding. Given a Galois embedding (α, γ) between $\mathcal{D}$ and a subset $\mathcal{D}^\sharp$, the Moore family is simply $\mathcal{D}^\sharp = \alpha(\mathcal{D})$. Given a Moore family, $\mathcal{D}^\sharp \subseteq \mathcal{D}$, the Galois embedding is given by:

$$\begin{array}{lcl} \alpha(S) & \stackrel{\text{def}}{=} & \bigcap \{\, S^\sharp \in \mathcal{D}^\sharp \mid S \subseteq S^\sharp \,\} \\ \gamma(S^\sharp) & \stackrel{\text{def}}{=} & S^\sharp \end{array} \tag{6.1}$$

The intersection in the definition of α always exists by virtue of the Moore family property. Thus, in the following, we will equate an abstract domain with a subset of $\mathcal{D}$ which forms a Moore family.

Partial order. In the previous chapters, when we provided a Galois connection, we always connected an abstract domain with the concrete domain. As all abstract domains are now subsets of $\mathcal{D}$, we can also compare two abstract domains, using the reverse inclusion $\supseteq$: a domain $\mathcal{D}_1$ is more precise than a domain $\mathcal{D}_2$ if $\mathcal{D}_1 \supseteq \mathcal{D}_2$. Hence, abstractions form a poset, ordered by $\supseteq$.

To simplify the presentation of the partial order, we illustrate this order on value abstract domains for integers, i.e., abstracting $\mathcal{P}(\mathbb{Z})$, but these can be lifted to abstractions of $\mathcal{P}(\mathcal{E})$ through (4.1). For instance, the simple sign domain (Fig. 4.2.(a)) is an abstraction of the interval domain, as any set of integers that can be represented by a sign can be represented by an interval. The congruence domain is not comparable with the interval domain as neither set of integer sets is included in the other. We illustrate this partial order in Fig. 6.1.

Lattice structure. An important result, stated by Cousot and Cousot [1979a], is that the poset of abstractions is actually a complete lattice. The least element, i.e., the most

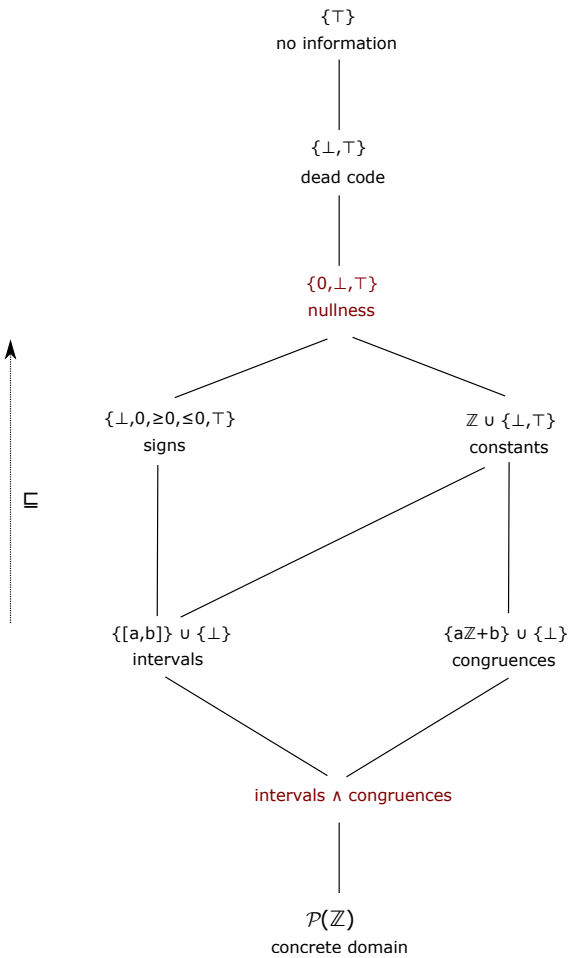


Figure 6.1: Hasse diagram for a part of the lattice of value abstractions, showing some domains from Chap. 4, and their combination by lub and glb.

precise domain, is naturally the concrete domain, at the bottom of Fig. 6.1. The greatest element, i.e., the least precise domain, at the top of Fig. 6.1, only features a single element $\{\top\}$, so that it brings no information. The domain immediately below in Fig. 6.1 has two elements: $\{\bot, \top\}$. While it may not seem very powerful as it cannot give any information about variable values, it is nevertheless able to distinguish definitely dead code ($\bot$, as the variable has no possible value) from potentially live code ($\top$).

The least upper bound of two domains gives the most precise domain which is coarser than both domains. As the intersection of two Moore families remains a Moore family, the lub simply selects the sets of integers representable in both domains. For instance, the lub of intervals and congruences gives the constant domain, as shown again in Fig. 6.1. The lub of signs and constants, which is the same as the lub of signs and congruences, gives a new domain, nullness $\{0, \bot, \top\}$, only able to state whether a variable is necessarily null or possibly not.

The greatest lower bound is far more interesting. It gives the coarsest domain that is more precise than both domains. It must naturally be able to represent properties from either domain. But, as the result is a Moore family and must be closed under intersection, the glb must be able to represent exactly *conjunctions* of properties from both domains. Generally, this includes many more properties than those in either domain. Another justification for this is that the union of two Moore families is generally not a Moore family, and must be enriched with new elements. In Fig. 6.1, we show the glb of the interval and the congruence domains. We obtain a new domain that can represent exactly integer sets of the form $[a, b] \cap (c\mathbb{Z} + b)$, such as for instance: $\{0, 2, 4\}$.

The glb of domains allows deriving new, more expressive domains, from existing ones, and is thus an important tool in static analysis design. The next section discusses how to effectively construct the glb of two, or more domains.

6.2 Product Domains

The previous section justified, from a semantic point of view, the existence of a domain combiner able to construct a new, more precise abstract domain from two existing domains.

We revert now to a more effective view of abstract domains. We thus assume that we are given two domains, $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$, following Def. 3.1 — or, when discussing non-relational domains, two value domains, $\mathcal{B}_1^\sharp$ and $\mathcal{B}_2^\sharp$, following Def. 4.1. We will denote as $\mathcal{D}_{1\times 2}^\sharp$ the glb of both domains. As seen in the last section, $\mathcal{D}_{1\times 2}^\sharp$ expresses conjunctions of properties from the argument domains. We will naturally represent them using a *pair* of properties, i.e.:

$$\mathcal{D}_{1\times 2}^\sharp \stackrel{\text{def}}{=} \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp \tag{6.2}$$

hence, $\mathcal{D}_{1\times 2}^\sharp$ is a *product domain*.

loc.	(a)	(b)	(c)	(d)
3	$V \in [11, 12]$	$V \in 2Z + 1$	$V \in [11, 12] \wedge V \in 2Z + 1$	$V = 11$
4	$V = 12$	$V \in 2\mathbb{Z} + 1$	$V = 12 \wedge V \in 2\mathbb{Z} + 1$	$\perp$
5	$V = 0$	$V \in 2\mathbb{Z}$	$V = 0 \wedge V = 0$	$\perp$
6	$V \in [0, 11]$	$V \in \mathbb{Z}$	$V \in [0, 11] \wedge V \in \mathbb{Z}$	$V = 11$

Table 6.1: Result of the analysis of the program (6.3) at selected program points 3–6 in: (a) the interval domain, (b) the congruence domain, (c) the direct product of intervals and congruences, and (d) their reduced product.

6.2.1 Motivating Example

We consider, as motivating example, the following program:

$$\begin{array}{ll} 1: & V \leftarrow 1; \\ 2: & \textbf{while } V \leq 10 \textbf{ do } V \leftarrow V + 2 \textbf{ done}; \\ 3: & \textbf{if } V \geq 12 \textbf{ then} \\ 4: & \qquad V \leftarrow 0 \\ 5: & \textbf{endif} \\ 6: & \end{array} \tag{6.3}$$

In the concrete, the loop iterates on odd numbers, from 1 to 11, at which point the loop exits with $V = 11$ and the **then** branch of the following conditional is not executed, so that V remains at 11 when the program ends at point 6. We now consider a static analysis in the non-relational domains of intervals (Sect. 4.5) and congruences (Sect. 4.8). The results are summarized in columns (a)–(b) of Table 6.1.

- The interval domain finds, as loop invariant, $[0, 12]$. It exits the loop with invariant $[11, 12]$. Hence, the **then** branch of the conditional can be executed with $V = 12$, V is reset to 0 and we get, at the end of the program, $V \in [0, 11]$ — the join of 0 and 11.

- The congruence domain finds, as loop invariant, $V \in 2\mathbb{Z} + 1$, i.e., V is odd. The test $V \geq 12$ is abstracted as the identity, so that the **then** branch is executed and we find $V = 0$ at the end of the branch (as congruences can also exactly represent singletons). After the conditional, we get that V is either odd or 0, hence, we loose all congruence information: $V \in \mathbb{Z}$.

Although the invariant $V = 11$ at the end of the program is expressible in both domains, neither domain can infer it on its own. We see, however, that interval and congruence information would benefit from being combined together to prove that $V = 12$ is not possible as V is necessarily odd during the loop.

6.2.2 Direct Product

A first idea to define the operators required on $\mathcal{D}^\sharp_{1\times 2} \stackrel{\text{def}}{=} \mathcal{D}^\sharp_1 \times \mathcal{D}^\sharp_2$ from those available on $\mathcal{D}^\sharp_1$ and $\mathcal{D}^\sharp_2$ is to apply them element-wise.

Definition 6.1 (Direct product)**.** *The direct product* $\mathcal{D}^\sharp_{1\times 2}$ *of* $\mathcal{D}^\sharp_1$ *and* $\mathcal{D}^\sharp_2$ *is defined as:*

- $\mathcal{D}^\sharp_{1\times 2} \stackrel{\text{def}}{=} \mathcal{D}^\sharp_1 \times \mathcal{D}^\sharp_2$
- $\gamma_{1\times 2}(A^\sharp_1, A^\sharp_2) \stackrel{\text{def}}{=} \gamma_1(A^\sharp_1) \cap \gamma_2(A^\sharp_2)$;
- $(A^\sharp_1, A^\sharp_2) \sqsubseteq^\sharp_{1\times 2} (B^\sharp_1, B^\sharp_2) \stackrel{\text{def}}{\iff} (A^\sharp_1 \sqsubseteq^\sharp_1 B^\sharp_1) \wedge (A^\sharp_2 \sqsubseteq^\sharp_2 B^\sharp_2)$;
- $\bot^\sharp_{1\times 2} \stackrel{\text{def}}{=} (\bot^\sharp_1, \bot^\sharp_2)$;
- $\top^\sharp_{1\times 2} \stackrel{\text{def}}{=} (\top^\sharp_1, \top^\sharp_2)$;
- $\alpha_{1\times 2}(S) \stackrel{\text{def}}{=} (\alpha_1(S), \alpha_2(S))$
  optionally, if $\mathcal{D}^\sharp_1$ *and* $\mathcal{D}^\sharp_2$ *feature abstraction functions;*
- $\mathbb{S}^\sharp[\![\, s \,]\!]_{1\times 2}(A^\sharp_1, A^\sharp_2) \stackrel{\text{def}}{=} (\mathbb{S}^\sharp[\![\, s \,]\!]_1 A^\sharp_1, \mathbb{S}^\sharp[\![\, s \,]\!]_2 A^\sharp_2)$;
- $\mathbb{C}^\sharp[\![\, c \,]\!]_{1\times 2}(A^\sharp_1, A^\sharp_2) \stackrel{\text{def}}{=} (\mathbb{C}^\sharp[\![\, c \,]\!]_1 A^\sharp_1, \mathbb{C}^\sharp[\![\, c \,]\!]_2 A^\sharp_2)$;
- $(A^\sharp_1, A^\sharp_2) \cup^\sharp_{1\times 2} (B^\sharp_1, B^\sharp_2) \stackrel{\text{def}}{\iff} (A^\sharp_1 \cup^\sharp_1 B^\sharp_1, A^\sharp_2 \cup^\sharp_2 B^\sharp_2)$;
- $(A^\sharp_1, A^\sharp_2) \cap^\sharp_{1\times 2} (B^\sharp_1, B^\sharp_2) \stackrel{\text{def}}{\iff} (A^\sharp_1 \cap^\sharp_1 B^\sharp_1, A^\sharp_2 \cap^\sharp_2 B^\sharp_2)$;
- $(A^\sharp_1, A^\sharp_2) \triangledown_{1\times 2} (B^\sharp_1, B^\sharp_2) \stackrel{\text{def}}{\iff} (A^\sharp_1 \triangledown_1 B^\sharp_1, A^\sharp_2 \triangledown_2 B^\sharp_2)$. ■

A pair $(A^\sharp_1, A^\sharp_2) \in \mathcal{D}^\sharp_{1\times 2}$ represents a conjunction of properties, i.e., the intersection of the sets defined by $\gamma_1(A^\sharp_1)$ and $\gamma_2(A^\sharp_2)$. We can prove easily that $\gamma_{1\times 2}$ is monotonic, that all the operators on $\mathcal{D}^\sharp_{1\times 2}$ are sound, provided that those on $\mathcal{D}^\sharp_1$ and $\mathcal{D}^\sharp_2$ are, that the widening enforces termination, and if (α_1, γ_1) and (α_2, γ_2) are Galois connections, then so is $(\alpha_{1\times 2}, \gamma_{1\times 2})$, i.e., we obey the definition of an abstract domain (Def. 3.1).

The product of two value abstract domains (Def. 4.1) is similarly defined pointwise. For instance, we would define a sound abstract addition as: $(A^\sharp_1, A^\sharp_2) +^\sharp_{1\times 2} (B^\sharp_1, B^\sharp_2) \stackrel{\text{def}}{\iff} (A^\sharp_1 +^\sharp_1 B^\sharp_1, A^\sharp_2 +^\sharp_2 B^\sharp_2)$.

Example analysis. Coming back to the program example of Sect. 6.2.1, we perform an analysis in the non-relational abstract domain obtained by instantiating the generic non-relational construction of Sect. 4.1 with the product of intervals and congruences. The results, shown in Table 6.1.(c), are disappointing: we still find $V \in [0, 11]$ at the end of the program.

Indeed, at program point 3, the analysis finds the pair of invariants $([11, 12], 2\mathbb{Z} + 1)$. Then, the test $V \geq 12$ is performed *independently* on both domains, which gives $(12, 2\mathbb{Z}+1)$ at location 4, and the assignment gives $(0, 0)$ at location 5. We get the same result by analyzing the program in the product domain as when performing the analyses separately, and then taking the conjunction of the results at each program point.

The issue is that, although we perform the analysis in a more expressive domain, our operators do not benefit from this. To improve the precision, it is necessary to allow some form of communication between the domains, so that the analysis in each domain benefits from the information available in the other domain.

6.2.3 Fully Reduced product

The notion of *reduced product* addresses the precision issues of the direct product from the previous section. It keeps the idea of applying the abstract operations in parallel in each domain, but it interleaves these operations with a so-called *reduction* step, that transfers information between the components of a pair of abstract elements.

We consider first the ideal case, where both $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ feature Galois connections. An optimal reduction can be then be defined as follows:

Definition 6.2 (Optimal reduction)**.** *The reduction* $\rho : \mathcal{D}_{1\times 2}^\sharp \to \mathcal{D}_{1\times 2}^\sharp$ *between the domains* $\mathcal{D}_1^\sharp$ *and* $\mathcal{D}_2^\sharp$ *is defined as:*

$$\rho(A_1^\sharp, A_2^\sharp) \stackrel{\text{def}}{=} (\alpha_1(\gamma_{1\times 2}(A_1^\sharp, A_2^\sharp)),\ \alpha_2(\gamma_{1\times 2}(A_1^\sharp, A_2^\sharp)))$$

■

Example 6.2 (Reduction between intervals and congruences)**.** *We define a reduction on value abstractions* $\rho_b : \mathcal{B}^\sharp \to \mathcal{B}^\sharp$ *where* $\mathcal{B}^\sharp \stackrel{\text{def}}{=} \mathcal{B}_{int}^\sharp \times \mathcal{B}_{cong}^\sharp$ *is the product of the interval domain and the congruence domain:*

$$\rho_b([a,b], c\mathbb{Z}+d) \stackrel{\text{def}}{=} \begin{cases} (\bot_b^\sharp, \bot_b^\sharp) & \text{if } a' > b' \\ ([a',a'], 0\mathbb{Z}+a') & \text{if } a' = b' \\ ([a',b'], c\mathbb{Z}+d) & \text{if } a' < b' \end{cases}$$

where:

$$\begin{aligned} a' &\stackrel{\text{def}}{=} \min\{\, x \geq a \mid x \equiv d\ [c] \,\} \\ b' &\stackrel{\text{def}}{=} \max\{\, x \leq b \mid x \equiv d\ [c] \,\} \end{aligned}$$

This reduction operates in two steps. Firstly, we use the congruence information $c\mathbb{Z}+d$ *to tighten the bounds of* $[a,b]$*, stating that the new bounds must be in* $c\mathbb{Z}+d$*; we get the interval* $[a',b']$*. Secondly, we consider the two special cases where the interval information can be exactly represented in the congruence domain and refine it: the empty interval* $\bot_b^\sharp$*, and the singleton* $[a',a']$*; otherwise, the second step returns the congruence information unchanged. We can check that these two steps are sufficient to actually compute the optimal reduction, as defined semantically in Def. 6.2. For instance,* $\rho_b([11,12], 2\mathbb{Z}+1) = ([11,11], 0\mathbb{Z}+11)$*.*

The reduction can then be extended point-wise to non-relational states $\mathcal{D}^\sharp$ *based on* $\mathcal{B}^\sharp$*:* $\rho(X^\sharp) \stackrel{\text{def}}{=} \lambda V \in \mathbb{V}.\ \rho_b(X^\sharp(V))$*.* ♦

Analysis with reduction. We now modify the direct product domain defined in Sect. 6.2.2, applying a reduction after each operation. More precisely:

Definition 6.3 (Reduced product)**.** *The reduced product $\mathcal{D}^\sharp_{1\times 2}$ of $\mathcal{D}^\sharp_1$ and $\mathcal{D}^\sharp_2$ is defined as the direct product of Def. 6.1 where the following operators are modified to apply the reduction ρ:*

- $\mathbb{S}^\sharp[\![s]\!]_{1\times 2}(A^\sharp_1, A^\sharp_2) \stackrel{\text{def}}{=} \rho(\mathbb{S}^\sharp[\![s]\!]_1 A^\sharp_1, \mathbb{S}^\sharp[\![s]\!]_2 A^\sharp_2)$;
- $\mathbb{C}^\sharp[\![c]\!]_{1\times 2}(A^\sharp_1, A^\sharp_2) \stackrel{\text{def}}{=} \rho(\mathbb{C}^\sharp[\![c]\!]_1 A^\sharp_1, \mathbb{C}^\sharp[\![c]\!]_2 A^\sharp_2)$;
- $(A^\sharp_1, A^\sharp_2) \cup^\sharp_{1\times 2} (B^\sharp_1, B^\sharp_2) \stackrel{\text{def}}{\Longleftrightarrow} \rho(A^\sharp_1 \cup^\sharp_1 B^\sharp_1, A^\sharp_2 \cup^\sharp_2 B^\sharp_2)$;
- $(A^\sharp_1, A^\sharp_2) \cap^\sharp_{1\times 2} (B^\sharp_1, B^\sharp_2) \stackrel{\text{def}}{\Longleftrightarrow} \rho(A^\sharp_1 \cap^\sharp_1 B^\sharp_1, A^\sharp_2 \cap^\sharp_2 B^\sharp_2)$.

Alternatively, if a reduction ρ_b is defined on an abstract value domain, we would apply it after every operator from Def. 4.1, for instance: $(A^\sharp_1, A^\sharp_2) +^\sharp_{1\times 2} (B^\sharp_1, B^\sharp_2) \stackrel{\text{def}}{=} \rho_b(A^\sharp_1 +^\sharp_1 B^\sharp_1, A^\sharp_2 +^\sharp_2 B^\sharp_2)$. ■

The benefit of this method is that we reuse all the data-structures and algorithms developed for each domain. Each new reduced product only requires a single additional function tied to the pair of chosen domains, which is much less effort than redesigning a new domain and all its operators from scratch.

However, this ease may come at the cost of precision. Indeed, applying a reduction operator after an abstract operator, even in the best scenario where both are optimal, amounts to combining optimal operators, and we know that it may not give the best abstraction of the combination.

Example analysis. Coming back to the example of Sect. 6.2.1, we perform an analysis in the reduced product of intervals and congruences. The results are shown in Table 6.1.(d). At point 3, the loop outputs the pair $([11, 12], 2\mathbb{Z}+1)$ which is reduced into $([11, 11], 0\mathbb{Z}+11)$ by the reduction from Ex. 6.2. Thus, the interval test $\mathbb{C}^\sharp[\![V \geq 12]\!]_{int}$ will output $\bot^\sharp_{int}$ which, by further reduction, gives $(\bot^\sharp_{int}, \bot^\sharp_{cong})$, despite the fact that the test is the identity in the congruence domain. Hence, the **then** branch is dead for both domains. The program terminates with the result of the **else** branch only, i.e., $V = 11$.

Remark 6.1. *Note that, when considering the reduced product of non-relational domains, we have a choice of either constructing the reduced product of the underlying value abstract domain, which is then lifted to a state domain following Def. 4.1, or constructing separate state domains following Def. 4.1, and then apply the reduction lifted to states. Although, on our example, this does not make any difference, in general, the former is more precise as the reduction is applied at a finer granularity — e.g., after each abstract operator $+^\sharp_b$, $-^\sharp_b$, etc., instead of once after each atomic statement $\mathbb{S}^\sharp[\![V \leftarrow e]\!]$, $\mathbb{C}^\sharp[\![c]\!]$.* ◊

Reduction and widening. Note that the reduced product of Def. 6.3 applies the reduction operator ρ after a join $\cup^\sharp$ and after an intersection $\cap^\sharp$, but refrains from applying it after a widening $\triangledown$. Indeed, although applying the widening on each component of the product independently ensures termination, this is no longer the case if we apply a reduction in-between widening steps. Intuitively, the reduction may strengthen the abstract information, such as providing finite interval bounds, while the widening enforces convergence by setting bounds to infinity.

Example 6.3. *The situation is similar to that of the zone widening from Sect. 5.4.4. We can actually rephrase the counter-example of Ex. 5.18, initially stated in the zone domain, as the reduced product of two domains: the interval domain, tracking the bounds of X and Y, and a special domain that tracks the bound of $X - Y$ only. The optimal reduction operator ρ would use the bounds discovered on $X - Y$ and X to refine Y, and the bound on $X - Y$ on Y to refine X — simulating the effect of the zone closure, and preventing convergence. Additional examples can be found in [Cousot et al., 2006].* ♦

Our solution is to avoid performing a reduction after a widening application. Other solutions proposed in the literature include strengthening the definition of the widening to make it robust against interference from reduction, such as proposed by Cousot et al. [2006].

6.2.4 Partially Reduced Products

The reduction of Def. 6.2 relies on Galois connections, and thus suffers from the same drawbacks as the optimal abstraction of operators F as $\alpha \circ F \circ \gamma$ (Thm. 2.5). In case no Galois connection exists, or Def. 6.2 cannot be efficiently implemented, we can settle for a *partial reduction*, that is, an operator on $\mathcal{D}^\sharp_{1\times 2}$ that is only guaranteed to be sound and to only improve precision, never degrade it:

Definition 6.4 (Partial reduction)**.** *An operator $\rho : \mathcal{D}^\sharp_{1\times 2} \to \mathcal{D}^\sharp_{1\times 2}$ is a partial reduction between the domains $\mathcal{D}^\sharp_1$ and $\mathcal{D}^\sharp_2$ if:*

$$(Y_1^\sharp, Y_2^\sharp) = \rho(X_1^\sharp, X_2^\sharp) \implies \begin{array}{l} \gamma_{1\times 2}(Y_1^\sharp, Y_2^\sharp) = \gamma_{1\times 2}(X_1^\sharp, X_2^\sharp) \wedge \\ \gamma_1(Y_1^\sharp) \subseteq \gamma_1(X_1^\sharp) \wedge \\ \gamma_2(Y_2^\sharp) \subseteq \gamma_2(X_2^\sharp) \end{array}$$

The first condition, i.e., the equality up to $\gamma_{1\times 2}$, states the soundness. The two inclusions state that, although the product concretization is the same, each element has been strengthened in its respective domain. ■

Our partial reduction can replace the optimal reduction in the reduced product of Def. 6.3 to obtain a more flexible, partially reduced product, applicable in the absence of a Galois connection and allowing various cost/precision trade-offs in the reduction.

Simple reduction. Given an arbitrary pair of domains $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$, we can define the following reduction that merges the least elements:

$$\rho(A_1^\sharp, A_2^\sharp) \stackrel{\text{def}}{=} \begin{cases} (\bot_1^\sharp, \bot_2^\sharp) & \text{if } (A_1^\sharp = \bot_1^\sharp) \vee (A_2^\sharp = \bot_2^\sharp) \\ (A_1^\sharp, A_2^\sharp) & \text{otherwise} \end{cases}$$

This reduction is extremely simple, yet, it is more precise than the direct product and can be useful in some cases. Consider, for instance, a conditional **if** $\cdots$ **else** $\cdots$ **endif**, where the end of one branch can be proven to be unreachable using one domain, i.e., we get an element $(\bot_1^\sharp, A_2^\sharp)$, while the end of the other branch can be proven to be unreachable using the other domain, as $(A_1^\sharp, \bot_2^\sharp)$. Then, the reduced product will reduce both branches to $(\bot_1^\sharp, \bot_2^\sharp)$ and discover that the code after the conditional is not reachable. A non-reduced product, however, will compute the join $(\bot_1^\sharp, A_2^\sharp) \cup_{1\times 2}^\sharp (A_1^\sharp, \bot_2^\sharp) = (A_1^\sharp, A_2^\sharp)$, and continue the analysis with a non-empty state after the conditional.

Reducing intervals and affine equalities. As a more interesting example, we consider a partially reduced product between the interval domain and the affine equalities domain from Sect. 5.2. Our goal is to construct a relational domain able to express bounds and also arbitrary affine equalities, without resorting to the polyhedra domain, which is rather costly.

We can exploit the methods developed in constraint programming to solve interval linear equality systems to design our reduction. For instance, the interval Gauss-Seidel method works as follows: given an interval information mapping an interval $[a_i, b_i]$ to each variable V_i, then, for each constraint $\sum_i \alpha_{ij} V_i = \beta_j$, denoting as V_k the variable appearing in leading position, i.e., $\alpha_{kj} = 1$, we refine the bounds of V_k by propagating the bounds of the V_i in the equation: $V_k = \beta_j - \sum_{i \neq k} \alpha_{ij} V_i$. Thus, we replace $[a_k, b_k]$ with $[a_k, b_k] \cap_b^\sharp (\beta_j -_b^\sharp \sum_{i\neq k}^\sharp \alpha_{ij} \times_b^\sharp [a_i, b_i])$, evaluated in the interval domain.

The algorithm has a quadratic time cost in the worst case, as each coefficient in the system is used once. As expected, it does not perform an optimal reduction. We refer the interested reader to the work of Chiu and Lee [2002] for more information and suggestions for improvements on the interval Gauss-Seidel method

Local iterations. In the reduction between intervals and affine equalities, we used one domain (affine equalities) to refine the other one (intervals). Another example, the optimal reduction between intervals and congruences (Ex. 6.2), could be decomposed into two steps, where each step refines only one of the domains, using the other one. This idea is generalized by Granger [1992], who suggests constructing $\rho : \mathcal{D}_{1\times 2}^\sharp \to \mathcal{D}_{1\times 2}^\sharp$ from two functions $\rho_1 : \mathcal{D}_{1\times 2}^\sharp \to \mathcal{D}_1^\sharp$ and $\rho_2 : \mathcal{D}_{1\times 2}^\sharp \to \mathcal{D}_2^\sharp$, refining one domain at a time. He further observes that not all information is always transported from one domain to the other in one application of ρ_1 or ρ_2, and it is worth iterating them. We then obtain a decreasing sequence, and actually retrieve the method of *local iterations*, already mentioned in non-relational tests (Sect. 4.6.3).

While, in constraint programming settings, such iterations are generally performed until the fixpoint is reached, we can, in order to be more efficient, only iterate a few times, or use a narrowing operator (Sect. 4.7.2).

In the reduction between intervals and congruences, only one application of each reduction is necessary, provided we first refine intervals from congruences, and then congruences from intervals. In the reduction between intervals and affine equalities, although we only refine one way, from affine equalities to intervals, it is worth iterating this process several times to gain more precision.

$N-$ary reduced product. So far, our (partially) reduced products only involved two abstract domains. It is naturally possible to build (partially) reduced products for more than two domains.

One simple idea is to consider a $n-$ary combination as a sequence of binary combinations. For instance, the product of $\mathcal{D}_1^\sharp$, $\mathcal{D}_2^\sharp$, and $\mathcal{D}_3^\sharp$ can be written as $(\mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp) \times \mathcal{D}_3^\sharp$ or $\mathcal{D}_1^\sharp \times (\mathcal{D}_2^\sharp \times \mathcal{D}_3^\sharp)$, or even $\mathcal{D}_2^\sharp \times (\mathcal{D}_1^\sharp \times \mathcal{D}_3^\sharp)$. Note that the parenthesizing matters, even if each product uses an optimal reduction, as optimality does not compose.

Alternatively, we can construct directly a product of N domains $\mathcal{D}_1^\sharp$, ..., $\mathcal{D}_N^\sharp$. The optimal reduction of Def. 6.2 generalizes easily to N domains, as:

$$\begin{aligned}
&\rho(A_1^\sharp, \ldots, A_N^\sharp) \stackrel{\text{def}}{=} (\alpha_1(\gamma(A_1^\sharp, \ldots, A_N^\sharp)), \ldots, \alpha_N(\gamma(A_1^\sharp, \ldots, A_N^\sharp))) \\
&\text{where } \gamma(A_1^\sharp, \ldots, A_N^\sharp) \stackrel{\text{def}}{=} \gamma_1(A_1^\sharp) \cap \cdots \cap \gamma_N(A_N^\sharp)
\end{aligned}$$

and, likewise, the properties a partial reduction for N domains must satisfy can be easily deduced by generalizing Def. 6.4.

The method by Granger [1992], discussed above on the case of two domains, provides however a much more practical solution: given $\mathcal{D}^\sharp \stackrel{\text{def}}{=} \mathcal{D}_1^\sharp \times \cdots \times \mathcal{D}_N^\sharp$, we provide a reduction $\rho_i : \mathcal{D}^\sharp \to \mathcal{D}_i^\sharp$ for each domain $\mathcal{D}_i^\sharp$, and apply them in turn, possibly using local decreasing iterations to improve the precision. With this method, it is extremely easy to add a new abstract domain to an analysis. Given a domain $\mathcal{D}_{N+1}^\sharp$ to add, we:

- add a reduction $\rho_{N+1} : \mathcal{D}^\sharp \to \mathcal{D}_{N+1}^\sharp$;
- optionally improve some of the ρ_i in case they can benefit from the information available in $\mathcal{D}_{N+1}^\sharp$, which is now included in $\mathcal{D}^\sharp$.

This leads to an attractive modular, scalable, and extensible design for static analyzers.

We refer the reader to the description of the reduced product used in the Astrée analyzer by Cousot et al. [2006] for further ideas on the subject and practical implementation guidelines.

6.2.5 Variable Packing

As a last application of the reduced product, we consider the case where an expensive domain, such as polyhedra, is not applied to the whole set $\mathbb{V}$ of variables, but rather

to several smaller subsets of $\mathbb{V}$: $\mathbb{V}_1, \ldots, \mathbb{V}_N \subseteq \mathbb{V}$, called *variable packs*, in order to trade precision for efficiency. On the one hand, we will miss relations between variables that do not belong to the same pack; on the other hand, if we keep the sizes of the packs bounded, and the number of packs linear in the number of variables, we can hope for an analysis that scales linearly with the number of variables instead of super-linearly.

More precisely, given a relational base abstract domain $\mathcal{D}^\sharp$, we construct an abstract domain $\overline{\mathcal{D}}^\sharp$ where an element is composed of an element of $\mathcal{D}^\sharp$ for each pack:

$$\overline{\mathcal{D}}^\sharp \stackrel{\text{def}}{=} \{1, \ldots, N\} \to \mathcal{D}^\sharp$$

which can be seen as a product of N copies of $\mathcal{D}^\sharp$, albeit with different variable sets.

In an abstract element $(X_1^\sharp, \ldots, X_N^\sharp) \in \overline{\mathcal{D}}^\sharp$, each $X_i^\sharp$ can only track the relationships between the variables in $\mathbb{V}_i$, which causes a loss of precision. To recover some precision, we allow variables to appear in several packs, i.e., we do not necessary have $\mathbb{V}_i \cap \mathbb{V}_j = \emptyset$ when $i \neq j$. Given $i \neq j$ such that $\mathbb{V}_i \cap \mathbb{V}_j \neq \emptyset$, then we can employ a reduction to transfer information about $\mathbb{V}_i \cap \mathbb{V}_j$ between $X_i^\sharp$ an $X_j^\sharp$. Variable packing is thus, indeed, an instance of reduced product.

Reduction. There are several possible reductions over $\overline{\mathcal{D}}^\sharp$, with different cost versus precision trade-offs:

1. The simplest reduction would use a value abstraction, such as intervals, to transfer information about each variable independently. More precisely, for each variable $V \in \mathbb{V}_i \cap \mathbb{V}_j$, we extract a value abstraction $A^\sharp$ for V from $X_i^\sharp$ and $B^\sharp$ from $X_j^\sharp$. We then compute the intersection, in the value abstraction, $A^\sharp \cap_b^\sharp B^\sharp$. Finally, we inject this value into $X_i^\sharp$ and $X_j^\sharp$. When using intervals, for instance, the injection can take the form of conditions $\mathbb{C}^\sharp[\![(a \leq V) \wedge (V \leq b)]\!]$ enforcing the new bounds.

2. We can be more precise, and more costly, by transporting relational information between $X_i^\sharp$ and $X_j^\sharp$. More precisely, we would first project both $X_i^\sharp$ and $X_j^\sharp$ over the variable set $\mathbb{V}_i \cap \mathbb{V}_j$ by eliminating spurious variables. Then we intersect the results in $\mathcal{D}^\sharp$ to get a core that is valid in both $X_i^\sharp$ and $X_j^\sharp$. Finally we inject the core into $X_i^\sharp$ and $X_j^\sharp$. One method to perform this injection is to extend the core to discuss about all variables in $\mathbb{V}_i$ — resp. $\mathbb{V}_j$ — by setting missing variables to $[-\infty, +\infty]$, and then intersect, in $\mathcal{D}^\sharp$, the extended core with $X_i^\sharp$ — resp. $X_j^\sharp$.

3. Projecting out variables before performing the intersection may lose information. Thus, we can imagine a more precise, but more costly method that avoids this loss of information by computing the intersection over $\mathbb{V}_i \cup \mathbb{V}_j$ instead of $\mathbb{V}_i \cap \mathbb{V}_j$. More precisely, both $X_i^\sharp$ and $X_j^\sharp$ are first extended to $\mathbb{V}_i \cup \mathbb{V}_j$ by adding missing variables, initialized to $[-\infty, +\infty]$. We then compute the intersection, and project back respectively onto $\mathbb{V}_i$ and $\mathbb{V}_j$ to get the new abstract elements, respectively $X_i^\sharp$ and $X_j^\sharp$.

This method, defined on pairs of packs, can be generalized to more than two packs at once. Alternatively, the pairwise reduction can be iterated over pairs of packs, following Granger [1992]'s local iteration method. Further reduction techniques, based on graph closure algorithms and targeting specifically packing domains, are discussed by Bouaziz [2012].

Packed domain. We now present the abstract operations on $\overline{\mathcal{D}}^\sharp$, derived from those on individual packs in $\mathcal{D}^\sharp$. All the binary operations: join $\cup^\sharp$, intersection $\cap^\sharp$, widening $\triangledown$, narrowing $\vartriangle$, are performed element-wise, applying the operation on the abstract elements corresponding to the same pack.

For assignments $\overline{\mathbb{S}}^\sharp[\![V \leftarrow e]\!]$, however, we only need to apply the abstract operation $\mathbb{S}^\sharp[\![V \leftarrow e]\!]X_i^\sharp$ to abstract elements $X_i^\sharp$ corresponding to packs $\mathbb{V}_i$ where V appear, i.e., $V \in \mathbb{V}_i$. Indeed, the abstract elements for packs where V does not occur are unchanged. The problem is that the expression e may feature variables that are not in $\mathbb{V}_i$, so that $\mathbb{S}^\sharp[\![V \leftarrow e]\!]$ cannot be interpreted in $X_i^\sharp$. A simple solution is to apply $\mathbb{S}^\sharp[\![V \leftarrow e']\!]X_i^\sharp$, where the expression e' is obtained from e by replacing any variable $W \notin \mathbb{V}_i$ with a sound interval, obtained from other packs where W occurs.

Similarly, a condition $\overline{\mathbb{C}}^\sharp[\![e_1 \bowtie e_2]\!]$ needs only be applied to abstract elements for packs containing at least one variable occurring in either e_1 or e_2. For each such pack $\mathbb{V}_i$, as for assignments, we project e_1 and e_2 on the variables of $\mathbb{V}_i$.

Packing heuristics. Similarly to the template domain from Sect. 5.5, the packing domain is configured by some external input provided by the user, here in the form of packs. A good strategy is to group variables that appear syntactically together in the same expressions, and perform a limited form of transitive closure: to limit the growth of packs, the accumulation of related variables by transitivity in a pack can be limited at natural syntactic boundaries, such as functions or code blocks. We refer the reader to [Miné, 2006a] for a more detailed description of a possible packing technique, applied to the special case of the octagon domain, which provides packs that are scalable in practice.

6.3 Disjunctive Completions

In general, abstract domains are closed by intersection, but not by union. For such domains, the abstract intersection $\cap^\sharp$ is exact, but the abstract union $\cup^\sharp$ is approximate. Viewing abstract elements as formulas expressible in some logic, these domains appear as closed under conjunction, but not under disjunction. Among the domains we presented, only the extended sign domain (Sect. 4.2) and the constant set domain (Sect. 4.4) were closed by union, but this was not the case for more expressive domains such as intervals nor the relational domains.[1]

[1] For the sake of completion, let us note that there also exist domains that are closed neither by union, nor by intersection, such as the zonotope domain by Ghorbal et al. [2009], although we do not discuss them in this tutorial.

Unfortunately, static analyses feature a large number of union computations, as they are involved in the semantics of conditionals and loops. A loss of precision in the abstract union can thus have a large negative effect on the overall analysis precision, as we will show in example (6.4).

In this section, we show how to solve this problem through disjunctive completion techniques. We present several domain combiners that, given a base abstract domain, construct a more precise abstract domain that can additionally express exactly some disjunctions of the properties expressible in the base domain. They all employ the same basic idea, generalizing our construction of the constant set domain from Sect. 4.4: we use several abstract elements to represent, symbolically, a disjunction of sets of concrete elements. The domain combiners differ in how they choose to create and maintain these sets of abstract elements.

6.3.1 Motivating Example

Before we present our disjunctive completion techniques, we present a simple example motivating the need for disjunctions, which will also serve to illustrate the various disjunctive constructions:

$$\begin{array}{ll} 1: & X \leftarrow [10, 20]; \\ 2: & Y \leftarrow [0, 1]; \\ 3: & \textbf{if } Y \geq 1 \textbf{ then } X \leftarrow -X \textbf{ endif}; \\ 4: & \textbf{assert } X \neq 0 \end{array} \tag{6.4}$$

This program stores a random positive value in $[10, 20]$ into X; then, depending on the value of the non-deterministic variable Y, X is negated or not. Our goal is to prove that, at the end of the program, $X \neq 0$.

Non-disjunctive analysis. A regular analysis in the interval domain (Sect. 4.5) will take, at point 4, the join of the **then** branch of the conditional, where $X \in [-20, -10]$, and the **else** branch, where $X \in [10, 20]$. Hence, we will find $X \in [-20, 20]$, which is not precise enough to rule out the value 0.

Note that the polyhedra domain (Sect. 5.3), while more precise, will not help here because, as the interval domain, it can only represent convex sets, and we need to represent the non-convex set $[-20, -10] \cup [10, 20]$.

We see, however, that, if we could keep *several intervals*, here $[-20, -10]$ and $[10, 20]$, in a single abstract element, then we could easily prove that $X \neq 0$.

6.3.2 Powerset Completion

A first, natural idea is to use sets of abstract elements or, more precisely, finite sets in order to maintain an effective representation.

Representation. Assume that $\mathcal{D}^\sharp$ is a domain obeying Def. 3.1, which we call the *base domain*. We will denote with a hat, such as $\hat{\mathcal{D}}^\sharp$, the new domain we construct and its operators, to distinguish them from those in $\mathcal{D}^\sharp$.

A first idea is to state $\hat{\mathcal{D}}^\sharp \stackrel{\text{def}}{=} \mathcal{P}_{finite}(\mathcal{D}^\sharp)$. However, we note that, given $A^\sharp \subseteq \mathcal{D}^\sharp$, then for any $Y^\sharp \in A^\sharp$, any other element $X^\sharp \in A^\sharp$ such that $X^\sharp \sqsubseteq^\sharp Y^\sharp$ would be useless and redundant. We forbid the presence of redundant base elements and define $\hat{\mathcal{D}}^\sharp$ as follows:

$$\hat{\mathcal{D}}^\sharp \stackrel{\text{def}}{=} \{\, A^\sharp \in \mathcal{P}_{finite}(\mathcal{D}^\sharp) \mid \forall X^\sharp \neq Y^\sharp \in A^\sharp \colon X^\sharp \not\sqsubseteq^\sharp Y^\sharp \,\} \tag{6.5}$$

Then, a set of base abstract elements represents, symbolically, a union, hence the following concretization:

$$\hat{\gamma}(A^\sharp) \stackrel{\text{def}}{=} \cup \{\, \gamma(X^\sharp) \mid X^\sharp \in A^\sharp \,\}$$

When applied to the interval domain, we obtain an abstract domain able to represent any finite union of boxes.

Order structure. Determining whether $\hat{\gamma}(A^\sharp) = \hat{\gamma}(B^\sharp)$, or even whether $\hat{\gamma}(A^\sharp) \subseteq \hat{\gamma}(B^\sharp)$, is very difficult. We thus use on $\hat{\mathcal{D}}^\sharp$ a much more relaxed ordering relation, called the *Hoare order*:

$$A^\sharp \hat{\sqsubseteq}^\sharp B^\sharp \iff^{\text{def}} \forall X^\sharp \in A^\sharp \colon \exists Y^\sharp \in B^\sharp \colon X^\sharp \sqsubseteq^\sharp Y^\sharp \tag{6.6}$$

which states that, for a set $A^\sharp$ to be smaller than a set $B^\sharp$, it is sufficient to ensure that any base element in $A^\sharp$ is included in some base element in $B^\sharp$. The relation $\hat{\sqsubseteq}^\sharp$ is indeed a partial order, provided that we only compare sets with non-redundant base elements, which is always the case given our definition of $\hat{\mathcal{D}}^\sharp$ (6.5) — without this technical condition, we would only have a pre-order, where distinct sets can compare equal for $\hat{\sqsubseteq}^\sharp$.

Naturally, this order implies set inclusion, i.e., $A^\sharp \hat{\sqsubseteq}^\sharp B^\sharp \implies \hat{\gamma}(A^\sharp) \subseteq \hat{\gamma}(B^\sharp)$, while the converse is not true. This order is, however, effective, and easy to implement: it reduces to testing the base ordering $\sqsubseteq^\sharp$ on every pair of base abstract elements.

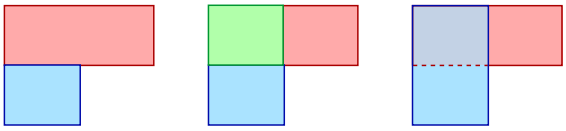
Example 6.4 (Partial order on the powerset domain)**.** *Consider the powerset completion of the interval domain on reals. Then, the following three base abstract elements:*

$$\begin{aligned} A^\sharp &= \{[0,1] \times [0,1],\ [0,2] \times [1,2]\} \\ B^\sharp &= \{[0,1] \times [0,1],\ [0,1] \times [1,2],\ [1,2] \times [1,2]\} \\ C^\sharp &= \{[0,1] \times [0,2],\ [0,2] \times [1,2]\} \end{aligned}$$

represent the exact same concrete set through $\hat{\gamma}$, although they are different. This is shown in Fig. 6.2.

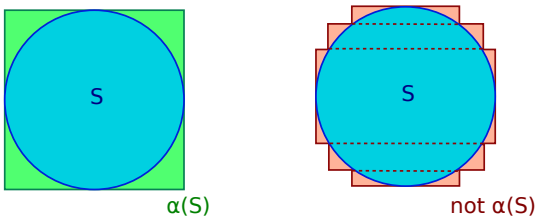
According to our order $\hat{\sqsubseteq}^\sharp$, few inclusions actually hold. We have $B \hat{\sqsubseteq}^\sharp A^\sharp \hat{\sqsubseteq}^\sharp C$, but not the converse inclusions. Hence, we cannot prove, in the abstract, that they represent the same concrete set. ♦

Fig. 6.2.(c) shows a more subtle form of redundancy, which is not ruled out by the condition $\forall X^\sharp \neq Y^\sharp \in A^\sharp \colon X^\sharp \not\sqsubseteq^\sharp Y^\sharp$ from (6.6). Here, we have two non-redundant boxes which overlap, so that we could reduce either box and still represent the same set through $\hat{\gamma}$, obtaining, e.g., Fig. 6.2.(a). Finally, although it is not redundant, Fig. 6.2.(b) is less efficient than Fig. 6.2.(a) as it uses more boxes to represent the same concrete set. We could, without loss of precision, merge two boxes in Fig. 6.2.(b) to get a more compact representation, such that the one in Fig. 6.2.(a).



(a) (b) (c)

Figure 6.2: Three decompositions with boxes of the same, non-convex shape: (a) with two boxes, (b) with three boxes, and (c) with two overlapping boxes.



(a) (b)

Figure 6.3: A disc S with (a) its best abstraction $\alpha(S)$ in the interval domain, and (b) one possible abstraction in the powerset of boxes, where no best abstraction exists.

Galois connection. Even if the base domain $\mathcal{D}^\sharp$ enjoys a Galois connection, this is not necessarily the case for $\hat{\mathcal{D}}^\sharp$. Consider, as example, a disc $S \stackrel{\text{def}}{=} \{\, (x, y) \in \mathbb{R}^2 \mid x^2 + y^2 \leq 1 \,\}$. Then, S has naturally a best abstraction in the interval domain: the box $[-1, 1] \times [-1, 1]$, as shown in Fig. 6.3.(a). However, there is no best abstraction in the powerset of boxes: Fig. 6.3.(b) shows one possible abstraction, but we could always refine it by adding more horizontal "strips" to fit tighter the curved shape of the disc. There is no limit to the number of "strips" in a finite union boxes.

Abstract operators. In $\hat{\mathcal{D}}^\sharp$, we can implement the abstract union $\hat{\cup}^\sharp$ very easily, by keeping the base abstract elements from both arguments without modification, i.e.:

$$A^\sharp \,\hat{\cup}^\sharp\, B^\sharp \stackrel{\text{def}}{=} A^\sharp \cup B^\sharp$$

The join is now an exact abstraction, which was indeed the goal of our construction.

The intersection can be abstracted by combining all possible pair-wise intersections in $\mathcal{D}^\sharp$:

$$A^\sharp \,\hat{\cap}^\sharp\, B^\sharp \stackrel{\text{def}}{=} \{\, X^\sharp \cap^\sharp Y^\sharp \mid X^\sharp \in A^\sharp,\, Y^\sharp \in B^\sharp \,\}$$

If $\cap^\sharp$ is exact in $\mathcal{D}^\sharp$, then so is $\hat{\cap}^\sharp$ in $\hat{\mathcal{D}}^\sharp$, by distributivity of $\cup$ over $\cap$ in the concrete. However, this operation is expensive as it incurs a potential quadratic blow-up in the number of base abstract elements.

Unary abstract operations can be performed element-wise on the base domain:

$$\hat{\mathbb{S}}^\sharp[\![\, s \,]\!] A^\sharp \stackrel{\text{def}}{=} \{\, \mathbb{S}^\sharp[\![\, s \,]\!] X^\sharp \mid X^\sharp \in A^\sharp \,\}$$
$$\hat{\mathbb{C}}^\sharp[\![\, c \,]\!] A^\sharp \stackrel{\text{def}}{=} \{\, \mathbb{C}^\sharp[\![\, c \,]\!] X^\sharp \mid X^\sharp \in A^\sharp \,\}$$

Note that, after all operations, it is necessary to remove redundant elements, i.e., $X^\sharp \in A^\sharp$ such that $\exists Y^\sharp \neq X^\sharp \in A^\sharp \colon X^\sharp \sqsubseteq^\sharp Y^\sharp$, in order to stay in $\hat{\mathcal{D}}^\sharp$.

Simplification. The additional expressiveness of $\hat{\mathcal{D}}^\sharp$ comes at a price in efficiency. In particular, the number of base abstract elements composing abstractions in $\hat{\mathcal{D}}^\sharp$ tend to grow large during an analysis through the computation of abstraction unions $\hat{\cup}^\sharp$ and intersections $\hat{\cap}^\sharp$, even if we take care to remove redundancies. It is thus useful to reduce the size of elements when they grow too large. One possibility is to replace two or more elements with their abstract join in $\mathcal{D}^\sharp$. For instance, when the size $|A^\sharp|$ exceeds a certain threshold, we can apply the following collapsing operation that returns an element of $\hat{\mathcal{D}}^\sharp$ reduced to a single base element, their join in $\mathcal{D}^\sharp$:

$$\mathit{collapse}(A^\sharp) \stackrel{\text{def}}{=} \{ \cup^\sharp \{X^\sharp \in A^\sharp\} \} \tag{6.7}$$

Naturally, this operator may lose a lot of precision. Deciding more precisely which base elements to merge, and when, is a difficult problem. Some domains, such as the interval or the polyhedra domain, feature algorithms to detect whether the abstract join of a set of base elements will result in a loss of precision, which can be useful to drive the collapse operation. We refer to the work of Bagnara et al. [2010] for more information on this subject.

Widening. $\hat{\mathcal{D}}^\sharp$ has infinite increasing chains, firstly, because the number of base elements is unbounded and, additionally, in case $\mathcal{D}^\sharp$ has infinite increasing chains. A simple solution to enforce convergence is to use the widening $\triangledown$ from $\mathcal{D}^\sharp$ after ensuring that our abstract elements contain only one base element, using the collapse operation (6.7):

$$A^\sharp \mathbin{\hat{\triangledown}} B^\sharp \stackrel{\text{def}}{=} \{ X^\sharp \triangledown Y^\sharp \mid \mathit{collapse}(A^\sharp) = \{X^\sharp\},\ \mathit{collapse}(B^\sharp) = \{Y^\sharp\} \}$$

While this widening guarantees the termination of the iterates, it looses much precision and prevents the inference of loop invariants that actually contain disjunctions — although disjunctions can appear freely during the analysis of the body of the loop. A slight improvement can be achieved using delayed widening and unrolling techniques from Sect. 4.7. More advanced widening techniques have been studied by Bagnara et al. [2004].

Motivating example revisited. We come back to the motivating example from Sect. 6.3.1 and perform an analysis in the powerset completion of the interval domain:

1. At line 3, before the conditional, we have a single interval abstract element $\{[10, 20] \times [0, 1]\}$, denoting the fact that $X \in [10, 20]$ and $Y \in [0, 1]$.

2. At the end of the **then** branch of line 3, we also get a single abstract element: $\{[-20, -10] \times [1, 1]\}$, due to the filter $Y \geq 1$ and the negation of X.

3. At the end of the implicit **else** branch, we have, similarly, $\{[10, 20] \times [0, 0]\}$.

4. At the end of the conditional, we take the join of these two branches, which gives us a state with two interval abstract elements: $\{[-20,-10] \times [1,1], [10,20] \times [0,0]\}$.

5. The assertion at line 4 is applied independently on the two interval abstract elements, and both pass the test $X \neq 0$ as neither interval for X contains 0.

Hence, while the plain interval domain is not precise enough to analyze our example, the powerset completion of the interval domain is. The shape of the invariant at line 4 also illustrates our claim that, even if the underlying domain $\mathcal{D}^\sharp$ is non-relational, its powerset completion $\hat{\mathcal{D}}^\sharp$ is relational: indeed, it expresses that $X \in [10,20]$ when $Y = 0$, and $X \in [-20,-10]$ when $Y = 1$, i.e., a relation between X and Y.

Remark 6.2 (Non-relational domains and powerset completion)**.** *When focusing on non-relational domains as basis $\mathcal{D}^\sharp$, we could alternatively construct the powerset at the level of a value domain $\mathcal{B}^\sharp$ obeying Def. 4.1. Note, however, that when applying the powerset completion before the non-relational construction of Def. 4.1, we get a drastically different result than if we first apply the non-relational construction, and then the powerset completion.*

Consider, for instance, the case of the interval domain. In the first case, we can represent Cartesian products of the form $D_1 \times \cdots \times D_{|\mathbb{V}|}$, where each D_i is a union of intervals. In the second case, we can represent arbitrary unions of finitely many boxes in $\mathbb{I}^{|\mathbb{V}|}$. This domain is much more expressive. Indeed, any element expressible as a Cartesian product of union of intervals can be expressed, albeit less compactly, as a union of boxes, by distributivity — for instance: $([1,2] \cup [5,6]) \times ([0,1] \cup [5,6]) = ([1,2] \times [0,1]) \cup ([1,2] \times [5,6]) \cup ([5,6] \times [0,1]) \cup ([5,6] \times [5,6])$. In fact, while the first domain remains non-relational, according to our definition based on the Cartesian abstraction of Sect. 4.9, the second domain is actually relational. $\Diamond$

6.3.3 State Partitioning

State partitioning is an alternative to powerset completion, introduced by Cousot [1981], which provides a more structured approach to expressing disjunctions. It assumes that the concrete space $\mathcal{E}$ is first partitioned into a finite set $P \subseteq \mathcal{P}(\mathcal{E})$ of parts. Instead of handling a single abstract element, the analysis will then track one abstract element per part in the partition, which focuses on abstracting the subset of the memory states included in this part. The partition is fixed and does not change during the analysis, only the abstract element in each part does.

Representation. Assume, as before, that a base abstract domain $\mathcal{D}^\sharp$ obeying Def. 3.1 is chosen. Our domain will associate to each part in the partition an element in $\mathcal{D}^\sharp$. To simplify the presentation, we will assume that the fixed partitioning of $\mathcal{E}$ is also given as a set of abstract elements in $\mathcal{D}^\sharp$ — although it is possible to use distinct domains. Hence,

we assume we are given a partition that is a set $P^\sharp$ such that:

$$\begin{aligned} & P^\sharp \in \mathcal{P}_{\mathit{finite}}(\mathcal{D}^\sharp) \\ & \cup \{\, \gamma(X^\sharp) \mid X^\sharp \in P^\sharp \,\} = \mathcal{E} \end{aligned}$$

Note that, technically, $P^\sharp$ is a covering, not a partitioning, as we do not enforce the elements in $\{\, \gamma(X^\sharp) \mid X^\sharp \in P^\sharp \,\}$ to be pairwise disjoint. Some overlapping is sometimes unavoidable: consider for instance the case where $\mathcal{D}^\sharp$ is the interval domain; then, all the parts in $\gamma(A^\sharp)$ are closed boxes that must overlap at least at the borders in order to cover $\mathcal{E}$. Overlapping should, however, be kept minimal — for instance, we will avoid having the interior of the boxes intersect. Intuitively, in case of overlapping, any concrete element belonging to several parts will be represented in several base abstract elements, which is wasteful. We will keep using the term *partitioning* as it is standard in the field of Abstract Interpretation, but the the framework accommodates using a covering without any problem.

Our derived domain, which we denote as $\tilde{\mathcal{D}}^\sharp$, with a tilde to distinguish it from $\mathcal{D}^\sharp$, associates an abstract element to each part:

$$\tilde{\mathcal{D}}^\sharp \stackrel{\text{def}}{=} P^\sharp \to \mathcal{D}^\sharp$$

An abstract element $A^\sharp \in \tilde{\mathcal{D}}^\sharp$ represents the join of the concrete sets $\gamma(A^\sharp(X^\sharp))$ represented by the basic abstract elements $A^\sharp(X^\sharp)$ in each part $X^\sharp \in P^\sharp$:

$$\tilde{\gamma}(A^\sharp) \stackrel{\text{def}}{=} \cup \{\, \gamma(A^\sharp(X^\sharp)) \cap \gamma(X^\sharp) \mid X^\sharp \in P^\sharp \,\}$$

Note that we restrict each base element $\gamma(A^\sharp(X^\sharp))$ to the corresponding part $\gamma(X^\sharp)$, to stress on the fact that each base element only gives information about one part of $\mathcal{E}$. In general, this intersection is not useful as we can ensure that $\gamma(A^\sharp(X^\sharp)) \subseteq \gamma(X^\sharp)$ at all time, i.e., base abstract elements in a part do not bleed over other parts, but we keep it for generality, as it is useful for domains where this inclusion is not always practical to achieve — e.g., for domains that are not closed under intersection.

Example 6.5 (Partitioning with boxes.)**.** *Figure 6.4 gives an example of a non-convex concrete set represented exactly in the partitioned domain over the interval domain. We use the partition $P^\sharp \stackrel{\text{def}}{=} \{P_1, \ldots, P_5\}$ of $\mathbb{R}^2$ defined as follows:*

$$\begin{aligned} P_1 &\stackrel{\text{def}}{=} [-\infty, 0] \times [-\infty, +\infty] & P_2 &\stackrel{\text{def}}{=} [0, 10] \times [0, +\infty] \\ P_3 &\stackrel{\text{def}}{=} [0, 10] \times [-\infty, 0] & P_4 &\stackrel{\text{def}}{=} [10, +\infty] \times [0, +\infty] \\ P_5 &\stackrel{\text{def}}{=} [10, +\infty] \times [-\infty, 0] \end{aligned}$$

The colored part of the figure is represented exactly as the following abstract element, associating a box to each part in $P^\sharp$:

$$\begin{aligned} X^\sharp = [& P_1 \mapsto [-6, -5] \times [5, 6],\ P_2 \mapsto \bot^\sharp, \\ & P_3 \mapsto [9, 10] \times [-\infty, -1],\ P_4 \mapsto \bot^\sharp, \\ & P_5 \mapsto [10, 12] \times [-3, -1]] \end{aligned}$$

Note that some parts in $P^\sharp$ do not contain any point from $X^\sharp$; they are associated an empty box, $\bot^\sharp$. ♦

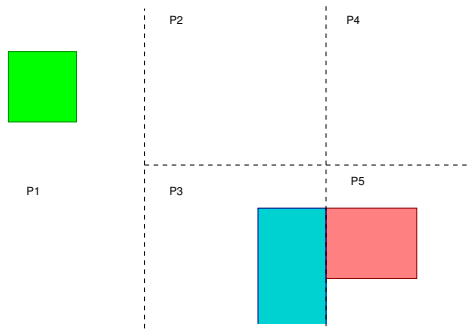


Figure 6.4: Representing a non-convex set in the partitioned interval domain. The boxes comprising the partition (delimited with dashed lines) and the abstract element (the colored boxes) are stated in Ex. 6.5.

Order structure. We can derive an order $\tilde{\sqsubseteq}^\sharp$ on $\tilde{\mathcal{D}}^\sharp$ from the order $\sqsubseteq^\sharp$ on $\mathcal{D}^\sharp$ simply, by element-wise extension:

$$A^\sharp \tilde{\sqsubseteq}^\sharp B^\sharp \overset{\text{def}}{\iff} \forall X^\sharp \in P^\sharp\colon A^\sharp(X^\sharp) \sqsubseteq^\sharp B^\sharp(X^\sharp)$$

This is naturally a partial order. Additionally, if $\sqsubseteq^\sharp$ corresponds exactly to set inclusion, i.e., if $X^\sharp \sqsubseteq^\sharp Y^\sharp \iff \gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$, then the same holds for $\tilde{\sqsubseteq}^\sharp$: $A^\sharp \tilde{\sqsubseteq}^\sharp B^\sharp \iff \tilde{\gamma}(A^\sharp) \subseteq \tilde{\gamma}(B^\sharp)$. The partitioning order is thus much more precise than the order $\hat{\sqsubseteq}^\sharp$ on powersets (6.6).

Galois connection. Another strong property states that, if $\mathcal{D}^\sharp$ features a Galois connection $(\mathcal{D}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$, then so does $\tilde{\mathcal{D}}^\sharp$, defining the abstraction function $\tilde{\alpha}$ as:

$$\tilde{\alpha}(S) \overset{\text{def}}{=} \lambda X^\sharp \in P^\sharp.\, \alpha(S \cap \gamma(X^\sharp))$$

This function abstracts independently, for each part $X^\sharp \in P^\sharp$, the portion of the concrete set S included in $\gamma(X^\sharp)$.

Abstract operators. The abstract intersection $\tilde{\cap}^\sharp$ and union $\tilde{\cup}^\sharp$ are defined element-wise:

$$A^\sharp \tilde{\cap}^\sharp B^\sharp \overset{\text{def}}{=} \lambda X^\sharp \in P^\sharp.\, A^\sharp(X^\sharp) \cap^\sharp B^\sharp(X^\sharp)$$
$$A^\sharp \tilde{\cup}^\sharp B^\sharp \overset{\text{def}}{=} \lambda X^\sharp \in P^\sharp.\, A^\sharp(X^\sharp) \cup^\sharp B^\sharp(X^\sharp)$$

Note that the abstract union is not exact in general. As it is not possible to keep several abstract elements in a single part $X^\sharp \in P^\sharp$, these elements must be merged with the base abstract union $\cup^\sharp$.

Abstracting conditions is also performed simply element-wise:

$$\tilde{\mathbb{C}}^\sharp[\![\, c \,]\!] A^\sharp \overset{\text{def}}{=} \lambda X^\sharp \in P^\sharp.\, \mathbb{C}^\sharp[\![\, c \,]\!]\, A^\sharp(X^\sharp)$$

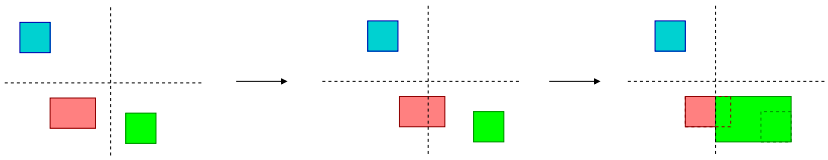


Figure 6.5: Effect of an assignment $X \leftarrow X + 1$ on a partitioned abstract element composed of three boxes: after the translation, some boxes cross parts and must be cut; some parts may end up with several abstract elements, which must be joined with $\cup^\sharp$.

as we filter out some values independently in each part.

The abstract assignment $\tilde{\mathbb{S}}^\sharp[\![s]\!]$ is more involved. It cannot be handled element-wise because the image of an abstract element from one part can escape this part and come into one or several other parts. It is thus necessary to cut the images on the boundaries of the partition. When several images from different original parts land in the same part of the partition, these must be merged with the base abstract union $\cup^\sharp$ as we can only keep a single abstract element in each part. This is illustrated graphically in Fig. 6.5. The corresponding assignment can be formalized as follows:

$$\tilde{\mathbb{S}}^\sharp[\![s]\!] A^\sharp \stackrel{\text{def}}{=} \lambda X^\sharp \in P^\sharp.\ \cup^\sharp\ \{ X^\sharp \cap^\sharp \mathbb{S}^\sharp[\![s]\!] A^\sharp(Y^\sharp) \mid Y^\sharp \in P^\sharp \}$$

which computes the abstract intersection of every part $X^\sharp \in P^\sharp$ with the image of every element from every part $A^\sharp(Y^\sharp)$, $Y^\sharp \in P^\sharp$.

Convergence acceleration. $\tilde{\mathcal{D}}^\sharp$ has infinite strictly increasing (resp. decreasing) sequences only if $\mathcal{D}^\sharp$ has. Convergence acceleration is surprisingly simple — especially compared to the case of the powerset completion — as we can define a widening and a narrowing from those in $\mathcal{D}^\sharp$ element-wise:

$$A^\sharp \tilde{\triangledown} B^\sharp \stackrel{\text{def}}{=} \lambda X^\sharp \in P^\sharp.\ A^\sharp(X^\sharp) \triangledown B^\sharp(X^\sharp)$$
$$A^\sharp \tilde{\vartriangle} B^\sharp \stackrel{\text{def}}{=} \lambda X^\sharp \in P^\sharp.\ A^\sharp(X^\sharp) \vartriangle B^\sharp(X^\sharp)$$

Motivating example revisited. We come back to the motivating example from Sect. 6.3.1 and perform an analysis in the interval domain with partitioning. We choose to partition with respect to the sign of the variable X, hence:

$$P^\sharp \stackrel{\text{def}}{=} \{X^\sharp_+, X^\sharp_-\} \text{ where}$$
$$X^\sharp_+ \stackrel{\text{def}}{=} [0, +\infty] \times \mathbb{Z}$$
$$X^\sharp_- \stackrel{\text{def}}{=} [-\infty, 0] \times \mathbb{Z}$$

where the first component of a Cartesian product is the interval of variable X, and the second one is the interval of variable Y.

1. At line 3, before the conditional, our abstract element is $[X^\sharp_+ \mapsto [10, 20] \times [0, 1],\ X^\sharp_- \mapsto \bot^\sharp]$.

2. At the end of the **then** branch at line 3, we get $[X_+^\sharp \mapsto \bot^\sharp,\ X_-^\sharp \mapsto [-20,-10]\times[1,1]]$, as the sign of X has changed.

3. At the end of the implicit **else** branch, we have, similarly, $[X_+^\sharp \mapsto [10,20] \times [0,0],\ X_-^\sharp \mapsto \bot^\sharp]$.

4. At the end of the conditional, we take the join of these two branches, which gives us $[X_+^\sharp \mapsto [10,20] \times [0,0],\ X_-^\sharp \mapsto [-20,-10] \times [1,1]]$.

5. The assertion at line 4 is applied independently on the two parts of the partition. As neither satisfies $X = 0$, the assertion is satisfied.

We are, again, able to prove that the programs ends with $X \neq 0$.

Decision tree domains. When the partitioning grows large, representing abstract elements as maps $P^\sharp \to \mathcal{D}^\sharp$ may be costly. A better representation can be constructed by exploiting the structure of the partitioning $P^\sharp$.

Consider, as concrete example, partitioning with respect to boolean variables. We assume that a subset $\mathbb{V}_\mathcal{B} \subseteq \mathbb{V}$ of the variables are boolean, i.e., can only have value 0 or 1. Assuming that the boolean variables behave in a non-linear way, we wish to partition the memory states with respect to them: we will use a distinct polyhedron to represent the memory states for each possible valuation of $\mathbb{V}_\mathcal{B}$. This would lead to a large number of parts, $2^{|\mathbb{V}_\mathcal{B}|}$, in the partition, which is prohibitive. To address this problem, we can exploit the *Binary Decision Diagram* (BDD) structure, introduced by Bryant [1986] to represent boolean functions, and adapt it to represent our partitioned state more concisely. A BDD is a binary tree where each level corresponds to a different variable. A node at a level has two sub-trees, one corresponding to states where the corresponding variable is 0, and the other where the variable is 1. We represent an abstract element in $\tilde{\mathcal{D}}^\sharp$ as a BDD where the leaves are abstract elements in $\mathcal{D}^\sharp$: the sequence of variable values in a path from the root to a leaf corresponds to a boolean valuation, i.e., one part of our partition, and the abstract value at the leaf corresponds to the base abstract element in this part — we can omit the boolean variables in the base abstract element as their value is completely determined by the path. The efficiency of BDD comes from the opportunity to share equal sub-trees, in case some numeric invariants are independent from the value of some boolean variables.

Figure 6.6 presents an example, where a numeric abstraction over two variables, expressed using polyhedra, is partitioned with respect to the value of boolean variables A, B, and C. When $A = 1$, the invariant is independent from the value of B, hence, the node at the level of B is omitted in this sub-tree. The case $(A = 1) \wedge (C = 0)$ is moreover unfeasible, hence, the corresponding leaf is set to $\bot^\sharp$. Finally, when $A = 0$, the cases $(B = 0) \wedge (C = 1)$ and $(B = 1) \wedge (C = 0)$ are identical, hence, they share the same sub-tree (here, a single leaf). As a consequence, this representation uses only four polyhedra, while an exhaustive partitioning map would use eight. Note that, in the worst case, we still need $2^{|\mathbb{V}_\mathcal{B}|}$ polyhedra, but experimental evidence shows that sharing occurs in practice. This

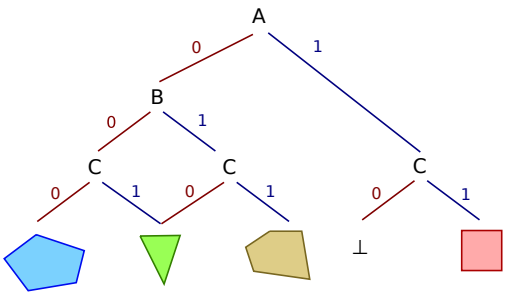


Figure 6.6: A binary decision diagram presenting numeric abstractions (at the leaves) expressed as polyhedra, and partitioned by the value of boolean variables (A, B, and C as internal nodes).

technique has been used effectively, for instance, in the Astrée analyzer [Bertrane et al., 2010]. It is further discussed by Schrammel and Jeannet [2011].

The concept of *decision tree* can be generalized to more complex settings. We can, for instance, use a $n-$ary tree allowing more values than 0 and 1, to partition with respect to enumerated types. We can also partition with respect to unbounded variables, or variables with large or continuous ranges by using intervals instead of values on arcs between a node and a sub-tree. This is implemented, for instance, in segmented decision trees by Cousot et al. [2010]. Decision trees can also feature more general constraints in the nodes, as in the domain by Urban and Miné [2014]: instead of variables, levels correspond to affine expressions, and the two sub-trees correspond to splitting the memory space into the states that satisfy the constraint, and those that do not.

Comparison with the powerset completion. Although both powerset completion and partitioning are based on the common idea of using several base abstract elements to represent a disjunction of properties, they have very different characteristics. Partitioning is very attractive as it has a strong ordering relation. Moreover, properties from the base domain, such as Galois connections, and operators, such as widening and narrowing, translate directly to the partitioned domain; on the contrary, the powerset completion has a much weaker ordering relation and hard to design widenings. Partitioning has a predictable cost, as the number of base abstract elements is fixed beforehand, while the powerset completion must rely on heuristics to limit its cost.

The drawbacks of partitioning are that the join is not exact, and that the domain relies on an externally-given partitioning. A too fine partitioning results in useless computations, while a too coarse partitioning results in a loss of precision, in particular for joins, where we may not exploit the benefit of representing several base abstract elements. The partition is generally defined by heuristics during a pre-analysis, such as the pre-analysis described in Bertrane et al. [2010] to partition memory states according to the possible valuations of boolean variables.

Several extensions of the basic partitioning scheme we presented have also been pro-

posed. The *reduced cardinal power*, introduced by Cousot and Cousot [1979a] as an earlier form of partitioning, uses different abstract domains $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ to represent, respectively, the partitioning domain and the partitioned domain, i.e., we get $\hat{\mathcal{D}}^\sharp \stackrel{\text{def}}{=} \mathcal{D}_1^\sharp \to \mathcal{D}_2^\sharp$. Bourdoncle [1992] proposes *dynamic partitioning* techniques, where the partition is not fixed *a priori*, but evolves during the analysis. This solves the problem of choosing a partitioning and naturally constructs well-adapted partitionings that are based on semantic information. A specific widening technique is introduced in order to terminate by enforcing the stabilization of both the partition and the basic elements in each part of the partition.

6.3.4 Path Partitioning

The last disjunctive domain we present is an alternative to state partitioning: we keep partitioning but, instead or relying on a criterion related to the memory state, we rely on the control-flow of the program to distinguish relevant parts of program executions. Observe that the main loss of precision addressed by disjunctive domains is the loss due to inexact abstract unions $\cup^\sharp$, and that joins are applied systematically to merge the branches of a conditional. Hence, a natural idea is to refrain form performing this join, and continue the analysis with both abstract elements. This method, called *path partitioning*, thus achieves a path-sensitive analysis.

Note that the number of paths in a program is generally unbounded, due to loops. Symbolic execution, introduced by King [1976], faces the same challenge, and resorts to a partial, thus unsound, exploration of the program behaviors. Our solution is to only partition with respect to the last occurrence of each conditional encountered during the program execution, so that the partitioning space remains finite. We thus achieve only partial path sensitivity, and still resort to abstract joins and widenings to ensure that our analysis terminates and remains sound. To sum up, instead of avoiding joins completely, we delay them. Nevertheless, this often results in an increase in precision: recall that, when discussing the non-distributivity of abstract domains, in Sect. 2.1.6, we presented an example where a late join in the interval domain, $(A^\sharp \cap^\sharp B^\sharp) \cup^\sharp (A^\sharp \cap^\sharp C^\sharp)$, gives a better result than an earlier one, $A^\sharp \cap^\sharp (B^\sharp \cup^\sharp C^\sharp)$.

Representation. In order to distinguish the conditionals occurring in the program, we assume that the syntax is enriched with control locations, as we used in the equational-style semantics from in Fig. 3.9. We denote by $\mathcal{C} \subseteq \mathcal{L}$ the set of control points denoting conditionals, i.e., in "$^{\ell 1}$ **if** c **then** $^{\ell 2}\, s_1\, ^{\ell 3}$ **else** $^{\ell 4}\, s_2\, ^{\ell 5}$ **endif** $^{\ell 6}$", we assume that only $\ell 1 \in \mathcal{C}$.

When computing an invariant at some program point $\ell \in \mathcal{L}$, we look back at the history of the computation we performed to get to this point and, for each conditional in $\mathcal{C}$, we associate a value:

- true if the latest evaluation of the conditional resulted in the **then** branch to be taken;

- false if the latest evaluation of the conditional resulted in the **else** branch to be taken;
- $\bot$ if the conditional has never been encountered.

Note that we go one step further than a flow-sensitive analysis, as we remember which branch we took even after exiting the conditional. For instance, in the program "**if** $X \leq 0$ **then** $X \leftarrow X + 1$ **else** $X \leftarrow X - 1$ **endif**; $Y \leftarrow X$", we will distinguish the evaluation of $Y \leftarrow X$ for executions that followed the **then** branch and incremented X, and for executions that followed the **else** branch and decremented X. We will abstract separately the memory states in these two cases, using two distinct abstract elements, even though the execution is past the merging point of the conditional.

Formally, the *abstract history of execution* $\mathcal{H}$ is defined as:

$$\mathcal{H} \stackrel{\text{def}}{=} \mathcal{C} \to \{\text{true}, \text{false}, \bot\}$$

and the partitioned abstract domain $\breve{\mathcal{D}}^\sharp$, constructed on top of the base abstract domain $\mathcal{D}^\sharp$, is defined as:

$$\breve{\mathcal{D}}^\sharp \stackrel{\text{def}}{=} \mathcal{H} \to \mathcal{D}^\sharp$$

with concretization:

$$\breve{\gamma}(A^\sharp) \stackrel{\text{def}}{=} \cup \{ A^\sharp(h) \mid h \in \mathcal{H} \}$$

As $\mathcal{C}$, and thus $\mathcal{H}$, are finite, an element in $\breve{\mathcal{D}}^\sharp$ can be effectively represented in memory.

Order structure. We simply use the element-wise ordering :

$$A^\sharp \breve{\sqsubseteq}^\sharp B^\sharp \stackrel{\text{def}}{\iff} \forall h \in \mathcal{H}\colon A^\sharp(h) \sqsubseteq^\sharp B^\sharp(h)$$

which is naturally a partial order.

It is not possible to provide a meaningful abstraction function α because our concrete domain $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E})$ only contains memory states and has no information about the control-flow; knowing the past control-flow is needed to decide in which element of $\mathcal{H}$ each memory element must go. We are crippled by our collecting semantics that only discusses about reachability and not history, which we chose for simplicity in this tutorial. When starting from a concrete collecting semantics of execution traces, instead of states, as done for instance by Mauborgne and Rival [2005], we would be able to provide an abstraction function and a Galois connection — provided $\mathcal{D}^\sharp$ has one. Here, we continue without an abstraction function α, leveraging the concretization-only framework of Abstract Interpretation.

Abstract operations. As in the state partitioning domain of Sect. 6.3.3, the join $\breve{\cup}^\sharp$, intersection $\breve{\cap}^\sharp$, widening $\breve{\nabla}$, narrowing $\breve{\Delta}$, and condition $\breve{\mathbb{C}}^\sharp[\![c]\!]$ are defined point-wise. Unlike state partitioning, however, the assignment $\breve{\mathbb{S}}^\sharp[\![X \leftarrow e]\!]$ is also defined point-wise. Indeed, while an assignment could switch a memory state from one part to another part

of a state partitioning, it has no influence on which part of the path partitioning we are in. However, executing a conditional "**if** $\cdots$ **then** $\cdots$ **else** $\cdots$ **endif**" does. For the first time in our design of abstract domains, we need to define a non-standard interpretation for a non-atomic statement — recall that all previous abstract domains and domain combiners handled conditionals by induction on the syntax, mimicking the concrete semantics.

In this presentation of path partitioning, we focus on the case of an abstract interpreter in denotational form, as described in Sect. 3.3, but the method can be applied as well to an equation-based analysis. Recall, from Fig. 3.10, the regular handling of conditionals in $\mathcal{D}^\sharp$:

$$\begin{aligned}&\mathbb{S}^\sharp[\![\,\textbf{if}\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{endif}\,]\!]R^\sharp \stackrel{\text{def}}{=}\\ &\quad \mathbb{S}^\sharp[\![\,s_1\,]\!](\mathbb{C}^\sharp[\![\,c\,]\!]R^\sharp) \cup^\sharp \mathbb{S}^\sharp[\![\,s_2\,]\!](\mathbb{C}^\sharp[\![\,\neg c\,]\!]R^\sharp)\end{aligned}$$

In our partitioned domain, a condition at location $\ell 1$ is handled as follows:

$$\begin{aligned}&\check{\mathbb{S}}^\sharp[\![\,{}^{\ell 1}\,\textbf{if}\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{endif}\,]\!]R^\sharp \stackrel{\text{def}}{=}\\ &\quad \check{\mathbb{S}}^\sharp[\![\,s_1\,]\!](\check{\mathbb{C}}^\sharp[\![\,c\,]\!]R^\sharp_{\text{true}}) \check{\cup}^\sharp \check{\mathbb{S}}^\sharp[\![\,s_2\,]\!](\check{\mathbb{C}}^\sharp[\![\,\neg c\,]\!]R^\sharp_{\text{false}})\\ &\text{where}\\ &R^\sharp_v \stackrel{\text{def}}{=} \lambda h.\ \begin{cases}\cup^\sharp\ \{\, R^\sharp(h[\ell 1 \mapsto t]) \mid t \in \{\text{true}, \text{false}, \bot\}\,\} & \text{if } h(\ell 1) = v\\ \bot^\sharp & \text{otherwise}\end{cases}\end{aligned}$$

where $R^\sharp_{\text{true}}$ (resp. $R^\sharp_{\text{false}}$) construct the input state for the **then** branch (resp. the **else** branch) in two steps:

- we first collapse together all the abstract states associated to histories h that only differ in $h(\ell 1)$, hence the join $\cup^\sharp\ \{\, R^\sharp(h[\ell 1 \mapsto t]) \mid t \in \{\text{true}, \text{false}, \bot\}\,\}$; this is necessary because we can only remember the latest branch of a conditional and, when encountering a new branch for that conditional, we cannot keep the old branches in distinct parts anymore;

- secondly, any history h such that $h(\ell 1) \neq$ true (resp. false) is set to $\bot^\sharp$, to state that the abstract memory state only lives in parts where the branch is taken.

The two branches are merged, at the end of the conditional, with a $\check{\cup}^\sharp$ join. Note, however, that the states from the **then** branch have non$-\bot^\sharp$ memory states only for histories h where $h(\ell 1) =$ true, and the states from the **then** branch have non$-\bot^\sharp$ memory states only for histories where $h(\ell 1) =$ false; hence, the join $\check{\cup}^\sharp$ will never actually merge with $\cup^\sharp$ two non$-\bot^\sharp$ memory states, and there will be no loss of precision due to this join — the loss of precision comes from the necessary collapse at the beginning of a conditional in step 1.

Motivating example revisited. We come back to the motivating example from Sect. 6.3.1 and perform an analysis in the interval domain with path partitioning. The analysis proceeds as follows:

- At program point 3, before the conditional is executed, we have a unique non$-\bot^\sharp$ part, and the abstract state is, omitting histories containing a $\bot^\sharp$ memory state, $[[\ell 1 \mapsto \bot] \mapsto [10, 20] \times [0, 1]]$.
- The **then** branch ends with the state $[[\ell 1 \mapsto \text{true}] \mapsto [-20, -10] \times [1, 1]]$.
- The **else** branch ends with the state $[[\ell 1 \mapsto \text{false}] \mapsto [10, 20] \times [0, 0]]$.
- At program point 4, after the merge at the end of the conditional, we have the juxtaposition of both previous states: $[[\ell 1 \mapsto \text{true}] \mapsto [-20, -10] \times [1, 1],\ [\ell 1 \mapsto \text{false}] \mapsto [10, 20] \times [0, 0]]$.
- As before, the assertion is tested on each basic abstract element and, as neither contains $X = 0$, the assertion is proved true.

Once again, we are able to prove that the programs ends with $X \neq 0$.

Comparison with powerset completion and state partitioning. Path partitioning is, as state partitioning, a partitioning technique, and it has the same benefits: the domain is well structured, with a strong notion of order, and easy to design abstract operators by point-wise extension, including convergence acceleration operators, which were difficult to design for powerset completions. The difference between state and path partitioning is the partitioning criterion: whether based on properties of the memory states, or based on the control flow of the program. On our motivating example, state partitioning required more input from the user to decide a good partitioning — the sign of X — while path partitioning flowed naturally – partitioning with respect to the conditional. For larger programs, however, it may not be possible to partition with respect to all conditionals; we may be forced to collapse parts at regular intervals — such as at the end of each function — to avoid an explosion in the number of parts, making path partitioning more adapted to local reasoning on the program rather than global reasoning. On the other hand, state partitioning can be useful to track the relationship between a global boolean and some numeric variables for the duration of the analysis, but it will not scale up to many variables. It may be more adapted than path partitioning if important control information is encoded into boolean or enumerated variables — for instance, for programs that implement finite state machines.

In practice, an effective static analyzer will not restrict itself to one disjunctive method, but will use all the three methods we presented here, choosing the most relevant one for each context. They can, also, be combined with packing techniques, as seen in Sect 6.2.5, to limit the scope of disjunctive combiners to a handful of variables at a time. For a practical example, we refer the reader to Bertrane et al. [2010] on the design of such an heterogeneous construction in the Astrée analyzer.

Loop partitioning. In our presentation, we focused on the loss of precision caused by joins in conditionals. However, loops also feature joins that can cause a significant loss

of precision. We already addressed this problem through loop unrolling, in Sect. 4.7.5. As it computes the effect of the first few iterations separately from the effect of the remaining iterations, loop unrolling has some similarities with path partitioning. However, loop unrolling required us to compute the merge of the cases — collapsing all the parts in the partition — at the end of the loop, in order to continue the analysis after the loop with a single abstract state. Path partitioning of loops can go one step further and keep these abstract states unmerged even after the loop ends. In practice, we can partition the program following the loop with respect to the number of iterations spent within the loop. This can be useful in case this number of iterations plays an important role after the loop.

Trace partitioning. The path partitioning we presented is a specific instance of a more general method, called *trace partitioning*, developed notably by Mauborgne and Rival [2005]. In trace partitioning, the concrete collecting semantics manipulates sequences of memory states annotated with control locations that remember precisely every single step of execution since the beginning of the program. We can view path partitioning as an abstraction that only remembers, for those traces, in addition to the last memory state, some of the key control points encountered during the execution, and partition the former with respect to the later. But trace partitioning is more general as it also allows partitioning criteria to take into account the values of the variables at some arbitrary point in the execution, not necessarily the last, i.e., current, memory state. We can, for instance, analyze the body of a function maintaining a separate abstract memory state for each possible value of some argument at the beginning of the function.

6.4 Summary

This chapter presented briefly the lattice of abstractions, a construction of theoretical interest that can be exploited to derive various domains from a semantic perspective, although we only applied it to the derivation of the reduced product. We spent more effort designing effective constructions of the reduced product. These constructions are parsimonious, as they reuse all the algorithms defined on the domains we combine, and only require a few additional operators, to implement the communication between the domains. As always in Abstract Interpretation, there is a wide spectrum of possible operators to implement the reduction: in some cases, we can define semantically the optimal reduction, but we can also settle for very partial reductions, obeying light soundness conditions. Another contribution of the chapter is the definition of the packing mechanism, which allows trading precision for efficiency in relational domains, by restricting the inferred relations to selected packs of variables. Finally, we presented a set of constructions to enrich an abstract domain with disjunctions. We presented three methods: powerset completion, state partitioning, and path partitioning. They rely on using several abstract elements from a base domain to represent a union symbolically, without loss of precision, and they differ in the way the sets of abstract elements are managed. Each construction has its pros and cons, and it is possible to combine them.

The domain operators we presented here — reduced product, packing, powerset completion, partitioning — play a major role in the practical design of robust and flexible static analyzers. For instance, the Astrée analyzer, described by Bertrane et al. [2010], employs a reduced product of a large set of abstract domains, including the interval, congruence, and octagon domains we presented. The analyzer can be improved by simply adding a new domain to the reduced product, in order to infer new kinds of invariants without changing the structure of the analyzer nor modifying existing abstractions. The available domains can then be selectively enabled, or disabled to improve the efficiency in case they are not needed. It also features packing and disjunctive completions that can be selectively tuned to achieve a cost versus precision sweet spot. Domain operators encourage the modular construction of static analyzers, and thus the scalability of their design.

6.5 Bibliographic Notes

The complete lattice of abstractions is present since the beginning of Abstract Interpretation by Cousot and Cousot [1979a]. It presents the semantic definition of the reduced product, while the algorithmic aspects are studied by Granger [1992]. A practical study of large-scale reduced products in the Astrée analyzer is presented by Cousot et al. [2006]. We refer to Cortesi et al. [2013] for a recent survey of product operators in Abstract Interpretation. The packing technique is described by Miné [2006a] and later extended by Bouaziz [2012].

Based on the complete lattice of abstractions, Giacobazzi and Ranzato [1997] introduce the notions of domain refinement and its inverse, domain compression, which respectively add and remove elements from an abstract domain. An application of compression, proposed by Giacobazzi and Ranzato [1998], is to remove from a domain abstract elements that are useless if the domain serves as a base in a disjunctive completion construction. Another application, by Cortesi et al. [1997], is domain complementation, which acts as the inverse of the product: it allows "dividing" a domain by another domain to decompose it as a reduced product. Applications of domain refinement include the construction of relational domains from non-relational ones by Giacobazzi and Scozzari [1998]. Finally, an abstract domain can be minimally refined or simplified to achieve completeness, as shown by Giacobazzi et al. [2000].

The complete lattice of abstractions by Cousot and Cousot [1979a] also introduced the reduced cardinal power of domains, which lead to state partitioning in Cousot [1981]. Decision-tree data-structures for partitioning take their roots in binary decision diagrams introduced by Bryant [1986]. A simple use in the Astrée analyzer is described by Bertrane et al. [2010], while Cousot et al. [2010] present the, more advanced, segmented decision tree data-structure. An application to inferring termination is also presented by Urban and Miné [2014]. Dynamic partitioning techniques are pioneered by Bourdoncle [1992] to infer function summaries with the interval domain, and later applied to the polyhedra domain by Jeannet [2003].

The powerset construction is also already present in [Cousot and Cousot, 1979a]. It is

later studied by Filé and Ranzato [1999], while Bagnara et al. [2004] study the design of widenings for the powerset of numeric domains, and Bagnara et al. [2010] propose algorithms to detect exact joins of numeric elements with application to simplifying elements in the powerset domain.

Trace partitioning, which is the basis for the path partitioning we presented, is introduced by Handjieva and Tzolovski [1998], and later developed by Mauborgne and Rival [2005]. Path-enumeration methods are classically used in data-flow analyses [Kildall, 1973] (in the "meet-over-all-paths" method), and also used in symbolic execution [King, 1976], but they are limited to finitary settings as, in the former case, the lattice of interpretation has a bounded height and, in the later case, only a finite number of paths are explored.

Chapter 7

Conclusion

7.1 Summary

In this tutorial, we have presented the basis of numeric invariant inference by Abstract Interpretation. We first designed an idealized toy-language, with only simple constructions (assignments, conditionals, loops) and numeric variables with perfect integers, rationals, or reals, in order to present formally and completely its concrete collecting semantics: the mathematical expression of the most precise invariants of every program — a well-defined but uncomputable expression. We then proposed two flavors of approximate computable static analyses, parameterized by a choice of abstraction: either as solving an abstract equation system, or by interpretation by induction on the syntax in the abstract world. We showed the main hypotheses necessary to ensure the soundness of the analysis, discussed the (optional) derivation of optimal operators, and the fixpoint acceleration techniques required to make the analysis effective in infinite-height abstract domains. We then presented some of the most widespread numeric abstractions, starting with non-relational domains: signs, constants, intervals, congruences; then, moving on to relational domains: affine equalities, affine inequalities (polyhedra), zones, octagons, templates. Each domain comes with its specific expressiveness and algorithms, and achieves some cost versus precision trade-off. Finally, we presented domain transformers that allow deriving more precise abstractions, either by combining several existing abstractions through a reduced product, or by lifting abstractions to represent exactly — or more precisely — disjunctions, through powerset completion, state partitioning, or path partitioning.

7.2 Principles

We now sum up a few key points of Abstract Interpretation and general lessons we encountered in this tutorial.

Concrete semantics. We start by defining the concrete collecting semantics: a fully formal definition of the properties of interest for all programs. As the soundness of the

analysis results are only guaranteed with respect to this semantics, it must be unambiguous and convey the intended meaning of programs clearly to the analysis user. It is expressed in rich mathematical worlds, using fixpoints of monotonic or continuous operators in CPO or complete lattices. In our case the semantics expresses the tightest invariants or, equivalently, the reachable memory states at every program point, and it is uncomputable.

Abstract domains. An abstract domain of interpretation is defined on two levels: a semantic level, stating a subset of concrete properties together with abstract versions of the semantic operators; and an algorithmic level, describing data-structures and effective implementations of the abstract operators. This second aspect must not be understated: we spent a large part of this tutorial describing the algorithmic aspects of abstract domains. An abstract domain has generally less structure than the concrete one: it only needs to be a poset, not a CPO nor a lattice (although it is in some cases), and the abstract operators need not be monotonic — which they are often not, due to widenings and reductions.

Soundness and optimality. The minimum connection we require between the abstract and the concrete is a concretization function that defines the notion of sound abstraction — in our case, a sound abstraction leads to an over-approximation of the set of memory states. On the other hand, while not necessary, Galois connections provide a stronger connection with a notion of best abstraction. In the concretization-only framework, the analysis designer must invent abstract operators and prove their soundness *a posteriori*. In the Galois connection framework, the abstract semantics can be derived systematically from the concrete semantics and the Galois connection, although this does not help us derive effective and efficient algorithms. It is possible to mix both approaches, on a per-operator basis. In all cases, the soundness of the analysis is formally established.

Expressiveness. The abstract domain must be expressive enough. It must not only be able to express the invariants at the end of the program, but also the invariants at all program points, including inductive loop invariants. Such loop invariants generally have a more complex shape than the invariants at the loop exit. Additionally, imprecisions in the abstract operators accumulate, as combining optimal operators does not give an optimal operator. This leads us to require more expressive domains than expected by the properties of interest only. For instance, the polyhedra domain might be needed even when inferring only non-relational variable bounds.

Operator design. Even after a domain has been chosen, there is a large leeway in the design of sound abstract operators, leading to a large choice of cost versus precision trade-offs, as well as practicality options. On the one hand, we are required to design sound operators for every language construction, even when the operators do not match the expressiveness of the domain. For instance, even though polyhedra can only handle natively affine assignments, we must support non-linear assignments as well. It is fortunate, then, that most operators have simple, fall-back abstract versions — such as non-deterministic

assignments, or even resorting to $\top^\sharp$. On the other hand, even when optimal operators exist semantically, they may have no efficient enough algorithm, or none at all, but it is easy to revert to approximate ones — such as using the interval assignment for non-linear assignments in polyhedra. In the end, there is hardly any reason, only drawbacks, to abandon soundness or to restrict the soundness of the analysis to an unrealistic subset of the language.

In addition to the approximation embedded in each operator, which is required by the static choice of a limited set of representable properties in the abstract, there is the need to approximate fixpoints when the domain has an infinite height. This source of approximation is more dynamic, as it is driven by the sequence of abstract states encountered during the execution of the analysis, during iterations. It worth noting that, even after the static approximation is fixed, there is room to improve the dynamic approximation by designing refined widenings and iteration strategies.

Modularity. Abstract Interpretation encourages modular designs. Firstly, program semantics are defined in a modular way, by combining a small alphabet of atomic semantic operations, hence, each such operation is abstracted independently, and it is possible to extend the analysis to new languages by adding new relevant operations. Secondly, abstract domains can be combined through reduced products or disjunctive completions. This encourages the design of highly reusable abstract domains, which focus on a single class of invariants at a time. An analyzer can then be constructed by choosing which building blocks to combine.

Design by refinement. The Abstract Interpretation framework supports a gentle design policy, where an analyzer is first designed to be sound and precise on a subset of programs of interest, while still being sound, but possibly coarser and less efficient on the rest of the programs allowed by the language. Such an analyzer can always be made more precise afterwards, if needed, either by refining the abstract operators in the current domains, or by adding new domains [Bertrane et al., 2010].

Local completeness. As the properties we infer are undecidable, a static analyzer cannot be complete: for every analyzer, there exists an infinite number of programs on which it will not give sufficiently precise results. Note, however, that, for each of the examples provided in the tutorial, we always found a combination of abstract domains, abstract operators, and fixpoint extrapolation operators that managed to analyze it precisely. This remark can be actually generalized: given a single program, there always exists an abstract domain that can infer the properties of interest. Indeed, for the sake of argument, note that it is sufficient to select, as abstract elements, the invariants that are used in a Hoare—Floyd proof of correctness of that specific program. The domain constructed that way is even finite, and does not require a widening. This theoretic domain has, naturally, no great value, as it is far too specialized.

On the other hand, a domain such as the interval domain, which has an infinite number

of elements, can find precise invariants for an infinite number of programs. This tells us that the refinement principle suggested in the previous paragraph always succeeds for a given program, and the end-result is an analyzer that is precise on an infinite family of programs, while being imprecise on an infinite family of programs. The design of specialized analyzers, such as Astrée [Bertrane et al., 2010], aims at finding a balance between the general incompleteness and the local completeness of static analysis, to construct analyzers that work well on reasonably large families of programs of interest.

7.3 Towards the Analysis of Realistic Programs

This tutorial focused on analyzing an idealized language to better illustrate the principles of Abstract Interpretation and present the core of numeric abstract domains. Actual static analyzers for real-life programming languages follow the same principles, and reuse these abstract domains. Going from the analysis of our language to the analysis of a language such as C nevertheless presents some additional challenges. To conclude this tutorial, we list succinctly some of these challenges and provide, without further explanations, some pointers to related works.

Firstly, we need to adapt our domains from abstracting a semantics on idealized numbers to a semantics on machine integers and floating-point numbers, actually used by programs [Miné, 2004, 2012, Simon and King, 2007]. Then, more complex data-structures, such as arrays and structures must be handled; abstracting them efficiently, especially when they are large or possibly unbounded, is difficult. An additional difficulty, in C, comes from the union types, which allow several data of incompatible types and bit-representations to share the same memory [Miné, 2006b]. We would also require an analysis for pointers. Pointer arithmetic in C can be reduced to integer arithmetic on offsets, which gives another use for our numeric domains. We also need to infer the set of variables a pointer may point to. In the presence of dynamic memory allocation, some abstraction of the possibly unbounded allocated memory blocks is necessary; these range from simple site-based abstractions to more complex non-uniform abstractions [Venet, 2004]. Most additional control structures found in actual languages do not pose much challenge: they can be handled either on a control-flow graph level, or through an encoding into continuations. Functions may require additional abstractions: while it is possible to inline function calls, provided that the call stack is bounded, this is not always efficient enough; a more scalable approach to, possibly unbounded, function calls (including the case of recursive functions) would employ a more modular approach [Ancourt et al., 2010].

For a practical example, the interested reader is encouraged to consult the description of the design of the Astrée static analyzer in [Bertrane et al., 2010].

Acknowledgements

This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 — MOPSA.

Bibliography

E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In *Proc. of the 16th European Symposium on Programming*, LNCS, pages 157–172. Springer, 2007.

G. Amato and F. Scozzari. The abstract domain of parallelotopes. *Electronic Notes in Theoretical Computer Science*, 287:17–28, 2012. Proceedings of the Fourth International Workshop on Numerical and Symbolic Abstract Domains, NSAD 2012.

C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3 – 16, 2010. Proceeding of the Second International Workshop on Numerical and Symbolic Abstract Domains: NSAD 2010.

R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *Static Analysis: Proceedings of the 9th International Symposium*, volume 2477 of *LNCS*, pages 213–229. Springer, 2002.

R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In *Proc. of the 5h Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2477 of *LNCS*, pages 135–148. Springer, 2004.

R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1–2):28–56, October 2005a.

R. Bagnara, E. Rodríguez-Carbonell, and E. Zaffanella. Generation of basic semi-algebraic invariants using convex polyhedra. In *Static Analysis: 12th International Symposium, SAS 2005*, pages 19–34. Springer, 2005b.

R. Bagnara, P. M. Hill, and E. Zaffanella. An improved tight closure algorithm for integer octagonal constraints. In *Verification, Model Checking and Abstract Interpretation: Proceedings of the 9th International Conference (VMCAI 2008)*, volume 4905 of *LNCS*, pages 8–21. Springer, 2008a.

R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008b.

R. Bagnara, P. M. Hill, and E. Zaffanella. Weakly-relational shapes for numeric abstractions: Improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.

R. Bagnara, P. M. Hill, and E. Zaffanella. Exact join detection for convex polyhedra and other numerical abstractions. *Computational Geometry: Theory and Applications*, 43 (5):453–473, 2010.

G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. of the Int. Conf. on Compiler Construction (CC'04)*, number 2985 in LNCS, pages 5–23. Springer, 2004.

V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *ACM PLDI'89*, pages 41–53. ACM Press, 1989.

F. Banterle and R. Giacobazzi. A fast implementation of the octagon abstract domain on graphics hardware. In *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings*, pages 315–332. Springer, 2007.

C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.

C. Bartzis and T. Bultan. Widening arithmetic automata. In *Computer Aided Verification, 16th International Conference, CAV*, volume 3114 of *LNCS*, pages 321–333. Springer, 2004.

F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revisiting hull and box consistency. In *Proc. of the 16th Int. Conf. on Logic Programming*, pages 230–244, 1999.

F. Benoy, A. King, and F. Mesnard. Computing convex hulls with a linear solver. *Theory and Practice of Logic Programming*, 5(1–2):259–271, 2005.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385 in AIAA, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), April 2010.

J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages (FnTPL)*, 2(2–3):71–190, 2015.

G. Birkhoff. Lattice theory. In *Colloquium Publications*, volume 25. Amer. Math. Soc., 3. edition, 1967.

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of the ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'03)*, pages 196–207. ACM, June 2003.

M. Bouaziz. TreeKs: A functor to make numerical abstract domains scalable. In *4th International Workshop on Numerical and Symbolic Abstract Domains (NSAD 2012)*, volume 287, pages 41–52. Elsevier, September 2012.

F. Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Funct. Program.*, 2 (4):407–423, 1992.

F. Bourdoncle. Abstract debugging of higher-order imperative languages. *SIGPLAN Not.*, 28(6):46–55, June 1993a.

F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93)*, volume 735 of *LNCS*, pages 128–141. Springer, June 1993b.

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35:677–691, 1986.

R. M. Burstall. Program proving as hand simulation with a little induction. *Information Processing*, pages 308–312, 1974.

L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *Proc. of the Sixth Asian Symp. on Programming Languages and Systems (APLAS'08)*, volume 5356 of *LNCS*, pages 3–18. Springer, December 2008.

L. Chen, A. Miné, J. Wang, and P. Cousot. Linear absolute value relation analysis. In *Proc. of the 20th European Symp. on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 156–175. Springer, March 2011.

N. V. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8 (6):282 – 293, 1968.

C. K. Chiu and J. H. M. Lee. Efficient interval linear equality solving in constraint logic programming. *Reliable Computing*, 8(2):139–174, April 2002.

E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8:244–263, 1986.

E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

A. Cortesi, G. Filé, F. Ranzato, R. Giacobazzi, and C. Palamidessi. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, January 1997.

A. Cortesi, G. Costantini, and P. Ferrara. A survey on product operators in abstract interpretation. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*, pages 325–336, 2013.

A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Computer Aided Verification: 17th International Conference, CAV 2005*, pages 462–475. Springer, 2005.

P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, September 1977. 15 p.

P. Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

P. Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press.

P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

P. Cousot. Abstracting induction by extrapolation and interpolation. In *Verification, Model Checking, and Abstract Interpretation: 16th International Conference, VMCAI 2015*, pages 19–42. Springer, 2015.

P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of the 2d Int. Symp. on Programming*, pages 106–130. Dunod, Paris, France, 1976.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, January 1977.

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conf. Rec. of the 6th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'79)*, pages 269–282. ACM Press, 1979a.

P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979b.

P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *Proc. of the Int. Workshop on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *LNCS*, pages 269–295. Springer, 1992a.

P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992b.

P. Cousot and R. Cousot. "à la Burstall" intermittent assertions induction principles for proving inevitability properties of programs. *Theoret. Comput. Sci.*, 120(1):123–155, 1993.

P. Cousot and R. Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In *Logics and Languages for Reliability and Security*, NATO Science Series III: Computer and Systems Sciences, pages 1–29. IOS Press, 2010.

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the 5th Annual ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *Proc. of the 11th Annual Asian Computing Science Conf. (ASIAN'06)*, volume 4435 of *LNCS*, pages 272–300. Springer, December 2006.

P. Cousot, R. Cousot, and L. Mauborgne. A scalable segmented decision tree abstract domain. In *Pnueli Festschrift*, volume 6200 of *LNCS*, pages 72–95. Springer, 2010.

R. Cousot. Reasoning about program invariance proof methods. Res. rep. CRIN-80-P050, Centre de Recherche en Informatique de Nancy (CRIN), Institut National Polytechnique de Lorraine, Nancy, France, July 1980.

P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. In *Proc. of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247. Springer, 2012.

E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis: 8th International Symposium, SAS 2001 Paris, France, July 16–18, 2001 Proceedings*, pages 194–212. Springer, 2001.

J. Feret. Static analysis of digital filters. In *Proc. of the 13th European Symp. on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 33–48. Springer, March 2004.

J. Feret. The arithmetic-geometric progression abstract domain. In *Proc. of the 6th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 42–58. Springer, January 2005.

G. Filé and F. Ranzato. The powerset operator on abstract interpretations. *Theoretical Computer Science*, 222(1):77–111, 1999.

R. W. Floyd. Assigning meanings to programs. In *Proc. of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–32, Providence, USA, 1967.

G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Interval analysis and machine arithmetic: Why signedness ignorance is bliss. *ACM Trans. Program. Lang. Syst.*, 37(1):1:1–1:35, January 2015.

K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain Taylor1+. In *Proc. of the 21st Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 627–633. Springer, June 2009.

R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In *Automata, Languages and Programming: 24th International Colloquium, ICALP '97*, pages 771–781. Springer, 1997.

R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32(1):177–210, 1998.

R. Giacobazzi and F. Scozzari. A logical model for relational abstract domains. *ACM Trans. Program. Lang. Syst.*, 20(5):1067–1109, September 1998.

R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, March 2000.

P. Granger. Static analysis of arithmetic congruences. *Int. Journal of Computer Mathematics*, 30:165–199, 1989.

P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proc. of the Int. Joint Conf. on Theory and Practice of Soft. Development (TAPSOFT'91)*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.

P. Granger. Improving the results of static analyses of programs by local decreasing iterations. In *Foundations of Software Technology and Theoretical Computer Science: 12th Conference*, pages 68–79. Springer, 1992.

P. Granger. Static analyses of congruence properties on rational numbers (extended abstract). In *Static Analysis: 4th International Symposium, SAS '97*, pages 278–292. Springer, 1997.

N. Halbwachs and J. Henry. When the decreasing sequence fails. In *Static Analysis: 19th International Symposium, SAS 2012*, pages 198–213. Springer, 2012.

M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis: 5th International Symposium, SAS'98*, pages 200–214. Springer, 1998.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10): 576–580, October 1969.

ISO/IEC JTC1/SC22/WG14 working group. C standard. Technical Report 1124, ISO & IEC, 2007.

B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, July 2003.

B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 661–667. Springer, June 2009.

G. Kahn. Natural semantics. Technical Report 601, INRIA, 1987.

M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

G. Kildall. A unified approach to global program optimization. In *Proc. of the 1st Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL'73)*, pages 194–206. ACM, 1973.

J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

S. C. Kleene. *Introduction to metamathematics*. Bibliotheca mathematica. North-Holland Pub. Co., 1964.

S. Lang. *Introduction to Linear Algebra*. Undergraduate Texts in Mathematics. Springer, 1997.

H. LeVerge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Science of Computer Programming*, 75(9):796–807, 2010.

A. Maréchal, D. Monniaux, and M. Périn. Scalable minimizing-operators on polyhedra via parametric linear programming. In *Static Analysis - 24th International Symposium*, pages 212–231, 2017.

I. Mastroeni. Algebraic power analysis by abstract interpretation. *Higher-Order and Symbolic Computation*, 17(4):297–345, December 2004.

L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In *Proc. of the 14th European Symp. on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 5–20. Springer, April 2005.

K. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. of the Second Symposium on Programs as Data Objects (PADO II)*, volume 2053 of *LNCS*, pages 155–172. Springer, May 2001.

A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. of the European Symp. on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 3–17. Springer, March 2004.

A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1): 31–100, 2006a.

A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM, June 2006b.

A. Miné. Abstract domains for bit-level machine integer and floating-point operations. In *Proc. of the 4th Int. Workshop on Invariant Generation (WING'12)*, number HW-MACS-TR-0097 in EPiC Series in Computing, page 16. Computer Science, School of Mathematical and Computer Science, Heriot-Watt University, UK, June 2012.

R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs N. J., USA, 1966.

M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *Proc. of the 14th European Symp. on Prog. (ESOP'05)*, volume 3444 of *LNCS*, pages 46–60. Springer, April 2005.

D. Nguyen Que. *Robust and generic abstract domain for static program analysis: The polyhedral case*. PhD thesis, École des Mines de Paris, 2010.

H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. *SIGPLAN Not.*, 47(6):229–238, June 2012.

S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. of the 11th Int. Conf. on Automated Deduction (CADE'92)*, volume 607 of *LNAI*, pages 748–752. Springer, June 1992.

G. D. Plotkin. A structural approach to operational semantics, 1981.

W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Commun. of the ACM*, 8:4–13, August 1992.

H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64 (1):54 – 75, 2007.

S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. of the 6th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 21–47. Springer, 2005.

D. Schmidt. Abstract interpretation from a topological perspective. In *Static Analysis: 16th International Symposium, SAS 2009*, pages 293–308. Springer, 2009.

P. Schrammel and B. Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *Static Analysis: 18th International Symposium, SAS 2011*, pages 233–248. Springer, 2011.

A. Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., 1986.

D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Report PRG-6, Oxford U. Computing Lab, 1971.

Y. Seladji. Finding relevant templates via the principal component analysis. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2017*, pages 483–499. Springer, 2017.

A. Simon and A. King. Analyzing string buffers in C. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, AMAST '02, pages 365–379. Springer, 2002.

A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In *Proc. of the 12th Int. Symp. on Static Analysis (SAS'05)*, volume 3672 of *LNCS*, pages 336–351. Springer, September 2005.

A. Simon and A. King. Taming the wrapping of integer arithmetic. In *Proc. of the 14th Int. Symp. on Static Analysis (SAS'07)*, volume 4634 of *LNCS*, pages 121–136. Springer, August 2007.

A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *Proc. of the 12th Int. Conf. on Logic based program synthesis and transformation (LOPSTR'02)*, volume 2664 of *LNCS*, pages 71–89. Springer, 2002.

G. Singh, M. Püschel, and M. Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 46–59. ACM, 2017.

A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.

A. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, 1949.

C. Urban and A. Miné. A decision tree abstract domain for proving conditional termination. In *Proc. of the 21st International Static Analysis Symposium (SAS'14)*, volume 8373 of *LNCS*, pages 302–318. Springer, September 2014.

A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *Proc. of the Int. Symp. on Static Analysis (SAS'04)*, number 3148 in LNCS, pages 149–164. Springer, 2004.