Abstract Constraint Programming

draft v0.1

PIERRE TALBOT, University of Luxembourg, Luxembourg
ACHIM JUNG, The University of Birmingham, United Kingdom
KAZUNORI UEDA, Department of Computer Science and Engineering, Waseda University, Japan
PETER VAN ROY, Catholic University of Louvain, Belgium

Abstract constraint reasoning is a recent field which studies constraint solvers from the perspective of abstract interpretation. The main purpose is to generalize solving techniques to a lattice-theoretic framework in order to reuse these on other domains. We propose new abstract domains capturing the essence of discrete and continuous constraint programming, search algorithms and optimization problems. Moreover, we introduce an abstract framework in which the model (variables and constraints) and the control aspects (search tree and strategies) of constraint solving are unified.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Theory of computation** → **Constraint and logic programming**; *Process calculi*; • **Software and its engineering** → **Constraint and logic languages**; **Constraints**; • **Computing methodologies** → *Concurrent programming languages*; • **Social and professional topics** → History of programming languages;

ACM Reference Format:
Pierre Talbot, Achim Jung, Kazunori Ueda, and Peter Van Roy. 2021. Abstract Constraint Programming draft v0.1. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (2021), 43 pages. https://doi.org/0000001.0000001

Contents

Abstract 1
Contents 1
1 Introduction 2
2 Background 4
2.1 First-order logic 4
2.2 Lattice theory 5
3 Abstract constraint programming 9
3.1 Concrete domain 10
3.2 Abstract domain 11
3.3 Propagate and search 13
3.4 Relationship between approximations and satisfiability 13
4 Domain of a variable 14

Authors' addresses: Pierre Talbot, pierre.talbot@uni.lu, University of Luxembourg, Maison du Nombre, Esch-sur-Alzette, L-4030, Luxembourg; Achim Jung, a.jung@cs.bham.ac.uk, The University of Birmingham, Birmingham, B15 2TT, United Kingdom; Kazunori Ueda, ueda@ueda.info.waseda.ac.jp, Department of Computer Science and Engineering, Waseda University, 3-4-1, Okubo, Shinjuku-ku, Tokyo, 169-8555, Japan; Peter Van Roy, peter.vanroy@uclouvain.be, Catholic University of Louvain, 2, Place Sainte Barbe, Louvain-la-Neuve, B-1348, Belgium.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0164-0925/2021/0-ART0 $15.00
https://doi.org/0000001.0000001

4.1 Constructions over unordered universe of discourse 14
4.2 Constructions over ordered universe of discourse 15
5 Propagation problem 18
5.1 An abstract domain for collection of variables 19
5.2 Propagation problem abstract domain 19
5.3 Compositionality of under-approximation 22
5.4 Compositionality of over-approximation 24
6 Search tree 25
6.1 Queuing strategy 26
6.2 Search tree abstract domain 27
6.3 Single solution abstract domain 29
6.4 Compositionality of over-approximation 29
6.5 Compositionality of under-approximation 30
7 Optimization problem 32
7.1 Under-approximating branch-and-bound 33
7.2 Over-approximating branch-and-bound 35
8 Discussion and related work 36
8.1 Existing abstract constraint solvers 36
8.2 Prospective abstract constraint solvers 37
8.3 Combination of abstract constraint solvers 39
8.4 Concurrent constraint programming 39
9 Conclusion 40
References 40

1 INTRODUCTION

Constraint reasoning is a large field encompassing many methods for finding a solution to a set of constraints. A constraint is a mathematical relation over variables such as $x > y$, $x \times y < z \times 9.5$, $s \in \{1, 2, 3\}$ or $s \subseteq v$. The semantics of a constraint is the set of valid functions from variables to values, called *assignments*, *e.g.*, $\{\{x \mapsto a, y \mapsto b\} \mid a > b, a, b \in \mathbb{Z}\}$ for the constraint $x > y$ over integers. The constraints may be assembled by logical connectives to form a logical formula. *Constraint satisfaction* is the problem of finding one or more assignments satisfying a logical formula. Another fundamental problem is *constraint optimization* where we seek a best possible solution according to some criterion, *e.g.*, find a solution with the smallest value of x. Constraint satisfaction and optimization may be applied to many problems from operation research such as scheduling and vehicle routing, but also from diverse fields including program verification, cryptography, mathematics, bioinformatics, and musical composition (see, *e.g.*, [RvBW06, BHvMW09]).

Abstract interpretation is a framework to statically analyze programs by over-approximating the set of values that the variables of a program can take [CC77a] (see also [Min17] for an introduction). This set of values is represented by a *concrete domain*, a mathematical structure which is not necessarily extensionally representable in a machine because it is infinite or very large. A key idea of abstract interpretation is to approximate this concrete domain with an *abstract domain* which has a practical representation. For instance, the set of real numbers $\{x \in \mathbb{R} \mid 1.1 \leq x \leq 2.1\}$, which is not representable because it is infinite, can be over-approximated by the floating point interval $[1.0..2.25]$. Alternatively, the concrete set can also be under-approximated by the interval $[1.25..2.0]$.

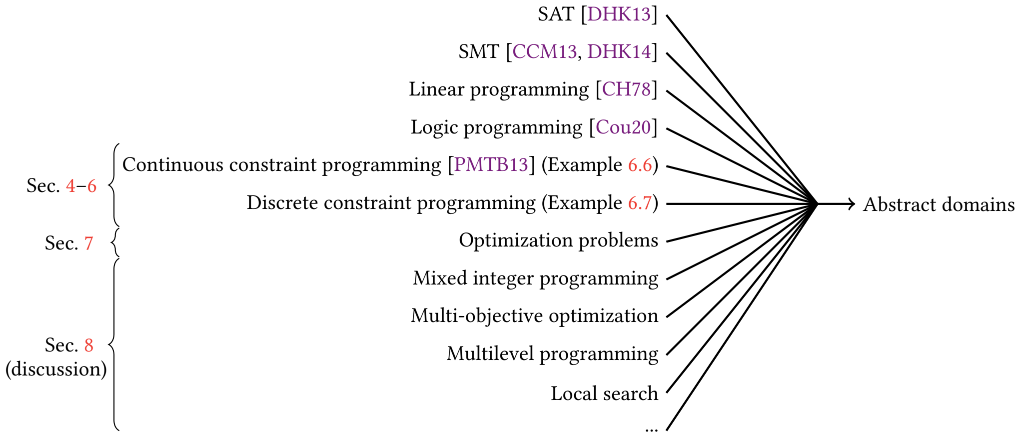


Fig. 1. Abstract constraint solvers

Abstract constraint reasoning is a recent field which studies constraint solvers from the perspective of abstract interpretation. The data structure of a constraint solver is conceptualized as a lattice, and the constraint solving techniques are formalized as extensive functions ($f(x) \geq x$) on this lattice. The lattice structure and its extensive functions together form an *abstract domain* of the constraint solver. We give in Figure 1 some references to existing abstract constraint solvers, as well as constraint solvers not yet studied with abstract interpretation. There are at least three advantages to be obtained from generalizing solving techniques to a lattice-theoretic framework:

(1) It imports the techniques from one field to the another.
(2) It allows us to combine the different approaches to obtain more efficient hybrid methods.
(3) It promises to give us a better understanding of the techniques of different fields in a unified theory.

In [DHK13], it is shown how the conflict driven clause learning algorithm of SAT solvers can be generalized and transfered to other abstract domains. Hybrid approaches can be easily designed thanks to the various combinations of abstract domains, namely *products*, available in abstract interpretation. For instance, the Nelson-Oppen theory combination procedure was shown to be a product of domains [CCM13]. We also studied different generic products in previous work [TCMT19, TMT20]. We further discuss existing and prospective abstract constraint solvers in Section 8.

In this paper, we contribute new abstract domains capturing the essence of discrete and continuous constraint programming, search algorithms and optimization problems. In particular, our work extends [PMTB13]—dedicated to continuous constraint programming—to a unified presentation of both discrete and continuous solving algorithms. We put an emphasis on filling the gap between practical implementations and theory. Indeed, the theoretical abstract framework presented in this paper is informed by our previous experience in implementing abstract domains for constraint reasoning[1] [TCMT19, TMT20].

Contributions and roadmap. In order to keep this paper self-contained, we start in Section 2 with background material on first-order logic and lattice theory. Section 3 introduces the concepts of concrete and abstract domains and their relations to satisfiability. Our definition of abstract domain

[1]https://github.com/ptal/AbSolute

refines those of previous work [PMTB13, TMT20]. Our first contribution is to unify both under- and over-approximating domains in a single framework.

A constraint solver is the combination of many components including the domain of the variables, the constraints and the search tree and search strategies. There is usually a distinction between the model (variables and constraints) and the control (search tree and search strategies). This distinction was already made in the early days of logic programming by the equation "logic + control = algorithm" [Kow79]. Our second contribution is to show that all these components can be represented as abstract domains with the same interface. The model and control are analogous parts of our framework. Moreover, this is the first time the search component of a constraint solver is formalized as an abstract domain. We present this contribution incrementally. We start by introducing several domains of variable in Section 4 including integers, floating-point numbers and sets. Section 5 discusses abstract domains for constraints which are built on domains of variable. The control part is discussed in Section 6 where we construct the search tree abstract domain. Along the way, we also survey and connect many existing works to our abstract framework.

In Section 7, we present an abstract optimization procedure on under- and over-approximating abstract domains. The third contribution is to lift two branch-and-bound (BAB) optimization algorithms to a lattice-theoretic framework. As we discuss in Section 8.2, our generalization encompasses the branch-and-bound algorithms used in multi-objective optimization and mixed integer programming.

Our fourth contribution is to establish theorems for combining under- and over-approximating abstract domains. In Sections 5.3 and 5.4, we study general conditions to correctly combine functions on abstract domains while preserving under- and over-approximations. In Sections 6.4 and 6.5, we extend these results to non-monotone functions over the search tree abstract domain. As our results are established in a very general lattice-theoretic framework, they can be reused to prove the correctness of new abstract constraint solvers. In particular, we establish the properties of the abstract optimization procedures in Section 7 thanks to these results.

Previously, we proposed a search strategy language [Tal19] based on lattice theory, which is incomplete due to the lack of treatment of under- and over-approximation properties. Our long term goal is to propose a programming language for computing with abstract domains, and automatically proving under- and over-approximating properties. This paper is a first step towards this goal. We discuss this aspect in Section 8.4.

2 BACKGROUND

2.1 First-order logic

We fix the terminology surrounding the syntax of first-order logic (FOL) used throughout this paper. A first-order signature S is a triple $\langle X, F, P\rangle$ where X is the set of variables, F the set of function symbols and P the set of predicate symbols. We write $x, y, \ldots \in X$ to denote variables, $f, g, \ldots \in F$ for function symbols and $p, q, \ldots \in P$ for predicate symbols. Each function and predicate symbol has an arity $n \geq 0$. A function symbol (resp. predicate symbol) with an arity of 0 is called a constant (resp. Boolean atom). A term is either a variable or a function whose arguments are terms. An atom (atomic formula) is a predicate $p(t_1, \ldots, t_n)$ whose arguments $t_1, \ldots, t_n$ are terms. A literal is either an atom a or its negation $\neg a$. A formula is built by the usual existential quantifier $\exists$, the universal quantifier $\forall$ and the (non-minimal) set of Boolean connectives $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$. We say that a formula φ (or term t) is closed (or ground) if there is no variable free in φ (or t). We denote the set of free variables of φ by $FV(\varphi)$. A sentence is a closed formula. A theory is a set of sentences based on a signature S. A clause is a disjunction of literals $\ell_1 \vee \ldots \vee \ell_n$. A formula in conjunctive

normal form (CNF) is a conjunction of clauses. The substitution $\varphi[x \mapsto t]$ denotes the formula φ in which all occurrences of a variable x free in φ have been replaced by the term t.

We consider *model-theoretic semantics*, pioneered by Tarski in 1933 [Tar33], which is the prominent semantics adopted in the field of constraint programming. Given a signature $S = (X, F, P)$, a *structure* A is a tuple $(D, [\![]\!]_F, [\![]\!]_P)$ where (i) D is a non-empty set of elements—called the universe of discourse, (ii) $[\![]\!]_F$ is a function mapping function symbols $f \in F$ with arity n to interpreted functions $f_A : D^n \to D$, and (iii) $[\![]\!]_P$ is a function mapping predicate symbols $p \in P$ with arity n to interpreted predicates $p_A \subseteq D^n$. An assignment is a function $X \to D$ mapping variables to values. We denote the set of assignment by *Asn*. Let $a \in Asn$, we write $a[x \mapsto d]$ the assignment in which we updated the value of x by d in a. The syntax and semantics are related by the ternary relation $A \vDash_a \varphi$, called the *entailment*, where A is a structure, $a \in Asn$ an assignment and φ a first-order formula. It is read as "the formula φ is satisfied by the assignment a in the structure A". We first give the interpretation function $[\![]\!]_a$ for evaluating the terms of the language:

$$
\begin{aligned}
[\![x]\!]_a &= a(x) \text{ if } x \in X \\
[\![f(t_1, \ldots, t_n)]\!]_a &= [\![f]\!]_F([\![t_1]\!]_a, \ldots, [\![t_n]\!]_a)
\end{aligned}
$$

The relation $\vDash$ is defined inductively as follows:

$$
\begin{array}{ll}
A \vDash_a p(t_1, \ldots, t_n) & \text{if } ([\![t_1]\!]_a, \ldots, [\![t_n]\!]_a) \in [\![p]\!]_P \\
A \vDash_a \varphi_1 \wedge \varphi_2 & \text{if } A \vDash_a \varphi_1 \text{ and } A \vDash_a \varphi_2 \\
A \vDash_a \varphi_1 \vee \varphi_2 & \text{if } A \vDash_a \varphi_1 \text{ or } A \vDash_a \varphi_2 \\
A \vDash_a \neg\varphi & \text{if } A \vDash_a \varphi \text{ does not hold} \\
A \vDash_a \exists x,\ \varphi & \text{if there exists } d \in D \text{ such that } A \vDash_{a[x \mapsto d]} \varphi \\
A \vDash_a \forall x,\ \varphi & \text{if for all } d \in D, \text{ we have } A \vDash_{a[x \mapsto d]} \varphi
\end{array}
$$

2.2 Lattice theory

In this section, we follow the definitions and style of the book [DP02], other references include [Bir67, Gra78]. A partially ordered set (poset) is a pair $\langle P, \leq \rangle$ such that P is a set and $\leq$, called the order of P, is a reflexive, antisymmetric and transitive relation. The strict order relation $<$ is defined as $x < y \Leftrightarrow x \leq y \wedge x \neq y$ for all $x, y \in P$. The reversed order $\geq$ is defined by $x \leq y \Leftrightarrow y \geq x$. Given a poset $\langle P, \leq \rangle$ and $S \subseteq P$, $x \in P$ is a *lower bound* of S if $\forall y \in S,\ x \leq y$. We denote the set of all lower bounds of S by S^ℓ. A lower bound $x \in P$ of S is the *greatest lower bound* of S if $\forall y \in S^\ell,\ x \geq y$. The *(least) upper bound* and the set of all upper bounds S^u are defined dually by reversing the order (using $\geq$ instead of $\leq$ in the definitions and vice-versa).

Definition 2.1 (Lattice). A poset $\langle L, \leq \rangle$ is a lattice if every pair of elements $x, y \in L$ has both a least upper bound and a greatest lower bound. A *bounded lattice* has a largest element $\top \in L$, called *top*, such that $\forall x \in L,\ x \leq \top$ and an least element $\bot \in L$, called *bottom*, such that $\forall x \in L,\ \bot \leq x$. A *complete lattice* has a least upper bound and greatest lower bound for every subset $S \subseteq L$.

As a matter of convenience and when no ambiguity arises, we simply write L instead of $\langle L, \leq \rangle$ when referring to ordered structures. Also, we refer to the ordering of the lattice L as $\leq_L$ and similarly for any operation defined on L.

Alternatively, a lattice can be viewed as an algebraic structure $\langle L, \sqcup, \sqcap \rangle$ where the *join operation* $x \sqcup y$ is the least upper bound of the set $\{x, y\}$ and the *meet operation* $x \sqcap y$ is its greatest lower bound. We use the notation $\bigsqcup S$ (resp. $\bigsqcap S$) to denote the least upper bound (resp. greatest lower bound) of the set S. The operation $\sqcup$ is associative ($(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$), commutative ($x \sqcup y = y \sqcup x$), idempotent ($x \sqcup x = x$) and absorbing ($x \sqcup (x \sqcap y) = x$). This is defined dually for $\sqcap$. The structures $\langle L, \leq \rangle$ and $\langle L, \sqcup, \sqcap \rangle$ are connected by the following lemma.

Lemma 2.2 (The Connecting Lemma [DP02]). *Let L be a lattice and $a, b \in L$. Then the following are equivalent: $a \leq b$, $a \sqcup b = b$ and $a \sqcap b = a$.*

This lemma implies that whenever we define the order $\leq$, the operators $\sqcup$ and $\sqcap$ are automatically entailed by $\leq$. However, because it is not always straightforward to derive one operator from the others, we usually define all three operators, without further mentioning this correspondence.

We understand the algebraic notation in the sense of information systems [Sco82]. An information system views the order $\leq$ as a "contains less information than" relation, the bottom element $\perp$ as the element with the least information, and the top element $\top$ as the element that contains all the information. In the lattices presented in this paper, $\top$ will always represent an inconsistent element, although other elements of the lattice might be inconsistent too. This will be made clear in Section 3. The elements of a set will represent the states of a computation, and functions over these states will represent a computation. Constraint solving algorithms will be the computation of interest in this paper. We capture several properties and terminology on functions as follows.

Definition 2.3. Let $\langle P, \leq \rangle$ and $\langle Q, \leq \rangle$ be posets.

(1) $f : P \longrightarrow P$ is *idempotent* if $\forall x \in P,\ f(f(x)) = f(x)$,
(2) $f : P \longrightarrow P$ is *extensive* if $\forall x \in P,\ x \leq f(x)$,
(3) $f : P \longrightarrow Q$ is *monotone* if $\forall x, y \in P,\ x \leq_P y \Rightarrow f(x) \leq_Q f(y)$,
(4) $f : P \longrightarrow Q$ is an *order-embedding* if $\forall x, y \in P,\ x \leq_P y \Leftrightarrow f(x) \leq_Q f(y)$,
(5) A function that is idempotent, extensive and monotone is called a *closure operator*,
(6) We write $(f \circ g)(x)$ for the functional composition $f(g(x))$, and f^i for $\underbrace{f \circ f \circ \ldots \circ f}_{i \text{ times}}, i \geq 0$.
(7) A fixed point of a function $f : P \longrightarrow P$ is an element $x \in P$ such that $f(x) = x$. We denote as $\mathbf{fp}(f) \stackrel{\text{def}}{=} \{x \in P \mid f(x) = x\}$ the set of fixed points of f.

The extensive property is one of the most important concepts in this paper. An extensive function captures a computation over a lattice which progresses from bottom upwards, as we gather more information as the computation goes on. We will show that many combinatorial solvers are actually extensive functions over suitable lattices. We also define chain conditions, which will be useful to relate extensive functions and termination.

Definition 2.4 (ACC and DCC). Let L be a lattice. L satisfies the ascending chain condition (ACC) if for all increasing chains $x_1 \leq \ldots \leq x_n \leq \ldots$ of elements in L, we have a $k \in \mathbb{N}$ such that $x_k = x_{k+1} = \ldots$. Dually, L satisfies the descending chain condition (DCC) if for all decreasing chains $x_1 \geq \ldots \geq x_n \geq \ldots$, we have $k \in \mathbb{N}$ such that $x_k = x_{k+1} = \ldots$.

As a computer scientist builds data structures from existing ones, we can also derive new lattices from more basic ones. We define a number of useful constructions for the rest of this paper. We will use and illustrate these constructions as we progress in the paper.

Definition 2.5 (Lattice constructions). Let $\langle L, \leq, \sqcup, \sqcap, \perp, \top \rangle$ and $\langle K, \leq, \sqcup, \sqcap, \perp, \top \rangle$ be two bounded lattices. Then new lattices can be obtained by the following constructions:

(1) *Duality*: $\langle L^\partial, \geq, \sqcap, \sqcup, \top, \perp \rangle$.
(2) *Cartesian product*: $\langle L \times K, \leq, \sqcup, \sqcap, \perp, \top \rangle$ where:
   - $L \times K \stackrel{\text{def}}{=} \{(x, y) \mid x \in L, y \in K\}$,
   - $(x, y) \leq (x', y') \stackrel{\text{def}}{=} x \leq_L x' \land y \leq_K y'$,
   - $(x, y) \sqcup (x', y') \stackrel{\text{def}}{=} (x \sqcup_L x', y \sqcup_K y')$,
   - $(x, y) \sqcap (x', y') \stackrel{\text{def}}{=} (x \sqcap_L x', y \sqcap_K y')$,

- $\bot \overset{\text{def}}{=} (\bot_L, \bot_K)$, $\top \overset{\text{def}}{=} (\top_L, \top_K)$,

  For $x_1 \in L, x_2 \in K$, we define the projection $\pi_i(x_1, x_2) = x_i$, for $i \in \{1, 2\}$. For the sake of readability, we also extend the projection over any subset $S \subseteq L \times K$ as $\pi_i[S] = \{\pi_i(x) \mid x \in S\}$.

(3) *Pointwise lifting*: $\langle [X \nrightarrow L], \leq, \sqcup, \sqcap, \bot, \top \rangle$ where:

- X is a set, $[X \nrightarrow L]$ is the set of all partial functions from X to L.
- $\sigma \leq \tau \overset{\text{def}}{=} \forall x \in \pi_1[\sigma],\ x \in \pi_1[\tau] \wedge \sigma(x) \leq_L \tau(x)$ where $\pi_1[\sigma]$ denotes the domain of σ,
- $\sigma \sqcup \tau \overset{\text{def}}{=} \lambda x. \begin{cases} \sigma(x) \sqcup_L \tau(x) & \text{if } x \in \pi_1[\sigma] \cap \pi_1[\tau] \\ \sigma(x) & \text{if } x \in \pi_1[\sigma] \setminus \pi_1[\tau] \\ \tau(x) & \text{if } x \in \pi_1[\tau] \setminus \pi_1[\sigma] \end{cases}$
- $\sigma \sqcap \tau \overset{\text{def}}{=} \lambda x. \sigma(x) \sqcap_L \tau(x)$,
- $\bot$ is the empty function, and $\top \overset{\text{def}}{=} \lambda x. \top_L$.

  This definition is similar for the lattice of all total functions that we write $[X \to L]$.

(4) *Powerset completion*: $\langle \mathcal{P}(L), \subseteq, \cup, \cap, \{\}, L \rangle$.

(5) *Linear sum*: $L \oplus K = \langle L \cup K, \leq, \sqcup, \sqcap, \bot, \top \rangle$ where:

- $x \leq y \overset{\text{def}}{=} x \leq_L y \vee x \leq_K y \vee (x \in L \wedge y \in K)$. Intuitively, the element in K are greater than all elements in L.
- $x \sqcup y \overset{\text{def}}{=} \begin{cases} x \sqcup_L y & \text{if } x, y \in L \\ x \sqcup_K y & \text{if } x, y \in K \\ x & \text{if } x \in K, y \in L \\ y & \text{if } y \in K, x \in L \end{cases}$ $\quad x \sqcap y \overset{\text{def}}{=} \begin{cases} x \sqcap_L y & \text{if } x, y \in L \\ x \sqcap_K y & \text{if } x, y \in K \\ x & \text{if } x \in L, y \in K \\ y & \text{if } y \in L, x \in K \end{cases}$
- $\bot \overset{\text{def}}{=} \bot_L$, $\top \overset{\text{def}}{=} \top_K$.

An issue with the powerset completion is that two distinct elements a and b, such that $a \leq b \in L$, are not ordered in $\mathcal{P}(L)$ since $\{a\} \not\subseteq \{b\}$. This stems from the fact that the powerset completion views its elements as atomic, and that its ordering is defined regardless of the structure of L. A traditional way of dealing with this issue is to take the down-set or up-set completion of the base lattice.

Definition 2.6 (Down-set and up-set). Let P be a poset, and $S \subseteq P$. The down-set $\downarrow S$ and up-set $\uparrow S$ are defined by:

$$\downarrow S = \{y \in P \mid \exists x \in S, y \leq x\} \qquad \uparrow S = \{y \in P \mid \exists x \in S, y \geq x\}$$

Let $a \in P$, then we write $\downarrow a$ for $\downarrow \{a\}$ and $\uparrow a$ for $\uparrow \{a\}$. The set of all down-sets of P is denoted $\mathcal{D}(P)$, and the set of all up-sets is denoted $\mathcal{U}(P)$.

Theorem 2.7. *$\langle \mathcal{D}(P), \subseteq, \cup, \cap, \{\}, P \rangle$ and $\langle \mathcal{U}(P), \supseteq, \cap, \cup, P, \{\} \rangle$ are complete lattices.*

This theorem is a standard result, see, *e.g.*, [DP02]. We remark that these two lattices are isomorphic as the complement of an upset is a downset and vice versa. From an implementation standpoint, a drawback of these completions is that an element $\{a, b\} \in \mathcal{D}(L)$ might contain redundant elements if $a \leq_L b$. From the viewpoint of information systems, b already contains all the information contained in a, thus we do not need a. To overcome this drawback, we consider the antichains of a lattice L. Intuitively, the elements of an antichain are not comparable to each other. We first give some definitions concerning antichains.

Definition 2.8 (Antichain, minimal and maximal elements). Let $\langle L, \leq \rangle$ be a lattice. An antichain is a set $S \subseteq L$ such that for all pairs of elements $a, b \in S$, we have $a \leq b \Leftrightarrow a = b$. Given a set $Q \subseteq L$, the set of its minimal and maximal elements are defined as follows:

$$\mathit{Min}\ Q = \{x \in Q \mid \forall y \in Q,\ \neg(x >_L y)\} \qquad \mathit{Max}\ Q = \{x \in Q \mid \forall y \in Q,\ \neg(x <_L y)\}$$

By definition, *Min Q* and *Max Q* are antichains.

Example 2.9. Consider the set of sets $S = \{\{0,1\},\{1,2\},\{0\},\{1\}\} \subset \mathcal{P}(\mathbb{Z})$ such that each element in S is ordered by subset inclusion. Then we have $Min\ S = \{\{0\},\{1\}\}$ and $Max\ S = \{\{0,1\},\{1,2\}\}$.

Lemma 2.10. *If Q is a non-empty finite set, then the sets of minimal and maximal elements Min Q and Max Q are not empty. Moreover, for all elements x in Q, there is a representative element $y \in Max\ Q$ that contains more information than x,* i.e., *$x \leq y$, and dually for Min Q.*

The second part of the lemma shows that the set $Max\ Q$ "contains all the information" contained in Q, *i.e.*, we do not lose information when taking the antichain. This is not generally true for infinite sets. Because an infinite chain C does not necessarily have a maximal element, we have $Max\ C = \{\}$. Therefore, the information about this chain is lost, and some elements lack a representative element.

We equip the set of antichains of a lattice with two isomorphic orderings called the *Hoare* and *Smyth* orderings. These orderings were first explored in domain theory, in the context of powerdomains [Plo76, Smy78]. An application of powerdomains is to give a semantics to nondeterministic language constructs such as guarded commands [Sco82]. We consider the Hoare and Smyth constructions in the context of lattice theory here. We write $\mathcal{P}_f(L)$ the set of finite subsets of L.

Definition 2.11 (Hoare construction). Let $\langle L, \leq \rangle$ be a lattice. Then the Hoare construction $\langle L^H, \leq, \sqcup, \sqcap, \bot, \top \rangle$ is defined as follows:

- $L^H = \{S \subseteq \mathcal{P}_f(L) \mid S \text{ is an antichain in } L\}$,
- $X \leq Y \stackrel{\text{def}}{=} \forall x \in X,\ \exists y \in Y,\ x \leq_L y$,
- $X \sqcup Y \stackrel{\text{def}}{=} Max\ (X \cup Y)$,
- $X \sqcap Y \stackrel{\text{def}}{=} Max\ \{x \sqcap_L y \mid x \in X \wedge y \in Y\}$,
- $\bot \stackrel{\text{def}}{=} \{\}$ and $\top \stackrel{\text{def}}{=} \{\top_L\}$.

Definition 2.12 (Smyth construction). Let $\langle L, \leq \rangle$ be a lattice. Then the Smyth construction $\langle L^S, \leq, \sqcup, \sqcap, \bot, \top \rangle$ is defined as follows:

- $L^S = \{S \subseteq \mathcal{P}_f(L) \mid S \text{ is an antichain in } L\}$,
- $X \leq Y \stackrel{\text{def}}{=} \forall y \in Y,\ \exists x \in X,\ x \leq_L y$,
- $X \sqcup Y \stackrel{\text{def}}{=} Min\ \{x \sqcup_L y \mid x \in X \wedge y \in Y\}$,
- $X \sqcap Y \stackrel{\text{def}}{=} Min\ (X \cup Y)$,
- $\bot \stackrel{\text{def}}{=} \{\bot_L\}$ and $\top \stackrel{\text{def}}{=} \{\}$.

When the base lattice L is clear from the context, we will write $\leq_H$ instead of $\leq_{L^H}$ to improve the readability (and similarly for other operations, as well as those of the Smyth construction).

We give some intuitions on these constructions, and then prove they generate lattices. The choice of the Hoare or Smyth lattice very much depends on the semantics we want to give to the base lattice. For example, let $\{a, b\}$ be an antichain in the base lattice L. Both orderings allow us to refine a or b with respect to $\leq_L$, thus we have $\{a, b\} \leq \{a, c\}$ if $b \leq_L c$. The essence of the Hoare lattice L^H lies in the fact that an antichain $\{a, b\}$ can be extended with any new element $d \in L$ that is not comparable to a or b, thus obtaining the new antichain $\{a, b, d\}$. This explains why $\bot_H$ is the empty set: we can add new elements as the computation progresses; $\top_H$ being the join of all information. Dually, the Smyth lattice allows us to forget about some uninteresting elements—for example inconsistent states—and thus we have $\{a, b\} \leq_S \{a\}$. The Smyth view of the world considers that a computation starts with *too much information* that needs to be refined.

We now prove that L^H and L^S are lattices if L is a lattice. To show that, we rely on embeddings from L^H to $\mathcal{D}(L)$, and from L^S to $\mathcal{U}(L)$.

Lemma 2.13. *Let L be a lattice. The functions*

$$\begin{array}{ll} \gamma_H : L^H \to \mathcal{D}(L) & \gamma_S : L^S \to \mathcal{U}(L) \\ \gamma_H(X) = \downarrow X & \gamma_S(X) = \uparrow X \end{array}$$

are order-embeddings.

Proof. We prove this lemma for γ_H, as the proof for γ_S is similar. Let $X, Y \in L^H$, then we have:

$$\begin{aligned} X \leq_H Y \Leftrightarrow &\quad \gamma_H(X) \leq_{\mathcal{D}(L)} \gamma_H(Y) \\ \forall a \in X,\ \exists b \in Y,\ a \leq_L b \Leftrightarrow &\quad \downarrow X \subseteq \downarrow Y \end{aligned}$$

- ($\Rightarrow$) For all $a \in X$, there is $b \in Y$ such that $a \leq_L b$. Hence, we have $\downarrow a \subseteq \downarrow b \subseteq \downarrow Y$. Since it holds for all $a \in X$, we have $\cup_{a \in X} \downarrow a = \downarrow X \subseteq \downarrow Y$.
- ($\Leftarrow$) Let $a \in \downarrow X$, then there exists $b \in \downarrow Y$ such that $a \leq_L b$. Moreover, there must exist $c \in Y$ such that $b \leq c$, because Y is a finite set. Therefore, for all $a \in \downarrow X$, hence all $a \in X$, there is an element $c \in Y$ such that $a \leq_L c$.

□

Lemma 2.14. *Let L be a lattice. γ_H and γ_S are lattice homomorphisms, that is, for all $X, Y \in L^H$, $\gamma_H(X \sqcup Y) = \gamma_H(X) \sqcup \gamma_H(Y)$ and $\gamma_H(X \sqcap Y) = \gamma_H(X) \sqcap \gamma_H(Y)$ (similarly for γ_S).*

Proof. We have $\gamma_H(X \sqcup_H Y) = \downarrow Max\ (X \cup Y) = \downarrow(X \cup Y) = \downarrow X \cup \downarrow Y = \gamma_H(X) \sqcup \gamma_H(Y)$. The meet is proved as follows: $\gamma_H(X \sqcap_H Y) = \downarrow(Max\ \{x \sqcap_L y \mid x \in X \land y \in Y\} = \downarrow(\{x \sqcap_L y \mid x \in X \land y \in Y\})$. We have $x \sqcap_L y \in \downarrow X \cap \downarrow Y$ for all $x \in X$ and $y \in Y$, therefore $\downarrow(\{x \sqcap_L y \mid x \in X \land y \in Y\}) \subseteq \downarrow X \cap \downarrow Y$. Let $z \in \downarrow X \cap \downarrow Y$, then necessarily there is $x \in X$ and $y \in Y$ such that $z \leq_L x$ and $z \leq_L y$, thus $z \leq_L x \sqcap_L y$ and $\downarrow z \subseteq \downarrow(x \sqcap_L y)$. Since this holds for all $z \in \downarrow X \cap \downarrow Y$, we have $\downarrow(\{x \sqcap_L y \mid x \in X \land y \in Y\}) \supseteq \downarrow X \cap \downarrow Y$. Because both the join and meet operations are preserved, we conclude that γ_H is a lattice homomorphism. The proof is similar for γ_S. □

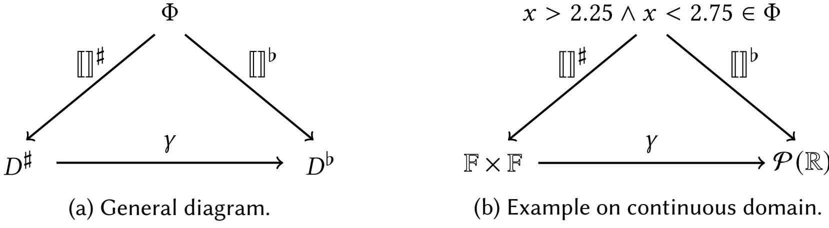
Proposition 2.15. *Let L be a lattice, then $\langle L^H, \leq \rangle$ and $\langle L^S, \leq \rangle$ are lattices.*

Proof. Because γ_H is an order-embedding (Lemma 2.13) and a homomorphism (Lemma 2.14), the sublattice $\gamma_H(L^H)$ of $\mathcal{D}(L)$ is isomorphic to L^H (this is a standard result, see, *e.g.*, [DP02]). Therefore, L^H must be a lattice. The same goes for L^S. □

As an additional result, Crampton and Loizou have shown that γ_H is a lattice isomorphism when the base lattice L is finite [CL01].

3 ABSTRACT CONSTRAINT PROGRAMMING

In the last decade, abstract interpretation has shown promising results towards providing a "grand unification theory" among the fields of constraint reasoning [PMTB13, DHK13, DHK14, CCM13, TMT20]. Figure 2a presents in a nutshell the fragment of abstract interpretation we are interested in. The syntax of a program, or in our case, of a constraint problem is represented by a set Φ of first-order formulas. We interpret a formula φ in a concrete or abstract domain respectively with $[\![\varphi]\!]^\flat$ and $[\![\varphi]\!]^\sharp$. The concrete domain represents the mathematical semantics of this formula, which is its exact set of solutions, possibly infinite and not extensionally representable in a computer. The abstract domain usually corresponds to the machine semantics of this formula, which might under- or over-approximate the set of solutions of the concrete domain. An over-approximation contains all solutions but might also contain non-solution elements. In contrast, an under-approximation contains only solutions but not necessarily all. Moreover, the abstract and concrete domains are



(a) General diagram.

(b) Example on continuous domain.

Fig. 2. Non-commuting diagram of the functions connecting a logical formula, a concrete domain and an abstract domain.

connected by a monotonic concretization function $\gamma : D^\sharp \rightarrow D^\flat$. We illustrate this fundamental notion of approximation on an example.

Example 3.1. Approximations are particularly illuminating on continuous domains, such as real numbers, which in practice can be approximated with floating point numbers. We show in Figure 2b the instantiation of the general diagram to a continuous domain. Let $x > 2.25 \land x < 2.75 \in \Phi$ be a logical formula. Then the *concrete solution set* of this formula is $\{x \in \mathbb{R} \mid x > 2.25 \land x < 2.75\}$. It is not possible to represent all real numbers in a machine. Therefore, we rely on the abstract domain of floating point intervals $\mathbb{F} \times \mathbb{F}$. We can approximate the concrete solution set in this abstract domain in two possible ways:

- *Over-approximation*: $[\![x > 2.25 \land x < 2.75]\!]^\sharp_\uparrow = [2.25..2.75] \in \mathbb{F}^2$ (2.25 and 2.75 are not solutions).
- *Under-approximation*: $[\![x > 2.25 \land x < 2.75]\!]^\sharp_\downarrow = [2.375..2.625] \in \mathbb{F}^2$ (2.3 and 2.7 are missing solutions). A tighter representation is possible by taking the closest floating point number after 2.25 and before 2.75, but some solutions will still be missing.

Note that with a more expressive abstract domain (such as open intervals), this formula might be exactly represented. However, in general, it is not always possible to exactly represent a formula.

Remark. In the field of abstract interpretation, under- and over-approximations are related to soundness and completeness properties of the abstraction, respectively. Unfortunately, in the field of constraint solving, under-approximation matches the completeness property and over-approximation the soundness property. To avoid confusion, we will simply talk about these properties as under- or over-approximations. We formally define them in Section 3.2.

We now make precise the definitions of the concrete and abstract domains. This section forms the theoretical backbone for the next sections in which we create various constraint solvers as abstract domains.

3.1 Concrete domain

The *universe of discourse* is the set of possible values for a variable in a solution. The concrete domain automatically unfolds from the universe of discourse.

Definition 3.2 (Concrete domain). Let V be a set of values, the universe of discourse, and X a set of variables. We have $Asn = [X \rightarrow V]$, the set of all assignments of the variables to values. The concrete domain $D^\flat = \langle \mathcal{P}(Asn), \supseteq \rangle$ is the dual lattice of the powerset of assignments.

The solution space is the dual of the powerset completion since we have fewer solutions when we add more constraints, thereby having more information about the problem. Given a structure A,

we connect a logical formula to an element of the concrete domain as follows:

$$\llbracket . \rrbracket^\flat : \Phi \to D^\flat$$
$$\llbracket \varphi \rrbracket^\flat = \{a \in Asn \mid A \vDash_a \varphi\}$$

We connect the entailment relation $\vDash$ to the lattice operators as follows:

$$\llbracket true \rrbracket^\flat \stackrel{\text{def}}{=} \bot^\flat = D^\flat \qquad \llbracket false \rrbracket^\flat \stackrel{\text{def}}{=} \top^\flat = \{\}$$
$$\llbracket \varphi_1 \land \varphi_2 \rrbracket^\flat = \llbracket \varphi_1 \rrbracket^\flat \sqcup \llbracket \varphi_2 \rrbracket^\flat \quad \llbracket \varphi_1 \lor \varphi_2 \rrbracket^\flat = \llbracket \varphi_1 \rrbracket^\flat \sqcap \llbracket \varphi_2 \rrbracket^\flat \quad \llbracket \neg\varphi \rrbracket^\flat = D^\flat \setminus \llbracket \varphi \rrbracket^\flat$$

Applying the interpretation function to a logical formula directly yields the set of solutions. The terms of the formula are interpreted according to the structure A.

Example 3.3. Discrete and continuous constraint satisfaction problems (CSPs) are two examples of concrete domains respectively over the domains of discourse of integer and real numbers [RvBW06]. As the definitions for both are similar, we let $\mathbb{A}$ be either the set $\mathbb{Z}$ or $\mathbb{R}$. A CSP is a tuple $\langle X, D, C \rangle$ where X is a finite set of variables, D the set of values of the variables, and C a set of relations over the variables. For each variable $x_i \in X$, we have $D_i \subseteq \mathbb{A}$. The tuple $\langle X, D, C \rangle$ is a structured presentation of the logical formula $\bigwedge_{1 \leq i \leq n} x_i \in D_i \land \bigwedge_{1 \leq i \leq |C|} C_i$. The term $x_i \in D_i$ is interpreted as $\llbracket x_i \in D_i \rrbracket^\flat \stackrel{\text{def}}{=} \{a \in Asn \mid a(x_i) \in D_i\}$. The constraints are interpreted in the standard structure of arithmetic. For instance, for the constraint $x < y$, we have $\llbracket x < y \rrbracket_P = \{(a, b) \in \mathbb{A} \times \mathbb{A} \mid a < b\}$.

In the case of an optimization problem, the solutions are further restricted to be the best according to an optimization objective. The best solutions are not necessarily unique, as it is often the case in multi-objective optimization, where best solutions form a Pareto front. Nevertheless, it does not change the definition of the concrete domain. We investigate optimization problems in Section 7.

In the following, abstract elements will represent formulas with a finite number of variables. It is useful to define a completion of a set of partial functions to a set of total functions (the concrete domain).

Definition 3.4 (Completion of a partial solutions set). We define $\hat{\gamma} : \mathcal{P}([X \nrightarrow V])^\partial \to D^\flat$ the completion of a set of partial assignments P to the values of all the variables in X not occurring in P.

$$\hat{\gamma}(P) = \{a \in Asn \mid \exists \sigma \in P,\ \forall x \in \pi_1[\sigma],\ \sigma(x) = a(x)\}$$

The concrete domain is an extensional mathematical formulation of the solution space of a logical formula. However, it is often too costly to directly manipulate due to its very large state space, or even impossible in the case of infinite domains such as with real numbers. This is why we need more compact representations and approximations of the concrete domain. This is the role of abstract domains.

3.2 Abstract domain

In abstract interpretation, an abstract domain is a partially ordered set equipped with useful operations for programs analysis. This notion has been adapted to constraint reasoning, where the main novelty is the addition of a split operator. Our approach extends the abstract framework described in [PMTB13]—specialized to over-approximations and continuous domains—to under-approximations and discrete domains. In this paper, "abstract domain" will refer to this modified notion of abstract domain for constraint reasoning that is defined as follows.

Definition 3.5 (Abstract domain). An abstract domain for constraint reasoning is a lattice $\langle A, \leq, \sqcap, \sqcup, \bot, \top\rangle$ where A is a set of computer-representable[2] elements equipped with the following operations:

- $\gamma : A \to D^\flat$ is a monotonic *concretization* function mapping an abstract element to its set of solutions.
- $[\![.]\!] : \Phi \nrightarrow A$ is a partial interpretation function approximating the set of solutions of a formula to an element of the abstract domain[3].
- *refine* : $A \to A$ is an extensive function refining the approximation of the interpreted formula.
- *split* : $A \to A^S$ is an extensive function, *i.e.*, $\forall a \in A,\ \{a\} \leq_S split(a)$, dividing an element of an abstract domain into a set of sub-elements. An element $a \in A$ such that $split(a)$ is a singleton is called *unsplittable*. We require $split(a) = \{a\}$ for all unsplittable elements.

The *constraint language* of an abstract domain is the domain of the function $[\![.]\!]$. We will say that φ is *interpretable* in A if $[\![\varphi]\!]$ is defined. We interpret conjunctive formulas and truth values as follows in an abstract domain A:

$$[\![true]\!] \stackrel{\text{def}}{=} \bot_A \qquad [\![false]\!] \stackrel{\text{def}}{=} \top_A \qquad [\![\varphi_1 \land \varphi_2]\!] \stackrel{\text{def}}{=} [\![\varphi_1]\!]_A \sqcup_A [\![\varphi_2]\!]_A$$

Indeed, all of the following domains view the join operator as adding conjunctive information in the lattice. Moreover, the interpretation function assumes that free variables are existentially quantified, since we usually want to find the models of the formula.

An abstract domain can be classified into two categories if it under- or over-approximates the set of solutions of the interpreted formula. We adapt these properties from abstract interpretation in our framework as follows. For all interpretable formulas φ:

$$\exists i \in \mathbb{N},\ (\gamma \circ refine^i \circ [\![.]\!])(\varphi) \subseteq [\![\varphi]\!]^\flat \qquad \text{(under-approximation)} \tag{1a}$$

$$\forall i \in \mathbb{N},\ (\gamma \circ refine^i \circ [\![.]\!])(\varphi) \supseteq [\![\varphi]\!]^\flat \qquad \text{(over-approximation)} \tag{1b}$$

We expect that successive applications of the refinement operator *eventually* leads to an under-approximation. Although we aim at computing an under-approximation, the initial element $[\![\varphi]\!]$ usually over-approximates $[\![\varphi]\!]^\flat$. Otherwise, if $[\![\varphi]\!]$ under-approximates $[\![\varphi]\!]^\flat$, no further refinement step is needed. Due to the extensive property of *refine*, once an element $a \in A$ under-approximates φ, then any element $refine^i(a)$ also under-approximates φ. The existential quantifier in the definition reflects these observations. In contrast, over-approximations are defined with a universal quantifier. In this case, given an element $a \in A$ over-approximating a formula, each element $refine^i(a)$ refining a must over-approximate φ as well.

From a computational viewpoint, computing an under-approximation is deeply connected to the termination of the solving procedure. According to the existential quantifier $\exists i \in \mathbb{N}$, there must be a finite number of refinement steps in order to obtain an under-approximation. There is no such restriction on the computation of over-approximations: we can infinitely refine an element in order to obtain a more precise over-approximation. Hence, we can terminate the refinement after any number of iterations, depending on the precision needed, because every element in the sequence over-approximates the formula.

An abstract domain is *under-approximating* if it satisfies equation (1a) for all interpretable formulas, and *over-approximating* if it satisfies equation (1b). If it satisfies both equations, we

[2] Each element of A occupies a finite amount of memory. Nevertheless, A might contain an infinite number of finite elements.
[3] The interpretation function is usually called *transfer function* in the context of program analysis. Alternatively, this function could be total and every unsupported formula mapped to $\bot$ which is a correct over-approximation. However, it prevents us from distinguishing between tautological formulas (since $[\![true]\!] = \bot$) and unsupported formulas. In the first case, we wish to interpret the formula in A, while in the second case we prefer to look for another, more suitable, abstract domain.

say that the abstract domain is *exact*. When the approximation kind matters, we denote under-approximating abstract domains and operators with $A_\downarrow, [\![.]\!]_\downarrow, \ldots$, and over-approximating ones with $A_\uparrow, [\![.]\!]_\uparrow, \ldots$.

3.3 Propagate and search

In abstract interpretation, the computation of an approximation is guided by the program to analyze. Therefore, an abstract analyser carries an abstract element which, at any stage, approximates the semantics of the program analysed so far. One major difference in constraint programming is the presence of backtracking. It splits an element into several sub-elements which are refined individually. Once an element cannot be split anymore, the solving procedure backtracks and refines another element. Splitting is necessary when the refinement operator cannot reach an under-approximation on its own, or to improve the precision of the over-approximation. We define this procedure over an abstract domain A as follows.

1: **function** `solve`$(a) : A \rightarrow \mathcal{P}(A)$
2: $\quad B \leftarrow (\mathit{split} \circ \mathit{refine})(a)$
3: $\quad$ **if** $|B| \leq 1$ **then return** B
4: $\quad$ **else**
5: $\quad\quad$ **return** $\bigcup_{b_i \in B}$ `solve`(b_i)
6: $\quad$ **end if**
7: **end function**

This algorithm follows the usual solving pattern in constraint programming of *propagate and search* (see, *e.g.*, [Apt03, Tac09]). We infer new information and improve the current approximation with *refine*, and then divide the problem into sub-problems with *split*. A base case is reached when the element becomes unsplittable or empty. We obtain a set of approximations of a formula φ in an abstract domain A with $\mathit{solve}([\![\varphi]\!])$.

This algorithm allows us to lift the refinement operator of an abstract domain to be under-approximating, although it might not be under-approximating initially. In this respect, we might be tempted to adapt the definition of under-approximation to this particular case. We will introduce in Section 6 the search tree abstract domain which has `solve` as a refinement operator. We will study its under- and over-approximations properties in detail at that time. Thanks to the search tree abstract domain, there is no need to adapt the definitions of under- and over-approximation to take into account the split function.

3.4 Relationship between approximations and satisfiability

The crux of constraint reasoning is generally to establish the satisfiability or unsatisfiability of a formula. We connect under-approximation to satisfiability and over-approximation to unsatisfiability.

Let $a \in A$ be an element under-approximating the formula φ. An under-approximation is not suited to prove unsatisfiability because it does not necessarily explore the full state space. Nonetheless, whenever $\gamma_A(a) \subseteq [\![\varphi]\!]^\flat$ and $\gamma_A(a) \neq \{\}$, we can deduce that φ is satisfiable, and that all elements in $\gamma_A(a)$ are solutions of φ. In contrast, we cannot deduce from an over-approximating element $a \in A$ that φ is satisfiable because $\gamma_A(a) \supseteq [\![\varphi]\!]^\flat$ and $[\![\varphi]\!]^\flat$ could be empty. Nonetheless, over-approximations are helpful to deduce unsatisfiability since $\{\} = \gamma_A(a) \supseteq [\![\varphi]\!]^\flat$ only if $[\![\varphi]\!]^\flat = \{\}$.

The issue with these formulations is that they rely on the concretization function and concrete interpretation which are "theoretical functions" that might not exist in practice. We must rely on functions which are computable to classify abstract elements into satisfiable and unsatisfiable elements. This classification task is the second role of the split function.

Let $a = refine^i(\llbracket\varphi\rrbracket)$ be an element under-approximating φ. Without loss of generality, we might further suppose that a is a fixed point of *refine* because it is generally not useful to further refine an under-approximation. Hence, we would like to classify the fixed points of *refine* as being empty ($\gamma(a) = \{\}$) or not. The split function is used for this purpose as follows:

$$split(a) \neq \{\} \Rightarrow \gamma(a) \neq \{\} \wedge \gamma(a) \subseteq \llbracket\varphi\rrbracket^\flat \tag{2}$$

where a is a fixed point of *refine*. To put it concretely, if $split(a) \neq \{\}$ then a is a solution of φ, otherwise we cannot conclude anything.

The case of over-approximation is dual and expressed as follows:

$$split(a) = \{\} \Rightarrow \gamma(a) = \{\} \tag{3}$$

where $a = refine^k(\llbracket\varphi\rrbracket)$ for any $k \in \mathbb{N}$. This property guarantees that a has no solution if the set $split(a)$ is empty.

4 DOMAIN OF A VARIABLE

The domain of a variable is an abstract domain in which formulas with a single variable can be interpreted. As the variable's name is not represented explicitly in these domains, without loss of generality, we arbitrarily fix the name of the variable to the underscore symbol _. It allows us to pass well-formed logical formulas to their interpretation functions. For instance, $_ > 4$ can be simply understood by replacing _ with a more traditional variable name, *e.g.*, $x > 4$. We see later in Section 5.1 a construction to index any domain of variable with a variable name. We first introduce two standard constructions to represent the domain of a variable without an ordering requirement on the universe of discourse. Then, we observe that if the universe of discourse can be viewed as a lattice, several additional constructions, potentially more efficient, can be proposed.

4.1 Constructions over unordered universe of discourse

We can obtain a lattice by equipping the universe of discourse with bottom and top elements, and by leaving its elements unordered.

Definition 4.1 (Flat lattice). Let S be a set, the universe of discourse, then $FL(S) = \{\bot\} \oplus S \oplus \{\top\}$ is an abstract domain where:

- Lattice operations are inherited from the linear sum (Def. 2.5(5)), and $\bot, \top \notin S$,
- $\gamma(a) \stackrel{\text{def}}{=} \hat{\gamma}(P)$ with $P = \begin{cases} \{\{_ \mapsto a\}\} & \text{if } a \notin \{\bot, \top\} \\ \{\} & \text{if } a = \top \\ \{\{_ \mapsto v\} \mid v \in S\} & \text{if } a = \bot \end{cases}$
- $\llbracket _ = v \rrbracket \stackrel{\text{def}}{=} v$ if $v \in S$,
- $split_\downarrow(a) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } a = \top \\ \{a\} & \text{if } a \notin \{\bot, \top\} \\ Q & \text{if } a = \bot \text{ and } Q \text{ a finite subset of } S \end{cases}$
- $split_\uparrow(a) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } a = \top \\ \{a\} & \text{if } a \notin \{\bot, \top\} \\ S & \text{if } a = \bot \text{ and } S \text{ is finite} \\ \{\bot\} & \text{otherwise} \end{cases}$
- $refine(a) \stackrel{\text{def}}{=} a$.

The concretization function maps each element of FL to its concrete set of solutions using the completion $\hat{\gamma}$ (Definition 3.4).

The operator *split* comes in two versions if the base set S is infinite. It under-approximates the state space by taking a finite subset of S. The over-approximating version does not attempt to split $\bot$ if S is infinite, otherwise it would result in a set that does not belong to $FL(S)^S$ (the image of *split*). We illustrate in Figure 3a the flat lattice derived from the set $\mathbf{3} = \{0, 1, 2\}$.

Another common variable's domain is the powerset construction which allows us to represent a collection of possible values, whereas a flat lattice stores none, one or all elements at a time.

Definition 4.2 (Powerset variable's domain). Let S be a set. We take the set of finite subsets of S to form the lattice $PVD(S) = \langle \mathcal{P}_f(S), \supseteq, \cap, \cup \rangle$ with $\top \stackrel{\text{def}}{=} \{\}$ and without bottom element. $PVD(S)$ is an exact abstract domain where ($T \in \mathcal{P}_f(S)$):

- $\gamma(T) \stackrel{\text{def}}{=} \hat{\gamma}(\{\{_ \mapsto v\} \mid v \in T\})$,
- $[\![_ \in T]\!] \stackrel{\text{def}}{=} T$ if T is finite,
- $[\![_ \notin T]\!] \stackrel{\text{def}}{=} S \setminus T$ if T is cofinite,
- $split(T) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } |T| = 0 \\ \{T\} & \text{if } |T| = 1 \\ \{T \setminus R, R\} \text{ with } R \subset T & \text{otherwise} \end{cases}$
- $refine(T) \stackrel{\text{def}}{=} T$.

Many formulas can be interpreted extensionally in this abstract domain, but this is often not very efficient. We require every set $T \in PVD(S)$ to be finite because it needs to be representable in a computer. Nevertheless, the solutions set $\gamma(Q)$ can be infinite if the base set S is infinite. As with the flat lattice, various *split* operators are possible depending on which subset of T is selected.

The powerset construction is notably used to represent the domains of discourse of finite integers and sets. In Minion [GJM06], they represent integer variables with bit vectors whenever the largest integer in the variable's domain is small enough. This allows them to take advantage of bitwise operations and improve memory consumption. Another example is given by the universe of discourse of sets $\mathcal{P}(S)$ for any set S. We can represent the domain of a set variable as a set of sets. The construction is thus $PVD(\mathcal{P}(S))$ where $\mathcal{P}(S)$ is the (unordered) universe of discourse. The interpretation function can be extended to support the subset notation (with $T \in \mathcal{P}_f(S)$):

$$[\![_ \subseteq T]\!] = \mathcal{P}(T) \text{ and } [\![_ \supseteq T]\!] = (\mathcal{P}(S) \setminus \mathcal{P}(T)) \cup T \quad \text{if } S \text{ is finite}$$

The language CLPS-B [BLP02] is based on this extensional representation of set. However, as the powerset representation grows exponentially in the size of the underlying set S, this representation is only practical for sets S of small cardinalities.

4.2 Constructions over ordered universe of discourse

We can devise new constructions taking advantage of the ordering of a universe of discourse, when one is available. For instance, a simple arithmetic constraint such as $x \geq 4$ cannot be represented effectively in the previous abstract domains. Nevertheless, if we equip the universe of discourse of integer numbers with its natural order $\langle \mathbb{N}, \leq \rangle$, we can represent this constraint by the element 4, which means that the value of x is any number equal to or greater than 4. The formal characterization of this idea is to view the concrete solution space of an abstract element as its up-set, *i.e.*, the abstract element and all the elements above it. As a first example, we consider an abstract domain for the universe of discourse of integers.

Definition 4.3 (Increasing integers). We denote the lattice of increasing integers $\mathbb{Z}^\sharp = \{-\infty\} \oplus \langle \mathbb{Z}, \leq \rangle \oplus \{\infty\} = \langle \mathbb{Z} \cup \{-\infty, \infty\}, \leq, max, min, -\infty, \infty \rangle$ where $\leq$ is the natural arithmetic ordering. $\mathbb{Z}^\sharp$ is an exact abstract domain with the following operators (with $v \in \mathbb{Z} \cup \{-\infty, \infty\}$):

- $\gamma(a) \stackrel{\text{def}}{=} \hat{\gamma}(\{\{_ \mapsto v\} \mid v \in \mathbb{Z}, v \geq a\})$,
- $[\![_ \geq v]\!] \stackrel{\text{def}}{=} v$ and $[\![_ > v]\!] \stackrel{\text{def}}{=} v + 1$,
- $split(a) \stackrel{\text{def}}{=} \begin{cases} \{a\} & \text{if } a \neq \infty \\ \{\} & \text{if } a = \infty \end{cases}$
- $refine(a) \stackrel{\text{def}}{=} a$.

We note that $split([\![_ \geq \infty]\!]) = \{\}$ since no integer is greater or equal to infinity.

Similarly, we define the abstract domain $\mathbb{Q}^\sharp = \langle \mathbb{Q} \cup \{-\infty, \infty\}, \leq, max, min, -\infty, \infty \rangle$ for rational numbers, with the difference that $[\![_ > v]\!]$ cannot be an exact interpretation since the set $\mathbb{Q}$ is dense. Although elements in $\mathbb{Q}$ are representable in a computer as a pair of integers, real numbers might not be. The usual representation is to approximate real numbers with floating point numbers.

Definition 4.4 (Increasing floating point numbers). The lattice of increasing floating point numbers $\mathbb{F}^\sharp = \langle \mathbb{F} \setminus \mathsf{NaN}, \leq, max, min, -\infty, \infty \rangle$, where $\mathbb{F}$ is defined according to IEEE 740 standard [IEE19] without the set NaN of *not a number* values[4]. We write $\lfloor v \rfloor$ (resp. $\lceil v \rceil$) the function returning the closest floating point number less or equal to v (resp. greater or equal to v). $\mathbb{F}^\sharp$ is an abstract domain with the following operators (with $v \in \mathbb{R}$):

- $\gamma(a) \stackrel{\text{def}}{=} \hat{\gamma}(\{\{_ \mapsto v\} \mid v \in \mathbb{R}, v \geq a\})$,
- $[\![_ \geq v]\!]_\uparrow \stackrel{\text{def}}{=} \lfloor v \rfloor$ and $[\![_ \geq v]\!]_\downarrow \stackrel{\text{def}}{=} \lceil v \rceil$,
- $[\![_ > v]\!]_\uparrow \stackrel{\text{def}}{=} \lfloor v \rfloor$,
- $[\![_ > v]\!]_\downarrow \stackrel{\text{def}}{=} \lceil v \rceil$ if $\lceil v \rceil \neq v$, otherwise $[\![_ > v]\!]_\downarrow \stackrel{\text{def}}{=} succ(v)$ where $succ(v)$ is the next representable floating point number greater than v.
- *split* and *refine* defined similarly than in Definition 4.3.

These three arithmetic abstract domains of increasing numbers possess a total order. It is also possible to define abstract domains for partially ordered domains of discourse. An example is the universe of discourse of sets.

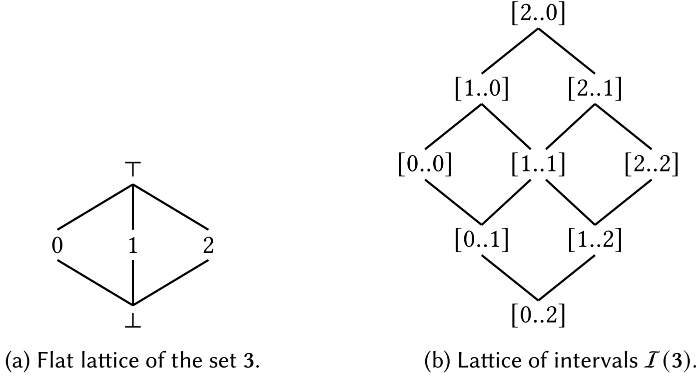
Definition 4.5 (Increasing sets). Let S be a set. Then $S^\sharp = \langle \mathcal{P}_f(S), \subseteq, \cup, \cap \rangle$ with $\bot \stackrel{\text{def}}{=} \{\}$ and without a top element unless S is finite. $S^\sharp$ is an abstract domain with the following operators:

- $\gamma(Q) \stackrel{\text{def}}{=} \hat{\gamma}(\{\{_ \mapsto T\} \mid T \in \uparrow Q\})$,
- $[\![_ \supseteq Q]\!] \stackrel{\text{def}}{=} Q$ if $Q \in \mathcal{P}_f(S)$,
- $split(Q) \stackrel{\text{def}}{=} \{Q\}$,
- $refine(T) \stackrel{\text{def}}{=} T$.

Similarly to $PVD(\mathcal{P}(S))$, $S^\sharp$ is able to represent the universe of discourse of sets. The difference lies in the representation of the sets. For instance, let the universe of discourse be $S = \{1, 2, 3\}$. The constraint $_ \supseteq \{1\}$ is represented by the element $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$ in $PVD(\mathcal{P}(S))$ and by $\{1\}$ in $S^\sharp$. All the sets that include $\{1\}$ are implicitly represented in $S^\sharp$, whereas they are explicitely represented in $PVD(\mathcal{P}(S))$. The tradeoff for this compact representation is that $S^\sharp$ can only interpret constraints of the form $_ \supseteq Q$, whereas $PVD(\mathcal{P}(S))$ also supports $_ \subseteq Q$ and $_ \not\subseteq Q$.

These abstract domains can only represent constraints bounding a variable in one direction. For instance, $\mathbb{Z}^\sharp$ supports $_ \geq v$ but not $_ \leq v$. By taking the dual of any of these lattices, we gain support for the dual constraints. Therefore, by considering the Cartesian product of an abstract domain and its dual, we can bound the possible values of a variable from below and above. This

[4] From a practical viewpoint, we might assume that NaN values generate exceptions, and thus are never manipulated.



(a) Flat lattice of the set **3**. (b) Lattice of intervals $\mathcal{I}(\mathbf{3})$.

Fig. 3. Examples of lattices of variable's domain.

is given by a general construction called *interval lattice*. Interval lattice has been studied in the context of combinatorial solving by Fernández and Hill [FH04], and we extend their definition to abstract domain. An advantage of this construction is that intervals of any partially ordered universe of discourse, such as sets, is supported. In order to generically define under- and over-approximating version of this abstract domain, without unnecessarily duplicating definitions, we annotate operators with $\updownarrow \in \{\uparrow, \downarrow\}$. The definition can then be instantiated to an under- or over-approximation by choosing the appropriate arrow.

Definition 4.6 (Interval abstract domain). Let A be an abstract domain and A^∂ its dual. Then $\mathcal{I}(A) = \langle A \times A^\partial, \leq, \sqcup, \sqcap, \bot, \top \rangle$ is an abstract domain where:

- $\leq, \sqcup, \sqcap, \bot, \top$ are directly inherited from the Cartesian product,
- We write $[l..u]$ the interval $(l, u) \in \mathcal{I}(A)$,
- $\gamma([l..u]) \stackrel{\text{def}}{=} \gamma_A(l) \cap \gamma_{A^\partial}(u)$,
- $[\![_ \square v]\!]_\updownarrow \stackrel{\text{def}}{=} [\bot_A..u]$ whenever $u = [\![_ \square v]\!]^{A^\partial}_\updownarrow$ is defined, where $\square \in \{\leq, <, \subseteq\}$,
- $[\![_ \square v]\!]_\updownarrow \stackrel{\text{def}}{=} [l..\top_{A^\partial}]$ whenever $l = [\![_ \square v]\!]^{A}_\updownarrow$ is defined, where $\square \in \{\geq, >, \supseteq\}$,
- $[\![_ = v]\!]_\updownarrow = [\![_ \geq v \land _ \leq v]\!]_\updownarrow$ if defined, otherwise $[\![_ = v]\!]_\updownarrow = [\![_ \subseteq v \land _ \supseteq v]\!]_\updownarrow$.
- $\mathit{refine}_\updownarrow([l..u]) \stackrel{\text{def}}{=} [\mathit{refine}^A_\updownarrow(l)..\mathit{refine}^{A^\partial}_\updownarrow(u)]$,
- $\mathit{split}_\updownarrow([l..u]) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } l >_A u \lor \mathit{split}^A_\updownarrow(l) = \{\} \lor \mathit{split}^{A^\partial}_\updownarrow(u) = \{\} \\ \{[l..u]\} & \text{if } l = u \\ \{[l..u'], [l'..u]\} & \text{where } u' = [\![_ \leq m]\!]^{A^\partial}_\updownarrow,\ l' = [\![_ > m]\!]^{A}_\updownarrow,\ l \leq_A m <_A u \end{cases}$

Example 4.7 (Interval abstract domains). The lattice of integer intervals $\mathcal{I}(\mathbb{Z}^\sharp)$ is an exact abstract domain. We illustrate this lattice in Figure 3b over the set of integers $\mathbf{3} = \{0, 1, 2\}$. We note that the interval $[0..0]$ contains more information than $[0..2]$. It captures the concept of a variable's domain in a CSP: we declare a variable with a domain between 0 and 2 because we do not know its exact value. For each $x \in \{[1..0], [2..1], [2..0]\}$, we have $\mathit{split}(x) = \{\}$. Alternatively, we could group these intervals into a single $\top$ element. Another interesting characteristics of intervals is to allow the representation of infinite sets such as some subsets of real numbers. In continuous CSP, the standard technique is to surround the real number, representing a solution, by a floating point interval. Its structure is formally defined by the abstract domain $\mathcal{I}(\mathbb{F}^\sharp)$, and has been studied in various

constraint languages such as CLP(BNR) [Old93], Newton [VHM95] and Numerica [VHMD97]. The challenge of computing with floating point intervals is to design refinement operators that compute tight bounds of arithmetic expressions, see, *e.g.*, [SBB⁺18]. It must be noted that dealing with round-off errors is a whole research area that we do not discuss here. Interval of sets, captured in the abstract domain $\mathcal{I}(S^\sharp)$, has been extensively studied in Conjunto [Ger94] and CLP($\mathcal{SET}$) [DPPR00].

This section only shows a fraction of the existing structured domains. For instance, it is possible to consider open interval as well, which allows to exactly interpret constraints such as $x > v$ with $v \in \mathbb{F}^\sharp$ as the element $(v, \infty]$. This extension does not pose further challenges and is formally defined in [FH04]. Another example is the set representation with cardinality, studied in Cardinal [Aze07]. The universe of discourse of graphs, with an interval variable's domain, has been proposed in [DDD05]. The domain of stream targets infinite sequence of data. Interestingly, an exact solving algorithm is given in [LLLS11] for ω-regular languages which are recognized by Büchi automata. This is an example of infinite domain which is both an under- and over-approximation. For non-regular languages, an over-approximating solving algorithm is studied in [BTM11]. There are many other domains of discourse including Boolean (SAT solving), Herbrand universe (notably useful in logic programming, see [Rey70, Plo70] for its corresponding lattice structure, and more recently [GNS⁺16, Cou20] for its corresponding abstract domains), string [Raj94], function [Lau78, Hni03] or relation [Lau78, FPÅ04]. We refer to the survey of Gervet in [RvBW06] (Chapter 17) for a more exhaustive account of structured domains.

5 PROPAGATION PROBLEM

We now tackle formulas with multiple variables by introducing abstract domain able to represent conjunctive collection of information. By "conjunctive collection of information", we mean a set of elements such that if an element is false, then the whole set is false. In abstract interpretation, abstract domains over multiple variables are divided in two categories:

- *Non-relational abstract domains* represent relations on variables individually from each other. More precisely, it only supports atoms with single variable in a formula.
- *Relational abstract domains* are more expressive as they can represent relations between variables. Hence, atoms with multiple variables are supported. However, as for variable's domain, some relational abstract domains can only represent atoms within a restricted constraint language.

Relational abstract domains are numerous in the field of abstract interpretation. We give a few examples, ordered from the less expressive ones to the more expressive ones, which are also more computationally intensive (x, y are variables and c a constant):

- *Octagon* [Min06] for constraints of the form $\bigwedge \pm x \pm y \leq c$, also known as difference logic, or *unit two variables per inequality* (UTVPI) domain [DMP91, JMSY94, HS97].
- *Two variables per inequality* [SKH03] for constraints of the form $\bigwedge c_1 x_1 + c_2 x_2 \leq c$.
- *Polyhedron* [CH78] for linear constraints of the form $\bigwedge \sum_{1 \leq i \leq n} c_i x_i \leq c$, which is central to the linear programming paradigm.
- We note that variants of these domains usually exist for integers, floating point numbers and rational numbers.

These abstract domains fit in our framework. For instance, octagon and polyhedron have been studied in a similar abstract constraint reasoning framework in [PTB14, TCMT19] and [ZMM⁺19] respectively.

In this section, we first introduce a non-relational *store of variables* abstract domain, useful to model array of variables. We then broaden the scope of relational abstract domains to combinatorial solving by integrating ideas from the field of constraint programming. In particular, we define an abstract domain in which elements are collection of refinement operators. Refinement operators are well-known in constraint programming under the name of *propagator functions*. Moreover, we study conditions which preserve under- and over-approximation when combining refinement operators. It gives the foundation for a bridge between the abstract interpretation and constraint programming fields.

5.1 An abstract domain for collection of variables

Until now, we supposed a variable named _ in the interpretation function, which is artificial because a variable's domain does not represent the name of the variable explicitly. Following standard practice, we rely on the lattice of partial functions to represent a store of variables.

Definition 5.1 (Store of variables). Let X be an arbitrary set and A be a variable's domain. Then we write the store construction of A as $Store(A) = \langle [X \twoheadrightarrow A], \leq, \sqcup, \sqcap, \bot, \top \rangle$ where:

- The usual lattice operations are inherited from the pointwise lifting (Def. 2.5(4)).
- $\gamma(\sigma) \stackrel{\text{def}}{=} \{a \in Asn \mid \forall x \in \pi_1[\sigma],\ a[x \mapsto _] \in \gamma_A(\sigma(x))\}$,
- $[\![\varphi]\!] \stackrel{\text{def}}{=} \{x \mapsto [\![\varphi[x \mapsto _]]\!]_A)$ if $FV(\varphi) = \{x\}$,
- $split(\sigma) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } \exists x \in \pi_1[\sigma],\ |split_A(\sigma(x))| = 0 \\ \{\sigma\} & \text{if } \forall x \in \pi_1[\sigma],\ |split_A(\sigma(x))| = 1 \\ \{\{x \mapsto a'\} \sqcup \sigma \mid a' \in split_A(\sigma(x))\} & \text{if } \exists x \in \pi_1[\sigma],\ |split_A(\sigma(x))| > 1 \end{cases}$
- $refine(\sigma) \stackrel{\text{def}}{=} \{\{x \mapsto refine_A(\sigma(x))\} \mid x \in \pi_1[\sigma]\}$

In the following we view an element of $Store(A)$ either as a partial function from X to A, or as a set of pairs in $X \times A$, called the graph of the function.

The concretization function deserves an additional explanation. We remove all the assignments $a \in Asn$ where, for a variable x, there is no concrete solution $a(x)$ in $\gamma_A(\sigma(x))$. A small technical step is to rename the variable x in the assignment a with the special name _.

Example 5.2. Let X be the set of variables $\{x, y\}$, and SI be the store of integers interval $Store(\mathcal{I}(\mathbb{Z}^\sharp))$. The stores $s_1 = \{x \mapsto [0..2]\}$ and $s_2 = \{x \mapsto [1..2], y \mapsto [3..4]\}$ are elements of SI. In contrast, the store $\{x \mapsto \bot, y \mapsto [1..2], y \mapsto [0..2]\}$ does not belong to SI because there are two variables with a location y. Also, notice that the order of the store is not the set inclusion. Indeed, $s_1 \leq s_2$ is not defined under the set inclusion order, but it is intuitive that the store s_2 contains more information than s_1 since we have $[0..2] \leq [1..2]$ and s_1 does not contain elements that are not in s_2.

5.2 Propagation problem abstract domain

Instead of developing a full-fledged abstract domain for a constraint language, constraint programming focusses on the careful design of many individual refinement operators, each implementing a particular constraint. In the context of constraint programming, the refinement operator is called a *propagator*. We consider the two concepts identical as a propagator is also an extensive function $p : A \to A$ over an abstract domain A. A propagation-based constraint solver consists of two key ingredients: propagators and a propagation engine. We start with propagators by defining the lattice $Prop$ of all propagators as follows.

Definition 5.3 (Propagators). Let A be an abstract domain. We denote $Prop \subset [A \to A]$ the set of all extensive functions on A, namely propagators. *Prop* inherits the lattice operations of the pointwise lifting (Definition 2.5). In particular, we have $p_1 \sqcup p_2 = \lambda a.p_1(a) \sqcup_A p_2(a)$.

The pointwise ordering on *Prop* captures the notion of propagator strength, in which we have $p_1 \leq p_2$ iff p_2 infers more information than p_1. The concomitant notion of *consistency* refers to properties achieved on the underlying abstract domain. Consistency is helpful to classify propagators strength over a same constraint. It has been widely studied in the field of constraint programming in order to compare the efficiency and pruning ratio of propagators. We will not need this notion in this paper, and we refer to [Apt03, Dec03, Lec09] for in-depth reviews of consistencies. Also, we point out Scott's dissertation [Sco16] which presents various consistencies as Galois connections in the framework of abstract interpretation.

A constraint solver consists of a collection of propagators on a fixed abstract domain, usually the store construction of a variable's domain of interest, *e.g.*, the store $Store(\mathcal{I}(\mathbb{Z}^\sharp))$. We illustrate the concept of propagators through two examples.

Example 5.4 ("Greater than" propagator). Propagators are defined over a base abstract domain A, and their exact definitions might vary depending on A. Nevertheless, it is still possible to define a propagator in a generic way. We show an arithmetic propagator for the constraint $x > y$, that assumes A provides the operators $\lfloor x \rfloor_a$ and $\lceil y \rceil_a$ to respectively retrieve the lower and upper bound of a variable x in $a \in A$.

$$[\![x > y]\!] = \lambda a.a \sqcup_A [\![x > \lfloor y \rfloor_a]\!]_A \sqcup_A [\![y < \lceil x \rceil_a]\!]_A$$

This propagator is defined generically for a large number of universe of discourse, such as integer, floating-point or rational number, but also sets if we equate $> \stackrel{\text{def}}{=} \subset$ and $< \stackrel{\text{def}}{=} \supset$. Moreover, this technique in conjunction with a general propagation algorithm such as HC4 [BGGP99], allows us to support a large constraint language with a single propagator.

Example 5.5 (Constructive disjunction [VHSD91]). Let φ_1 and φ_2 be logical formula implemented by the propagators p_1 and p_2 over an abstract domain A. The constructive disjunction is a propagator for disjunctive formula that relies on the meet operator of A:

$$[\![\varphi_1 \vee \varphi_2]\!] = \lambda a.\ p_1(a) \sqcap_A p_2(a) \text{ with } [\![\varphi_i]\!] = p_i, i \in \{1, 2\}$$

Information both discarded by p_1 and p_2 will be removed from a. It is useful in scheduling problems for precedence constraints of the form $x + y \leq c_1 \vee y + z \leq c_2$. To further exemplify, consider that y is defined on integer intervals, with an initial domain of $[1..5]$. If $x + y \leq c_1$ shrinks the domain of y to $[2..4]$, and $y + z \leq c_2$ to $[1..4]$, then we can safely prune the domain of y to $[2..4] \sqcap [1..4] = [1..4]$ without committing to a branch of the disjunction. Note that the formulas in the disjunction must share variables otherwise no additional pruning can be achieved in comparison to a normal disjunction. This technique can be applied to many logical connectors as shown in [GMS20].

In addition to the design of propagators, the second key ingredient of a propagation-based constraint solver is the propagation engine. The propagation engine computes a simultaneous fixed point of a collection of propagators $\{p_1, \ldots, p_n\}$. That is, an element $a \in A$ such that $p_i(a) = a$ for all $1 \leq i \leq n$. This fixed point is computable only if it is reachable in a finite number of steps, otherwise we must approximate it. It is always computable in the case of discrete constraint solving over finite domains [Apt99, Apt00, SS08, Tac09]. In continuous constraint solving, the termination criterion is usually based on a precision measure: we stop refining an element when it is small enough [Kea87, FH04, CJ09]. We introduce below an abstract domain, called *propagation problem*, encapsulating the propagation step as a refinement step. In the terms of our framework, this abstract

domain lifts the refinement operator to the structure of the lattice itself. It implies that *refine* can evolve dynamically as the computation progresses. Moreover, an important observation is that *refine* is not required to be idempotent. Therefore, we can view it as a function producing a stream of increasingly refined approximations, *i.e.*, $refine(a) \leq refine^2(a) \leq \ldots$. It enables users to choose the right termination criterion according to their requirements and the underlying abstract domain, without forcing it inside our abstract domain.

Definition 5.6 (Propagation problem). Let A be an abstract domain. Then the propagation problem $PP = \langle A \times Prop^H, \leq, \sqcup, \sqcap, \bot, \top \rangle$ is an abstract domain with the following operators:

- The lattice operations are inherited from the Cartesian product and the Hoare construction.
- $\gamma((a, P)) \stackrel{\text{def}}{=} \gamma_A(a)$
- $[\![\varphi]\!] \stackrel{\text{def}}{=} \begin{cases} ([\![\varphi]\!]_A, \{refine_A\}) & \text{if } [\![\varphi]\!]_A \text{ is defined} \\ (\bot_A, \{refine_A, p\}) & \text{where } p \text{ is a propagator implementing } \varphi \end{cases}$
- $split((a, P)) \stackrel{\text{def}}{=} \{(a', P) \mid a' \in split_A(a)\}$
- $refine_i((a, \{p_1, \ldots, p_n\})) \stackrel{\text{def}}{=} (p_i(a), \{p_1, \ldots, p_n\})$,
- $refine_{\text{SEQ}}((a, \{p_1, \ldots, p_n\})) \stackrel{\text{def}}{=} ((p_1 \circ \ldots \circ p_n)(a), \{p_1, \ldots, p_n\})$,
- $refine_{\text{PAR}}((a, \{p_1, \ldots, p_n\})) \stackrel{\text{def}}{=} ((p_1 \sqcup \ldots \sqcup p_n)(a), \{p_1, \ldots, p_n\})$.

We discuss several aspects surrounding this definition including the choice of the Hoare construction instead of the powerset completion, the concretization function, the split function, the multiple refinement operators, the additional monotone property of propagators, and the strengths and weaknesses of the propagator approach. The next sections will focus on under- and over-approximations theorems in a slightly more general setting.

Traditionally, a set of propagators is represented by the powerset completion $\mathcal{P}(Prop)$ [Ben96, Tac09]. The powerset does not capture the possibility that a propagator might be refined to a stronger one. For instance, it seems natural that $x < 4$ and $x + y \leq 3$ can be refined to $x < 3$ and $x + y \leq 2$ respectively, because it is an extensive operation on the solutions set, *e.g.*, $[\![x < 3]\!]^\flat \subseteq [\![x < 4]\!]^\flat$. The Hoare construction captures both the powerset behavior and the refinement of existing propagators to stronger ones.

The concretization of an abstract element represents the *current* set of solutions of this element. This is why the concretization is solely defined by $\gamma_A(a)$, as the propagators are part of the refinement operator. It implies that adding new propagators in the problem, without performing any refinement step, does not increase the precision of the abstract element, which seems to be the correct assumption.

The split function entirely relies on the underlying abstract domain. *Streamlining* [GS04] is an advanced example of split operator over *PP*. A streamlined constraint is problem-dependent and carefully designed to pick a small part of the state space that is known to contain solutions. The streamlined constraints are usually added at the top of the search tree, and the rest is explored with a more standard solving algorithm. It is a split operator because the complement of the streamlined constraints gives a second sub-problem to explore if the first did not lead to a solution.

We provide two basic propagation engines encapsulated in $refine_{\text{SEQ}}$ and $refine_{\text{PAR}}$ for a sequential and a fully parallel scheduling of the propagators. The sequential $refine_{\text{SEQ}}$ operator applies each propagator in turn and only once. Static scheduling of propagators is a viable strategy when the propagators are few and of similar complexity, as it sometimes occurs in continuous problems. However, for discrete problems, it has been shown that dynamically scheduling the propagators is a better strategy. We refer to [SS08] for an exhaustive performance analysis of numerous sequential propagation engines on discrete domains. In our framework, sequential propagation engines can be

implemented on top of *PP* by relying on the atomic refine_i operators. However, further development along this road is out of scope in this paper. Propagation engines as fixed point computations over partial order have been notably developed in [Ben96, Apt99, Apt00, GM03]. In contrast, $\mathit{refine}_{\text{PAR}}$ is a fully parallel propagation engine. Interestingly, $\mathit{refine}_{\text{PAR}}$ coincides with the join $\bigsqcup_{Prop} P$ of all propagators in the pointwise lifting of propagators *Prop*. It is due to the definition of the join on *Prop*, defined as $p_1 \sqcup p_2 = \lambda a.p_1(a) \sqcup_A p_2(a)$, where every propagator is applied to the domain individually. However, purely parallel computation suffers from the "parallel decoupling phenomena" where convergence may be faster when propagators are sequentially executed [GM03]. Nevertheless, this observation is worth to be re-explored in the context of massively multicore computing as initiated by recent work [GMN+18]. These different refinement operators can also be composed to obtain mixed propagation engines, as shown in [GM03]. More recently, in the field of abstract interpretation, Kim *et al.* [KVT20] have studied how to compute fixed points in parallel while taking into account sequential dependencies among functions.

We now discuss the case when the propagators and refine_A are monotone, that is, when $x \leq y \Leftrightarrow f(x) \leq f(y)$. Intuitively, it means that if we feed more information y to a function, this function maps to at least the same amount of information as if given less initial information x. The main additional result is that the fixed point of $\{p_1, \ldots, p_n\}$, if it exists, is the smallest one, regardless of the order of applications. This result stems from Knaster-Tarski fixed point theorem and chaotic iterations of abstract interpretation [CC77b]. It is studied by Apt [Apt99] in the context of propagators. We rely on monotone refinement operators in Section 5.3 to prove compositionality results of under-approximation.

We conclude our presentation of propagators by briefly reviewing their strengths and weaknesses. Due to the low requirements on propagators, which are mere extensive functions, many solving algorithms from the field of operation research, such as for scheduling, vehicle routing and assignment problems, have been integrated in constraint solvers as propagators. The chief advantage is to seamlessly combine by logical conjunction the constraints implemented by these propagators. The price to pay for the granted flexibility is that, because their internal structures are hidden, the conjunctive combination of two propagators can only be achieved through functional composition, even if they use the same underlying data structures. Moreover, as a propagator only models the refinement operator of an abstract domain, it cannot be split or queried. For instance, consider two propagators p_1 and p_2 implementing the constraints $x + y \leq 3$ and $y - z \leq 2$ respectively. These constraints can be treated more efficiently in the octagon abstract domain than by computing the fixed point of $p_1 \circ p_2$. Moreover, the octagon abstract domain can split its elements efficiently whereas, using propagators, only the underlying abstract domain is usually split, *e.g.*, the variables in the store. Pelleau *et al.* [PTB14] give various split operators for octagons on continuous domains, and a more detailed comparison between propagators and octagons can be found in [TCMT19].

5.3 Compositionality of under-approximation

The propagation problem abstract domain combines different refinement operators. However, what are the properties conserved when combining refinement operators? In particular, are the under- and over-approximation properties of two refinement operators preserved upon functional composition? We provide elements of answer here and in the next section, which are useful to establish the properties of *PP*, but also for the composition of refinement operators in general.

We first look at the elements at the end of a refinement chain. We show that if we refine separately two formulas, their results can be combined by join.

Lemma 5.7 (Preservation of under-approximation). *Let $a, b \in A$ two elements respectively under-approximating φ_1 and φ_2. Then $a \sqcup b$ under-approximates $\varphi_1 \wedge \varphi_2$.*

Proof. We have $\gamma(a) \subseteq [\![\varphi_1]\!]^\flat$ and $\gamma(b) \subseteq [\![\varphi_2]\!]^\flat$. By monotonicity of γ, $\gamma(a \sqcup b) \subseteq \gamma(a) \cap \gamma(b)$. We also have $[\![\varphi_1 \wedge \varphi_2]\!]^\flat = [\![\varphi_1]\!]^\flat \cap [\![\varphi_2]\!]^\flat$. Thus, $a \sqcup b$ is an under-approximation as we have $\gamma(a \sqcup b) \subseteq [\![\varphi_1]\!]^\flat \cap [\![\varphi_2]\!]^\flat$. Therefore, under-approximations are preserved under the join operation. □

Lemma 5.8 (Persistence of under-approximation). *Let $a \in A$ an element under-approximating φ. Then for all $b \in A$, $a \sqcup b$ also under-approximates φ.*

Proof. We have $\gamma(a) \subseteq [\![\varphi]\!]^\flat$. By monotonicity of γ, $\gamma(a \sqcup b) \subseteq [\![\varphi]\!]^\flat$ for any $b \in A$. □

The previous results can be lifted to refinement operators. A direct way to combine refinement operators is to run them in parallel, and combine their results with the join operator. Let $f^i([\![\varphi_1]\!])$ and $g^j([\![\varphi_2]\!])$ be under-approximations. Then by Lemma 5.7, the element $f^i([\![\varphi_1]\!]) \sqcup g^j([\![\varphi_2]\!])$ is an under-approximation of $\varphi_1 \wedge \varphi_2$. It is a safe way to combine two under-approximating refinement operators.

However, this "parallel approach" bears a number of weaknesses. Firstly, the two refinement operators only share information at the end of their computations. This is not always efficient, as sharing information *during* the computation might accelerate the convergence [GM03, SS08]. Secondly, the individual approximations contain more elements than their intersection. Especially for under-approximations, sharing information might help to reduce the memory consumption of the approximations. For these reasons, we often benefit from interleaving the refinement operators. However, the properties of sequential composition are not as easily preserved as with the parallel composition.

To illustrate the challenge, consider the two following under-approximating refinement operators over $\mathbb{Z}^\sharp$, for the constraint $x = \infty$:

$$g(a) = \begin{cases} \infty & \text{if } a \text{ is even} \\ a+1 & \text{otherwise} \end{cases} \qquad h(a) = \begin{cases} \infty & \text{if } a \text{ is odd} \\ a+1 & \text{otherwise} \end{cases}$$

For all $a \in \mathbb{Z}^\sharp$, both functions g and h are at a fixed point after two applications at most. Moreover, their fixed points is equal to ∞ which correctly under-approximates the initial constraint. However, the composition $g \circ h$ has no fixed point reachable in a finite number of steps for even numbers, thus it is not a correct under-approximating refinement operator. The issue here is that g and h are in some sense "not compatible" as they disrupt the computation of each other.

The missing property of g and h is monotonicity. Indeed, the composition of monotone refinement operators preserves under-approximations. Intuitively, it implies that the computation of an under-approximation cannot be disrupted by new information external to the refinement operator.

Proposition 5.9 (Preservation of under-approximating refinement). *Let $f : A \to A$ be an under-approximating and monotone refinement operator over an abstract domain A. Let $f^i([\![\varphi]\!])$ be an under-approximation. Then, for all $a \in A$ and $j, k \in \mathbb{N}$ with $j + k = i$, $f^j(f^k([\![\varphi]\!]) \sqcup a)$ is also an under-approximation.*

Proof. We have $f^k([\![\varphi]\!]) \sqcup a \geq f^k([\![\varphi]\!])$. By monotonicity of f, we have $f^k([\![\varphi]\!]) \sqcup a \geq f^k([\![\varphi]\!]) \Rightarrow f^j(f^k([\![\varphi]\!]) \sqcup a) \geq f^j(f^k([\![\varphi]\!]))$. We also have $f^j(f^k([\![\varphi]\!])) = f^i([\![\varphi]\!])$, Therefore, by Lemma 5.8, $f^j(f^k([\![\varphi]\!]) \sqcup a)$ is also an under-approximation since it is greater than $f^i([\![\varphi]\!])$. □

Corollary 5.10 (Composition of under-approximating refinements). *Let $f : A \to A$ and $g : A \to A$ be two under-approximating and monotone refinement operators. Then, $f \circ g$ is under-approximating.*

This corollary shows that the refinement operators $\mathit{refine}_{\text{SEQ}}$ and $\mathit{refine}_{\text{PAR}}$ of *PP* are under-approximating if the propagators are themselves under-approximating and monotone. In fact, we can state an even more powerful corollary.

Corollary 5.11. *Let $f : A \to A$ be a monotone refinement operator under-approximating φ. Then, for any extensive function $g : A \to A$, $f \circ g$ under-approximates φ.*

We can compose an under-approximating refinement operator with any function that freely prunes the domains without hindering the under-approximating property. This is often useful to accelerate the convergence. We will return to this aspect in more details in Section 6.5.

The adequacy of the monotone property has been pushed forwards by different authors, but in slightly different settings. Extensive and monotone refinement operators have been studied in depth by Apt [Apt99, Apt00] for studying the properties of generic propagation engines. What's more, with the addition of idempotence, these functions are at the heart of the concurrent constraint programming (CCP) paradigm [SRP91]. The chief advantages of CCP are modularity and correctness as it guarantees any program written in this paradigm to be closure operators (extensive, monotone and idempotent functions) by syntactic construction. However, idempotence suggests that functions are combined in parallel, the intermediate computation steps are internalized. As we mentioned before, only combining the "end result" might lead to slower convergence. In the context of our work, the key is that monotonicity is a sufficient condition to preserve under-approximation under functional composition. Besides, not all refinement operators are monotone, and we discuss their compositionality conditions later in Section 6.5. We discuss CCP in more detail in Section 8.4.

5.4 Compositionality of over-approximation

We now turn to the composition of over-approximating refinement operators. This section follows the same organization than the previous section, and we start with the composition of over-approximating elements at the end of a refinement chain. We obtain similar results than for under-approximation with the additional notion of *join-preserving* concretization function.

Definition 5.12 (Join-preserving function). Let L and K be lattices. The function $f : L \to K$ is *join-preserving* if $f(a \sqcup b) = f(a) \sqcup f(b)$ for all $a, b \in L$.

Lemma 5.13 (Preservation of over-approximation). *Let $a, b \in A$ two elements respectively over-approximating φ_1 and φ_2. Then $a \sqcap b$ over-approximates $\varphi_1 \land \varphi_2$. Moreover, if γ_A is join-preserving, then $a \sqcup b$ also over-approximates $\varphi_1 \land \varphi_2$.*

Proof. The first part is obtained by duality of Lemma 5.7. We have $\gamma(a) \supseteq [\![\varphi_1]\!]^\flat$ and $\gamma(b) \supseteq [\![\varphi_2]\!]^\flat$. By monotonicity of γ, we have $\gamma(a \sqcap b) \supseteq \gamma(a) \cup \gamma(b) \supseteq [\![\varphi_1]\!]^\flat \cap [\![\varphi_2]\!]^\flat$. The second part is shown as follows. Due to γ being join-preserving, we have $\gamma(a \sqcup b) = \gamma(a) \cap \gamma(b) \supseteq [\![\varphi_1]\!]^\flat \cap [\![\varphi_2]\!]^\flat$. Hence, $\gamma(a \sqcup b)$ over-approximates $[\![\varphi_1 \land \varphi_2]\!]^\flat$. □

Lemma 5.14 (Persistence of over-approximation). *Let $a \in A$ an element over-approximating φ. Then for all $b \in A$, $a \sqcap b$ also over-approximates φ.*

Proof. Obtained by duality of Lemma 5.8. □

Similarly than for under-approximation, combining over-approximating elements give rise to compositionality condition of over-approximating refinement operators. It is obtained from Lemma 5.13 which shows us that $f^i([\![\varphi_1]\!]) \sqcap g^j([\![\varphi_2]\!])$ is an over-approximation, or if γ is join-preserving, that the join of these two elements is also an over-approximation.

In comparison to under-approximation, the join of an over-approximation and an arbitrary element does not necessarily yield an over-approximation. Indeed, the arbitrary element could prune some solutions from the formula being over-approximated. Therefore, we only look at the composition of over-approximating refinement operators. For this purpose, we need an additional property, which guarantees that an over-approximating refinement operator does not prune solution:

$$\forall a \in A,\ \gamma(f(a)) \supseteq \gamma(a) \cap [\![\varphi]\!]^\flat \tag{4}$$

That is, a refinement operator should not remove the remaining solution of φ in any element $a \in A$. This view generalizes the definition of over-approximation, since it also applies to elements that are not initially over-approximation. The following lemma shows that this property is compositional.

Lemma 5.15. *Let f and g be two refinement operators satisfying (4) and over-approximating φ_1 and φ_2 respectively. Then we have $\forall a \in A,\ \gamma((f \circ g)(a)) \supseteq \gamma(a) \cap [\![\varphi_1 \wedge \varphi_2]\!]^\flat$, that is $f \circ g$ satisfies (4).*

Proof. We have $\gamma(g(a)) \supseteq \gamma(a) \cap [\![\varphi_2]\!]^\flat$. Since (4) is defined for all element $a \in A$, thus including $g(a)$, we have $\gamma(f(g(a))) \supseteq \gamma(g(a)) \cap [\![\varphi_1]\!]^\flat$. It follows that $\gamma(f(g(a))) \supseteq \gamma(a) \cap [\![\varphi_1]\!]^\flat \cap [\![\varphi_2]\!]^\flat$. □

The compositionality property follows this lemma.

Proposition 5.16 (Composition of over-approximating refinement). *Let $f : A \to A$ and $g : A \to A$ be two refinement operators satisfying (4) and over-approximating φ_1 and φ_2 respectively. Then, $f \circ g$ is over-approximating.*

Proof. From Lemma 5.15, we know that $f \circ g$ satisfies (4). Moreover, $f \circ g$ is over-approximating since, supposing $[\![.]\!]$ is over-approximating, we have $\gamma((f \circ g)([\![\varphi]\!])) \supseteq \gamma([\![\varphi]\!]) \cap [\![\varphi]\!]^\flat$ and $[\![\varphi]\!] \supseteq [\![\varphi]\!]^\flat$. Therefore, we have $(\gamma \circ (f \circ g) \circ [\![.]\!])(\varphi) \supseteq [\![\varphi]\!]^\flat$. Finally, by induction on $i \in \mathbb{N}$, the refinement operator $(f \circ g)^i \circ (f \circ g)$ preserves over-approximation for all i. □

It follows this proposition that $refine_{\text{SEQ}}$ is an over-approximating refinement operator if the propagators of *PP* satisfy (4). For $refine_{\text{PAR}}$, we also need Lemma 5.13 to show that it is over-approximating, due to the composition by join of the intermediate results.

The results presented in this section allow us to create new refinement operators from existing ones, thus achieving modularity. Indeed, it is easier to check the properties of each refinement operator individually rather than verify approximation properties on their composition as a whole. Moreover, we gave sound arguments for the composition in parallel of refinement operators. This will be useful to prove the correctness of implementation of parallel solving algorithms on multi-core architectures.

6 SEARCH TREE

In the previous section, we introduce abstract domains to represent conjunctive collection of information. For that, we relied on the Hoare lattice to represent set of refinement operators. We now consider the dual case of disjunctive collection of information. That is, if one element in the collection is true, then the approximated formula is satisfiable. Alternatively, if none of the element are true, then the approximated formula is unsatisfiable. We introduce the *search tree* abstract domain which encapsulates disjunctive collections of information using the Smyth lattice construction.

There are at least two reasons to consider the Smyth construction. Firstly, in the case of over-approximation, it is helpful to increase the precision of an abstract domain. Consider the concrete element $\{\{x \mapsto 1.25\}, \{x \mapsto 2.1\}\}$. In the lattice of intervals, it can be over-approximated with $[1.25..\lceil 2.1 \rceil]$, which contains the superfluous elements $\lceil 1.25 \rceil, \ldots, \lfloor 2.1 \rfloor, \lceil 2.1 \rceil$. By considering a collection of intervals, we obtain the more precise over-approximation $\{[1.25..1.25], [\lfloor 2.1 \rfloor..\lceil 2.1 \rceil]\}$.

A second reason to use the Smyth construction is to "complete" some refinement operators to help them reach an under-approximation. For instance, the propagator for the constraint $x > y$ given in Example 5.4 is not under-approximating. Indeed, if we compute the fixed point of this propagator on, for instance, $x = [0..2]$ and $y = [0..2]$, it yields $x = [1..2]$ and $y = [0..1]$. However,

the concrete assignment $\{x \mapsto 1, y \mapsto 1\}$ belonging to this element does not under-approximate $x > y$. Thanks to the Smyth construction, we can split this elements into several parts in order to help the propagator to reach an under-approximation. This is possible if the propagator meets certain conditions that are made clear in Section 6.5.

From the viewpoint of abstract interpretation, the search tree abstract domain is an adaptation to constraint reasoning of the disjunctive completion [CC79]. Both are based on the Smyth order. The disjunctive completion represents disjunctive properties of a program, such as occurring in conditional statements, in order to increase the precision of the approximation. The difference is that the search tree abstract domain refines the elements one by one, whereas the disjunctive completion usually views the set of elements as a whole.

From the viewpoint of constraint programming, the search tree abstract domain encapsulates the propagate-and-search algorithm (Section 3.3) into a refinement operator. In particular, we show that the Smyth lattice is suited to the task of representing search trees. In addition, we generalize propagate-and-search by equipping the Smyth lattice with a *queuing strategy*, which guides the exploration of the search tree. We formally define the concept of queuing strategy, introduce the search tree abstract domain, and then give under- and over-approximation theorems.

6.1 Queuing strategy

Let A be an abstract domain and A^S its Smyth construction. We call an element $Q \in A^S$ a *queue of nodes*[5]. Intuitively, a queue of nodes is the frontier of the search tree that is to be explored. We classify the nodes in a queue into the *unsplittable* elements and the *unknown* elements as follows ($Q \in A^S$):

$$\begin{aligned} \mathit{unsplittable}(Q) &\stackrel{\text{def}}{=} \{a \in Q \mid |\mathit{split}_A(a)| = 1\} \\ \mathit{unknown}(Q) &\stackrel{\text{def}}{=} \{a \in Q \mid |\mathit{split}_A(a)| > 1\} \end{aligned}$$

As we will see later, the elements $a \in Q$ for which $|\mathit{split}_A(a)| = 0$ are automatically removed from the queue by the refinement operator. We manipulate a queue through two functions *pop* and *push*, respectively to extract the next node to process, and to add the next nodes to explore onto the queue. It is a standard way to explore tree-shaped structures in practice. We call this pair of functions a *queuing strategy*.

Definition 6.1 (Queuing strategy). A queuing strategy is a pair of functions (*push*, *pop*) defined as follows:

$$\begin{aligned} &\mathit{push}_\downarrow : A^S \times A^S \to A^S && \mathit{push}_\uparrow : A^S \times A^S \to A^S \\ &\mathit{push}_\downarrow(Q, B) \stackrel{\text{def}}{=} Q \sqcup_H B && \mathit{push}_\uparrow(Q, B) \stackrel{\text{def}}{=} Q \sqcap_S B \end{aligned}$$

$$\begin{aligned} &\mathit{pop} : A^S \to A^S \times A \\ &\mathit{pop}(Q) \stackrel{\text{def}}{=} \begin{cases} (Q \setminus \{a\}, a) & \text{iff } \exists a \in \mathit{unknown}(Q) \\ (Q, \top_A) & \text{otherwise} \end{cases} \end{aligned}$$

The function *push* merges the set of child nodes B into Q. We have two versions of this function in the case where some nodes in B are comparable to some nodes in Q. Firstly, $\mathit{push}_\downarrow$ merges the nodes with the Hoare join, which is defined as $Q \sqcup_H B \stackrel{\text{def}}{=} \mathit{Max}\ (Q \cup B)$. It only keeps the most refined nodes and discard all the others. For instance, if $Q = \{[0..2], [4..6]\}$ and $B = \{[2..2]\}$, we will have $Q \sqcup_H B \stackrel{\text{def}}{=} \{[2..2], [4..6]\}$. In contrast, the over-approximating $\mathit{push}_\uparrow$ function merges the two sets of nodes with the Smyth meet, defined as $Q \sqcap_S B \stackrel{\text{def}}{=} \mathit{Min}\ (Q \cup B)$. Using the previous example, we

[5]Despite the name, this terminology of "queue" does not refer to the queue data structure, but to any collection of nodes.

have $Q \sqcap_S B \stackrel{\text{def}}{=} \{[0..2], [4..6]\}$. The information $[0..1]$ is removed by under-approximation, but it might still contain solutions, and therefore we must keep it with over-approximations.

The function *pop* does not have two versions as it simply extracts one node from the queue. Moreover, *pop* is under-specified as any unknown node can be popped. Depending on the selection of nodes, we obtain various queuing strategies such as depth-first search (DFS), breadth-first search (BFS) or best-first search [Kor93]. Additional information might be required by the queuing strategy in order to select a node. For DFS and BFS, the selection criterion is based on the insertion position of the nodes in the queue. The insertion position can be attached to each node by considering $(A \times \mathbb{N})^S$. However, we observe that this information does not hinder the precision of the approximation, and thus we leave such "control information" implicit in the forthcoming development of the theory.

The *push* function is not an extensive operation on the Smyth lattice, but in cooperation with *pop* and an extensive node processing function, we can retrieve extensiveness. We capture it formally in the next proposition.

Proposition 6.2. *Let A be an abstract domain, and A^S its Smyth lattice. For all extensive functions $f : A^S \times A \to A^S \times A^S$,* i.e., $(Q, \{a\}) \leq f(Q, a)$ *with a component-wise Smyth ordering, the functions*

$$push_{\downarrow} \circ f \circ pop \qquad\qquad push_{\uparrow} \circ f \circ pop$$

are extensive on A^S.

Proof. The first case is $pop(Q) = (Q \setminus \{a\}, a)$ when a is an unknown node in Q. We have $(Q', B) = f(Q \setminus \{a\}, a)$ which is extensive, *i.e.*, $\{a\} \leq_S B$ and $Q \setminus \{a\} \leq_S Q'$. We prove that $push_{\downarrow}(Q', B)$ is also extensive: $Q \leq_S push_{\downarrow}(Q', B) \Leftrightarrow Q \leq_S Max\ (Q' \cup B) \Leftrightarrow \forall y \in Max\ (Q' \cup B),\ \exists x \in Q,\ x \leq y$. To show that, observe we have $Max\ (Q' \cup B) \subseteq Q' \cup B$, hence either $y \in Q'$ or $y \in B$. If $y \in Q'$, then $Q \leq_S Q' \leq_S \{y\}$, so $\exists x \in Q,\ x \leq y$. If $y \in B$, then $\{x\} \leq_S B \leq_S \{y\} \Rightarrow x \leq y$. It proves $Q \leq_S push_{\downarrow}(Q', B)$. The over-approximation case $Q \leq_S push_{\uparrow}(Q', B)$ is proved in the same way because we also have $Min\ (Q' \cup B) \subseteq Q' \cup B$.

The second case is $pop(Q) = (Q, \top_A)$ when $unknown(Q) = \{\}$. We have $(Q', B) = f(Q, \top_A)$ which is extensive. The set B is either equal to $\{\}$ or $\{\top_A\}$. In the case of an empty set, the push function is the identity on Q'. For $\{\top_A\}$, we have $push_{\downarrow}(Q', \{\top_A\}) = \{\top_A\}$ or $push_{\downarrow}(\{\}, \{\top_A\}) = \{\}$, and $push_{\uparrow}(Q', \{\top_A\}) = Q'$. All of which are extensive.

We conclude that both $push_{\downarrow} \circ f \circ pop$ and $push_{\uparrow} \circ f \circ pop$ are extensive operations. □

6.2 Search tree abstract domain

The search tree abstract domain is an adaptation to constraint reasoning of the disjunctive completion of abstract interpretation [CC79]. Both are based on the Smyth order. The disjunctive completion represents disjunctive properties of a program, such as occurring in conditional statements, in order to increase the precision of the approximation. The difference is that the search tree abstract domain refines the elements one by one using a queuing strategy, whereas the disjunctive completion usually views the set of elements as a whole. Now we focus on and define the search tree abstract domain.

Definition 6.3 (Search tree). Let A be an abstract domain and $(push, pop)$ a queuing strategy. Then the search tree construction $ST(A) = \langle A^S, \leq, \sqcup, \sqcap, \bot, \top \rangle$ is an abstract domain with the following operators:

- The lattice operations are inherited from the Smyth lattice (Definition 2.12),
- $\gamma(Q) \stackrel{\text{def}}{=} \bigcup_{a \in Q} \gamma_A(a)$,
- $[\![\varphi]\!] \stackrel{\text{def}}{=} \{[\![\varphi]\!]_A\}$,

- $[\![\varphi_1 \vee \varphi_2]\!] \stackrel{\text{def}}{=} [\![\varphi_1]\!] \sqcap [\![\varphi_2]\!]$ if $[\![\varphi_1 \vee \varphi_2]\!]_A$ is undefined,
- $split(Q) \stackrel{\text{def}}{=} \{s \cup unsplittable(Q), t \cup unsplittable(Q)\}$ where $t \mathbin{\dot\cup} s = unknown(Q)$,
- $refine \stackrel{\text{def}}{=} push \circ (id \times (split_A \circ refine_A)) \circ pop$

The order relation of the Smyth construction allows us to refine nodes already in the queue, either through *refine* or *split*, or to discard nodes, for instance if $split(a) = \{\}$. The concretization of a queue is the union of the concretization of its elements. The interpretation function encapsulates the interpretation of the base domain in a singleton set. Initially, the queue will only contain a single element, before being refined and split. If the base domain does not support disjunction, it can be represented by a queue with two root nodes, one for each component of the disjunction.

The *split* operator divides the unexplored state space into two parts. If only unsplittable elements remain, then this operator maps to a singleton. As we discuss later in Section 8.2, the fact that the search tree is itself an abstract domain, and has a split operator, enables nested combinatorial solving in structures such as $ST(ST(A))$.

The main part of this abstract domain resides in the refinement operator. We first show that *refine* is extensive, thus is indeed a refinement operator.

Lemma 6.4. *$refine_{ST}$ is an extensive function.*

Proof. By Lemma 6.2, it is sufficient to prove that $(id \times (split_A \circ refine_A))$ is extensive. We have $(id \times (split_A \circ refine_A))(Q, a) \leq (Q, a)$, because Q is unchanged, and $split_A \circ refine_A$ is extensive over a by definition, *i.e.*, we have $\{x\} \leq_S (split_A \circ refine_A)(x)$ for any $x \in A$. □

The next lemma shows that the refinement process terminates whenever there is no unknown node to explore anymore.

Lemma 6.5. *For all $Q \in A^S$, $refine^2_{ST}(Q) = Q \Leftrightarrow unknown(Q) = \{\}$.*

Proof. We prove both directions of the equivalence.

(1) $refine_{ST}(Q) = Q \Rightarrow unknown(Q) = \{\}$. Assume $unknown(Q) \neq \{\}$ and Q is a fixed point of $refine_{ST}$. Then $\exists x \in Q$, $|B = split(x)| > 1$, but $push(Q, B) \neq Q$ because $B >_S \{x\}$, thus a fixed point is not reached ↯.
(2) $refine^2_{ST}(Q) = Q \Leftarrow unknown(Q) = \{\}$. By the definition of *pop*, if $unknown(Q) = \{\}$, we have $pop(Q) = (Q, \top_A)$. By Lemma 6.4, $B = (split_A \circ refine_A)(\top_A)$ is extensive, we necessarily have $\{\top_A\} \leq B$, thus $B = \{\}$ or $B = \{\top_A\}$. In the first case, pushing an empty set results in Q, hence a fixed point. The second case results in $\{\top_A\}$ if $push_\downarrow$ (see Proposition 6.2). A second application of $refine_{ST}(\{\top_A\})$ necessarily yields a fixed point.

□

In comparison to the algorithmic formulation, shown in Section 3.3 with `solve`, *refine* is generic w.r.t. the queuing strategy and it supports infinite over-approximation sequence. With `solve`, an infinite sequence prevents the termination of the algorithm. Alternatively, such as explored in [PMTB13], the termination criterion can be embedded in the algorithm. The issue is that the termination criterion depends on the nature of the problem solved. Usual termination criteria include time limit, nodes limit, depth of the search tree or a precision measure on the size of the elements, as it is often the case in continuous domain. In contrast, the refinement operator can be called an unbounded number of times until a user-defined precision is reached. In essence, we delegate the termination problem to the user.

As an example, we now give an abstract domain at the core of continuous constraint programming obtained from the combination of the abstract domains previously defined.

Example 6.6 (Continuous constraint programming). In continuous constraint programming, one is usually interested by finding an over-approximation of the set of all solutions of a logical formula (see, *e.g.*, [BGGP99, CJ09, AR16]). The core structures and algorithms to solve continuous constraint satisfaction problems are captured by the abstract domain $ST(PP(\mathcal{I}(\mathbb{F}^\sharp)))$. In Section 6.4, we study the properties required by propagators to over-approximate the set of solutions.

6.3 Single solution abstract domain

We show in this section a simple refinement operators combination, and how we can use the results of Section 5.3 to prove its correctness. The refinement operator of the search tree abstract domain approximates the entire space of solutions of a formula. It can be modified to search for a single solution, in which case we will refer to the search tree abstract domain as ST_1. Let $f : A^S \to A^S$ be a refinement operator over A^S. We wish to reach a fixed point on the first unsplittable element:

$$
\begin{aligned}
\mathit{refine} &\stackrel{\text{def}}{=} \mathit{stop_on_unsplittable} \circ f \\
\mathit{stop_on_unsplittable}(Q) &\stackrel{\text{def}}{=} \begin{cases} \mathit{unsplittable}(Q) & \text{if } |\mathit{unsplittable}(Q)| \geq 1 \\ Q & \text{otherwise} \end{cases}
\end{aligned}
$$

As *stop_on_unsplittable* only removes elements from Q, we conclude that it is an extensive operation. By Proposition 5.11, it is immediate that $\mathit{stop_on_unsplittable} \circ f$ is under-approximating if f is under-approximating. Further, f does not need to be monotone because the output of *stop_on_unsplittable* is either a fixed point of f, or the output of f itself. Moreover, if split_A satisfies (2), and the fixed point of *refine* is not empty, the formula φ is guaranteed to be satisfiable. Dually, if f is over-approximating, and the fixed point of *refine* is empty, then the problem is proven unsatisfiable. Of course, *stop_on_unsplittable* is not over-approximating as it removes potential solutions, therefore the composition $\mathit{stop_on_unsplittable} \circ f$ is not over-approximating. In the case of finite discrete problems, the underlying abstract domain is both an under- and over-approximation. This situation is ideal since we always know with certainty if the formula is satisfiable or not.

Example 6.7 (Discrete constraint programming). Propagate and search is a core algorithm in many discrete constraint solvers such as GeCode [STL14] and Choco [PFL17], and constraint logic programming systems such as Eclipse [AW07] and GNU Prolog [DAC12]. As an example, we show the abstract domain of such a solver for the very common case where one looks for a single solution to a constraint satisfaction problem. The underlying structures and algorithms are concisely captured by the abstract domain $ST_1(PP(\mathcal{I}(\mathbb{Z}^\sharp)))$. As seen in Section 4, this abstract domain can be adapted to other domains of discourse such as sets. We present in the next two sections the necessary properties on propagators to guarantee that the refinement operator of this abstract domain finds a solution if one exists.

6.4 Compositionality of over-approximation

We give a series of results to design correct under- and over-approximating refinement operators over search trees. We will also apply these results to refine_{ST}. The properties studied are generic with regard to the queuing strategy provided $\mathit{push}_\downarrow$ is used for under-approximation and $\mathit{push}_\uparrow$ is used for over-approximation.

For generality purposes, we consider node processing functions of the form $f : A^S \times A \to A^S \times A^S$. Indeed, $\mathit{push} \circ f \circ \mathit{pop}$ is more general than refine_{ST} which is just an example of refinement operator. It encapsulates both the refinement and split operators in a single function. In order to reuse the definitions given in Sections 5.3 and 5.4, it is useful to extend the concretization function to Cartesian product. For the input of f we have $\gamma((Q, a)) = \gamma_{ST}(Q) \cup \gamma_A(a)$, and for its output, we have $\gamma((Q, B)) = \gamma_{ST}(Q) \cup \gamma_{ST}(B)$. This allows us to state the following proposition.

Proposition 6.8. *Let f be a node processing function satisfying (4). Then $push_{\uparrow} \circ f \circ pop$ is an over-approximating refinement operator.*

Proof. By Lemma 6.2, we know that $push_{\uparrow} \circ f \circ pop$ is extensive. Moreover, $push_{\uparrow}$ is defined as the meet in the Smyth lattice. From Lemma 5.13, the meet operation preserves over-approximation, thus $push_{\uparrow}$ is over-approximating. The function pop is also over-approximating as we have $\gamma(Q) = \gamma(pop(Q))$. Therefore, by Proposition 5.16, the function composition $push_{\uparrow} \circ f \circ pop$ preserves over-approximation. □

We can rely on Proposition 5.16 to show that, given two node processing functions f and g satisfying (4), $(push_{\uparrow} \circ f \circ pop) \circ (push_{\uparrow} \circ g \circ pop)$ is an over-approximating refinement operator. This gives us a correct way to compose over-approximating refinement operators over search trees. Furthermore, we note that the two queuing strategies do not need to be identical, and that each refinement operator can be called several times before the other one is called. For instance, interleaved depth-first search [Mes97] mix breadth- and depth-first search strategies, and could be expressed in this framework.

However, this composition technique is unusual, as we often wish to combine two node processing functions acting on a same node at the same time, and not on separated nodes of the search tree. We obtain finer results by viewing the node processing function as the functional composition $p \circ s \circ r$ of several components:

$$\begin{aligned} r &: A^S \times A \to A^S \times A && \text{(refinement)} \\ s &: A^S \times A \to A^S \times A^S && \text{(split)} \\ p &: A^S \times A^S \to A^S \times A^S && \text{(prune)} \end{aligned}$$

As long as these components satisfy (4), their compositions $p \circ s \circ r$ is over-approximating as well. The pruning component has the purpose to cut off some branches of the search tree, thus it is generally used in the context of under-approximation, that we discuss just next.

6.5 Compositionality of under-approximation

We now discuss the dual case of under-approximation. We saw in the introduction of this section that the propagator of $x > y$ is not an under-approximating refinement operator. Formally, the issue is that the set of fixed points of $[\![x > y]\!]$ do not only yield under-approximations. Nevertheless, after a sufficient number of splits, this propagator eventually generates under-approximations. In cooperation with the search tree and a suited split operator, the under-approximating property (1a) can be recovered. The key is that we do not need to look at all the fixed points of the propagator, but only at its set of unsplittable fixed points. More precisely, if the unsplittable fixed points of a node processing function f are under-approximating, then $push_{\downarrow} \circ f \circ pop$ is under-approximating. In order to formally describe this statement, we extend the notion of unsplittable elements to node processing function as follows:

$$\begin{aligned} unsplittable(Q) &\stackrel{\text{def}}{=} \{a \in Q \mid |(\pi_2 \circ f)(Q \setminus \{a\}, a)| \leq 1\} \\ unsplittable(Q, B) &\stackrel{\text{def}}{=} unsplittable(Q \cup B) \end{aligned}$$

The definition of pop is adapted accordingly as well.

Proposition 6.9 (Eventually under-approximating). *Let $f : A^S \times A \to A^S \times A^S$ be a node processing function approximating φ. For every queue $Q \in A^S$, and element $a = pop(Q)$, if f satisfies:*

$$\forall u \in unsplittable(f(a)),\ \gamma(u) \subseteq [\![\varphi]\!]^{\flat} \tag{5}$$

then the fixed point of $push_{\downarrow} \circ f \circ pop$ is an under-approximation of φ. Moreover, if the fixed point is reachable in a finite number of steps, we say f is eventually under-approximating.

Proof. A fixed point Q of $push_{\downarrow} \circ f \circ pop$ is only reached when $Q = unsplittable(Q)$. If Q is not a fixed point, we must have $(Q', B) = (f \circ pop)(Q)$ with $|B| > 1$ by definition of pop. Now, $push_{\downarrow}(Q', B)$ cannot be equal to Q because $\{a\} \sqcup_H Q < B \sqcup_H Q'$ with $a = \pi_1(pop(Q))$. Indeed, we have $\{a\} <_H B$ since $|B| > 1$, thus $B \neq \{a\}$. Therefore, Q cannot be a fixed point of $push_{\downarrow} \circ f \circ pop$ if it contains unknown elements. □

In order to design an under-approximating refinement operator, we need two things. Firstly, the fixed point of $push_{\downarrow} \circ f \circ pop$ must exist and be reachable in a finite number of steps. This is for example the case if A satisfies ACC; we covered more conditions in Section 5.3. Secondly, as indicated by the previous proposition, the node processing function must decide if unsplittable elements are under-approximations. This last condition generalizes to arbitrary lattices the checking condition of propagators in discrete CSP [Tac09]. In the simpler case of a node processing function of the form $id \times (split \circ refine)$, such as with $refine_{ST}$, Proposition 6.9 boils down to the following condition:

$$split(refine(a)) = \{refine(a)\} \Rightarrow \gamma(refine(a)) \subseteq [\![\varphi]\!]^{\flat}$$

It is simpler because the queue Q is not modified by $split$ or $refine$.

We now study sufficient conditions to compose eventually under-approximating node processing function with other extensive functions. The key is that as long as we have an under-approximating "core" node processing function, we can compose it with other extensive functions while preserving under-approximation. However, unlike in Section 5.3, we cannot rely on monotonicity because the split operator is not necessarily monotone.

Example 6.10 (Non-monotone split operator). Consider for example a split over the store of integer intervals $Store(\mathcal{I}(\mathbb{Z}^{\sharp}))$ which splits on the variable with the largest interval. We develop an example which shows the non-monotonicity of this split operator.

$$\begin{array}{ll}
 & (\{x \mapsto [1..5], y \mapsto [1..6]\} \leq \{x \mapsto [1..5], y \mapsto [1..4]\} \\
\Rightarrow & split(\{x \mapsto [1..5], y \mapsto [1..6]\}) \leq_S split(\{x \mapsto [1..5], y \mapsto [1..4]\})) \\
\Leftrightarrow & \{\{x \mapsto [1..5], y \mapsto [4..6]\}, \{x \mapsto [1..5], y \mapsto [1..3]\}\} \leq_S \\
 & \{\{x \mapsto [1..2], y \mapsto [1..4]\}, \{x \mapsto [3..5], y \mapsto [1..4]\}\} \\
\Rightarrow & \{x \mapsto [1..5], y \mapsto [4..6]\} \leq \{x \mapsto [1..2], y \mapsto [1..4]\} \vee \\
 & \{x \mapsto [1..5], y \mapsto [1..3]\} \leq \{x \mapsto [1..2], y \mapsto [1..4]\} \\
\Rightarrow & [4..6] \leq [1..4] \vee [1..3] \leq [1..4] \\
\Rightarrow & false \quad \lightning
\end{array}$$

Therefore this split operator is not a monotone function.

We introduce different finiteness conditions for non-monotone operators in order to preserve under-approximation under composition. For generality purposes, we can consider the refinement operator $f : A \to A$ instead of the node processing function $np : A^S \times A \to A^S \times A^S$. Nonetheless, the following definitions also apply to node processing functions by considering the refinement operator $push_{\downarrow} \circ np \circ pop$.

Firstly, an under-approximation is defined as $f^i([\![\varphi]\!])$, that is the first element is always an abstract element obtained from a formula. Therefore, it does not guarantee that an under-approximation can be reached from $[\![\varphi]\!] \sqcup a$, for any element $a \in A$. The definition of under-approximation can be extended to take that into account:

$$\forall a \geq [\![\varphi]\!],\ \exists i \in \mathbb{N},\ \gamma(f^i(a)) \subseteq [\![\varphi]\!]^{\flat} \tag{6}$$

Another possibility is to show that γ is injective, that is, each abstract element uniquely describes a concrete element, and thus a single logic formula. In that case, the original definition of under-approximation (1a) is equivalent to (6). However, this is not sufficient to recover compositionality, for instance, g and h in the example above satisfy this condition but are not under-approximating.

The second requirement is to constrain the space on which the refinement operates to satisfy ACC. Of course, this is the case if A satisfies ACC. Otherwise, we can check that the set of all chains generated by f is a subset of A satisfying ACC:

$$f(A) \setminus \mathbf{fp}(f) \text{ satisfies ACC} \tag{7}$$

This condition implies that a function cannot change forever a state along an execution, and must stabilize at some points. We note that this condition is immediately satisfied for discrete constraint solvers which work over finite domains with a finite number of variables. Using this condition, we can retrieve compositionality of non-monotone under-approximating refinement operators.

Definition 6.11 (Compositional refinement operators). An under-approximating refinement operator is *compositional* if it satisfies (6) and (7). An eventual under-approximating node processing function f is *compositional* if $push_{\downarrow} \circ f \circ pop$ is compositional.

Proposition 6.12 (Composition of non-monotone under-approximating refinements). *Let $f : A \to A$ be a compositional refinement operator under-approximating φ. Then, for any extensive function $g : A \to A$, $f \circ g$ under-approximates φ.*

Proof. Condition (7) ensures us that f eventually reaches a fixed point, and condition (6) that the fixed point is under-approximating. Suppose $f \circ g$ generates an infinite chain $x_1 < x_2 < \ldots < x_k < \ldots$, then necessarily we must have an element x_k, $k \in \mathbb{Z}$, such that $f(x_k) = x_k$, otherwise condition (7) is violated. By condition (6), x_k must be an under-approximation since for all $i \in \mathbb{Z}$, $f^i(x_k) = f(x_k)$. □

Under the same conditions, we can combine node processing function with additional prune and refinement operators as follows.

Proposition 6.13 (Composition of eventually under-approximating refinements). *Let $f : A^S \times A \to A^S \times A^S$ be a compositional node processing function eventually under-approximating φ. Then, for any pair of extensive functions $p : A^S \times A^S \to A^S \times A^S$ and $r : A^S \times A \to A^S \times A$,*

$$push_{\downarrow} \circ p \circ f \circ r \circ pop$$

under-approximates φ.

Proof. The function *pop* only extracts nodes when f can split on these nodes. Due to condition (7), f can only split nodes a finite number of times. Hence, regardless of the information added by the functions p and r, each element of the queue must become unsplittable. By König lemma the search tree is finite since each path is finite and each node has a finite number of children. The leaves of the search tree are unsplittable, therefore it is a fixed point of $push_{\downarrow} \circ f \circ pop$, and by condition (6), it must be under-approximating. As under-approximations are persistent (Lemma 5.8), $push_{\downarrow} \circ p \circ f \circ r \circ pop$ under-approximates the formula φ. □

7 OPTIMIZATION PROBLEM

We turn to optimization problems where one seeks the optimal solutions of a formula. The optimality criterion is generally expressed as a function $f(x_1, \ldots, x_n)$ which we seek to minimize or maximize. In the following, we treat the optimization function as a constraint $x = f(x_1, \ldots, x_n)$ where x is the *objective variable* to optimize. Moreover, we focus on minimization problem as the maximization problem can be obtained by duality on the operations involving the objective variable.

Given a logical formula φ, and an objective variable x to optimize, the associated minimization problem is stated by the following formula:

$$\mathbf{best}(\varphi, x) \stackrel{\text{def}}{=} \varphi \wedge \forall y,\ \varphi[x \mapsto y] \Rightarrow \neg(x > y) \qquad \text{with } y \notin FV(\varphi)$$

There exists an objective variable x such that φ is satisfied and none of the other solutions—in which every x is renamed to y so we can compare them—is strictly better. We remark that the equivalence $\neg(x_i > y_i) \Leftrightarrow x_i \leq y_i$ does not always hold when the universe of discourse is a partial order. For instance, with the universe of discourse of sets, we can try to minimize a set variable in which case several incomparable sets might occur. The concrete solutions space $[\![\mathbf{best}(\varphi, x)]\!]^\flat$ of an optimization problem contains a set of incomparable assignments minimizing the objective. Following multi-objective optimization terminology, we call this set the *Pareto front*.

In our framework, an under-approximation of the Pareto front is a subset of the best solutions. It implies that we either find at least one of the best solutions or nothing at all. We are not aware of any solving techniques using under-approximation in this sense, because it is often better to find a non-optimal solution rather than nothing at all. Instead, the optimization methods usually under-approximate the set of all solutions $[\![\varphi]\!]^\flat$ by keeping only the best solutions obtained so far. In contrast, an over-approximation is a superset of the best solutions. It means that we can discard solutions of the problem that are provably not the best. We give some definitions that will be useful to define these two forthcoming optimization algorithms.

Let A be an abstract domain, X be a set of variable, and D be a lattice representing the underlying universe of discourse. We equip A with a pair of projective functions $\lfloor . \rfloor : X \times A \to D$ and $\lceil . \rceil : X \times A \to D^\partial$. Let $a \in A$ and $x \in X$ a variable, we write $\lfloor x \rfloor_a$ the lower bound of x in a, and $\lceil x \rceil_a$ its upper bound. We already informally relied on these functions in Example 5.4. We give several examples of these projective functions on domains introduced in the previous sections:

- Let $a \in \mathbb{Z}^\sharp$, then $\lfloor _ \rfloor_a \stackrel{\text{def}}{=} a$ and $\lceil _ \rceil_a \stackrel{\text{def}}{=} \infty$.
- Let $a = [\ell..u] \in \mathcal{I}(A)$, then $\lfloor _ \rfloor_a \stackrel{\text{def}}{=} \lfloor _ \rfloor_\ell$ and $\lceil _ \rceil_a \stackrel{\text{def}}{=} \lceil _ \rceil_u$.
- Let $a \in \mathit{Store}(A)$, then $\lfloor x \rfloor_a = \lfloor x \rfloor_{a(x)}$ and $\lceil x \rceil_a = \lceil x \rceil_{a(x)}$.
- Let $Q \in ST(A)$, then $\lfloor x \rfloor_Q = \bigsqcap_{a \in Q} \lfloor x \rfloor_a$ and $\lceil x \rceil_Q = \bigsqcap_{a \in Q} \lceil x \rceil_a$.

As an example, consider $Q = \{[0..2], [-1..1], [1..3]\} \in ST(\mathcal{I}(\mathbb{Z}^\sharp))$. We have $\lfloor _ \rfloor_Q = \bigsqcap_{\mathbb{Z}^\sharp}\{0, -1, 1\} = -1$ and $\lceil _ \rceil_Q = \bigsqcap_{(\mathbb{Z}^\sharp)^\partial}\{2, 1, 3\} = 3$.

We present in the next sections two refinement operators encapsulating two optimization algorithms on top of the search tree abstract domain. The first one is based on an under-approximating abstract domain, and is the optimization algorithm found in most discrete constraint solvers. The second one over-approximates the set of best solutions, and is generally found in numerical constraint solvers.

7.1 Under-approximating branch-and-bound

Let A be an under-approximating abstract domain and $ST(A)$ its search tree. An inefficient optimization method is to compute the fixed point of refine_{ST}, and then to keep only the best solutions. There is a better algorithm called *branch-and-bound* (BAB) which tracks the best solutions found so far, and prunes subtrees when they cannot improve at least one of the solutions. We adapt refine_{ST} with two new functions $\mathit{filter_best}_x : A^S \to A^S$ and $\mathit{minimize}_x : A^S \times A \to A^S \times A$. The subscript x in the name of these functions is the objective variable. The first function filters the non-optimal unsplittable elements from the queue. It is defined as follows:

$$\mathit{filter_best}_x(Q) = \mathit{unknown}(Q) \cup \{a \in \mathit{unsplittable}(Q) \mid \forall b \in \mathit{unsplittable}(Q),\ \neg(\lfloor x \rfloor_a > \lfloor x \rfloor_b)\}$$

This function is extensive as it can only remove nodes from Q. It actually computes an antichain on the unsplittable elements, similarly to $Min\ S$ on a set S, but projected on the objective variables only. The second function constrains each new explored node to lead to a better solution, if any.

$$minimize_x(Q, a) = (Q, a \sqcup_A [\![\bigwedge_{b \in B} \neg(x > \lfloor x \rfloor_b)]\!]_A) \qquad \text{with } B = unsplittable(Q)$$

The objective variable of the abstract element a must not be worst than any of the known solutions. In the case that the universe of discourse is totally ordered, the minimization constraint boils down to $[\![x \leq \lfloor x \rfloor_B]\!]_A$, or $[\![x < \lfloor x \rfloor_B]\!]_A$ to find a strictly better solution.

We assemble these two functions together by modifying the refinement operator of the search tree as follows:

$$ua_minimize_x \stackrel{\text{def}}{=} push \circ (id \times (split_A \circ refine_A)) \circ minimize_x \circ pop \circ filter_best_x$$

We notice that the new functions are added in a modular way, as existing functions composing $refine_{ST}$ are left unchanged. We summarize the properties of the refinement operator $ua_minimize_x$, in a slightly more general form, w.r.t. the concrete domain in the next two propositions.

Proposition 7.1. *Let f be a compositional refinement operator eventually under-approximating φ, then*

$$push \circ f \circ minimize_x \circ pop \circ filter_best_x$$

under-approximates φ.

Proof. By Proposition 6.13, $r \stackrel{\text{def}}{=} push \circ f \circ minimize_x \circ pop$ is under-approximating as $minimize_x$ is extensive. Moreover, by Proposition 6.12, $r \circ filter_best_x$ is under-approximating as $filter_best_x$ is extensive. □

Proposition 7.2. *Let f be a compositional refinement operator over-approximating* and *eventually under-approximating φ, then*

$$push_{\uparrow} \circ f \circ minimize_x \circ pop \circ filter_best_x$$

exactly represents $\mathbf{best}(\varphi, x)$.

Proof. We must show that $r \stackrel{\text{def}}{=} push_{\uparrow} \circ f \circ minimize_x \circ pop \circ filter_best_x$ under- and over-approximates $\mathbf{best}(\varphi, x)$. Let Q be $r^i([\![\mathbf{best}(\varphi, x)]\!])$ for any $i \in \mathbb{N}$.

Over-approximation case: $filter_best_x$ only removes an element $a \in Q$ if there is another element $b \in Q$ such that $\lfloor x \rfloor_a > \lfloor x \rfloor_b$. If this is the case, then $\gamma(a) \cap [\![\mathbf{best}(\varphi, x)]\!]^{\flat} = \{\}$, as the best assignment in $\gamma(a)$ has a value of x strictly greater than another assignment in $\gamma(b)$. Therefore, $filter_best_x$ over-approximates $\mathbf{best}(\varphi, x)$. The function $minimize_x$ follows the same reasoning. As the composition of over-approximation preserves over-approximation (by Proposition 5.16), and that $push_{\uparrow} \circ f \circ pop$ is over-approximating by definition, r is over-approximating.

Under-approximation case: By Proposition 7.1, r under-approximates φ. We note that using the over-approximating $push_{\uparrow}$ is necessary to avoid losing solutions, but it does not change the fact r is under-approximating. Suppose Q is a fixed point of r. If r does not under-approximate $\mathbf{best}(\varphi, x)$, then there is an element $a \in Q$ which is a non-optimal solution, *i.e.*, $\lfloor x \rfloor_a > \lfloor x \rfloor_b$ for an element $b \in Q$. In this case, $filter_best_x$ removes a from Q, thus Q was not a fixed point of r $\lightning$.

Therefore, the function r is both an over- and under-approximation, thus exactly represent $\mathbf{best}(\varphi, x)$. □

7.2 Over-approximating branch-and-bound

We generalize to abstract domains a standard branch-and-bound algorithm from the field of *global optimization*, as described in the survey of Araya and Reyes [AR16]. This algorithm corresponds to the computation of an over-approximation of $[\![\mathbf{best}(\varphi, x)]\!]^\flat$. The structure of the algorithm is similar to the under-approximating BAB. In the context of a minimization problem, the main addition is to keep track of an interval $[lb..ub]$ such that the optimal values in $[\![\mathbf{best}(\varphi, x)]\!]^\flat$ are included in this interval. A first over-approximation of this interval, for an objective variable x and a queue Q, is $[\lfloor x \rfloor_Q..\lceil x \rceil_Q]$.

One of the crucial components of an over-approximating BAB is the *upper-bounding* procedure. It consists in finding a feasible point inside an abstract element, and to use this point to prune the upper bound. In other words, the upper-bounding function extracts an under-approximation from an abstract element. As surveyed in [AR16], there exists many upper bounding procedures. A simple upper-bounding consists in extracting the middle point of an element, and checking for its satisfiability. We take a general approach by equipping an abstract domain A with an extensive function $extract : A \to A$. For all $a \in A$, $a \geq [\![\varphi]\!]$, we require $extract(a)$ to under-approximate the initial formula φ, *i.e.*, $(\gamma \circ extract)(a) \subseteq [\![\varphi]\!]^\flat$. This function guarantees that if $split(extract(a)) \neq \{\}$, the projection of the objective variable in $extract(a)$ is a certified upper bound of the problem. We note that any under-approximating refinement operator is an extraction operator as well: $extract \stackrel{\text{def}}{=} refine^i$. When an under-approximation cannot be efficiently found, this function can safely return $\top$.

Due to partial domains of discourse, we consider a set of incomparable upper bounds. The underlying lattice is given by the Hoare lattice $(D^\partial)^H$. Indeed, we can either refine an existing upper bound, or find a new one. Therefore, the branch-and-bound refinement operator works over the lattice $OT = ST(A) \times (D^\partial)^H$.

Now we create two new functions to obtain our over-approximating BAB. The first one constrains every popped node to not be worst than any of the current upper bounds:

$$minimize_x((Q, UBs), a) = ((Q, UBs), a \sqcup_A [\![\bigwedge_{u \in UBs} \neg(x >_D u)]\!]_A)$$

Whenever a popped node has a lower bound that is greater than a known upper bound, we can safely discard this node because it will not improve any upper bound. The second function is the upper-bounding procedure $upper_bounding_x(Q, UBs, a)$, which given a queue Q and a set of upper-bounds UBs, extracts an upper bound u of a and discards all the elements in Q with a lower bound greater than u, following the same principle than in $minimize_x$.

$$upper_bounding_x((Q, UBs), a) = \begin{cases} ((\{c \in Q \mid \neg(\lfloor x \rfloor_c >_D \lfloor x \rfloor_b)\}, UBs \sqcup \{\lfloor x \rfloor_b\}), a) & \text{iff } \{b\} = extract(a) \land \forall u \in UBs,\ \neg(\lfloor x \rfloor_b >_D u) \\ ((Q, UBs), a) & \text{otherwise} \end{cases}$$

When an under-approximation can be found, that is $\{b\} = extract(a)$, we check that the objective value of x in b is not worst than a known upper bound. If these two conditions are fulfilled, we update the set of upper bounds UBs with the new bound $\lfloor x \rfloor_b$. We also remove from Q all the nodes with a lower bound greater than this new upper bound. Now we can assemble these two functions into an over-approximating BAB refinement operator:

$$oa_minimize_x \stackrel{\text{def}}{=} push_\uparrow \circ (id \times split_A) \circ upper_bounding_x \circ (id \times refine_A) \circ minimize_x \circ pop$$

Similarly to the under-approximating BAB, we can prove that $oa_minimize_x$ is an over-approximating refinement operator.

Proposition 7.3. *Let A be an over-approximating abstract domain. Then the refinement operator* $oa_minimize_x$ *over-approximates* $\mathbf{best}(\varphi, x)$.

Proof. By Proposition 5.16, it is sufficient to show that $minimize_x$ and $upper_bounding_x$ are over-approximating. In order to show that, it is useful to show that for each $u \in UBs$, there is no solution $s \in [\![\mathbf{best}(\varphi, x)]\!]^\flat$, such that $s(x) > u$. If that was the case, then s would be less optimal than the solution we found with upper bound u, which is not possible. Therefore, $minimize_x$ preserves over-approximation by interpreting $\neg(x > u)$ for every $u \in UBs$ in an element a. The same reasoning applies to $upper_bounding_x$. It also preserves over-approximation by removing all elements which cannot have a better optimal value x than an existing solution. Therefore, since both functions and A are over-approximating, we conclude that $oa_minimize_x$ preserves over-approximation. □

This algorithm does not seem to be adapted to work with exact abstract domain. The reason is that, for exact solving such as in discrete constraint satisfaction or optimization problems, finding an under-approximation is computationally expensive. Therefore, computing an under-approximation with the function *extract* might be as hard as solving the initial problem.

We have generalized to partial order and proved correct two refinement operators which implement two standard branch-and-bound algorithms. In particular, the over-approximating BAB showcases a synergy between under-approximation, with the function *extract*, and over-approximation. The manipulation of both kind of approximations in a single algorithm is crucial in many fields, such as in integer linear programming where it is routine to relax a discrete problem to its continuous version, thus to compute an over-approximation in order to find quicker an under-approximation (or an exact solution).

8 DISCUSSION AND RELATED WORK

We now informally discuss other abstract domains capturing other solving techniques and problems.

8.1 Existing abstract constraint solvers

Constraint programming solver. This paper finds its roots in earlier work by Benhamou [Ben96] and more specifically Pelleau et al. [PMTB13]. The framework in [PMTB13] only considers over-approximating refinement operators, thus it is mostly useful for continuous constraint programming. A lattice-based constraint programming solver is proposed by Fernández and Hill [FH04] for the domain of indexicals over intervals. Indexicals are a constraint language to construct propagators which automatically fulfil certain properties [VHSD91, VHSD98]. The framework of Apt [Apt99, Apt00] describes propagators over ordered structures. We discussed it in Section 5.2. Our work can be viewed as extending these previous approaches with the formalization of non-monotone refinement functions, and the search tree and optimization problem abstract domains. In particular, search strategies can now be thought as functions, similar to propagators, but over the search tree.

SAT solver. Abstract conflict driven learning (ACDL) [DHK13] is a generalization of the conflict driven clause learning (CDCL) algorithm in SAT solver [SS96] to lattice domains. ACDL and our work share the same foundation, and are both based on abstract interpretation. Both work should be seen as complementary.

On the first hand, ACDL formalizes conflicts learning which is not tackled in the present paper. Conflict learning is treated as an under-approximation operator over $D^\flat \setminus [\![\varphi]\!]^\flat$, that is the set of counter-examples of a formula φ. Therefore, a conflict is an abstract element which does not

admit any solution in φ. Hence, there is a duality between over-approximation used for deduction and under-approximation used for learning. In the present paper, we considered both under- and over-approximation for deduction, and we did not formalize conflict learning. An alternative view of conflict learning in our framework is to view it as a refinement operator over the search tree. As a conflict is a new fact, we can add it conjunctively in any abstract element. Moreover, as a conflict is a general fact, we can add it conjunctively in all abstract elements of a queue.

On the other hand, we developed a theoretical framework which attempts to reduce the gap between theory and practice. We can give three differences with ACDL where our formalism is more practical. Firstly, we do not rely on the concretization function to describe our algorithms, whereas ACDL relies on γ-completeness as a stop criterion. Secondly, their formalization depends on the down-set and up-set completions which we argued in Section 2.2 were not practical. Thirdly, the refinement operator, called *transformer* in their work, is necessarily monotone, which we have shown was not a necessary requirement for all purposes. Once again, this is in order to reduce the gap between theory and solver implementation [ST09]. There are more differences that go in this direction such as their split operator, called an extrapolation operator, which only allows binary branching.

In addition, we discuss two aspects relevant to the lattice-theoretic presentation of ACDL and ours, which might be confusing when reading both works. In abstract interpretation, the more we interpret a program, the more possible values a variable can take. This is dual to constraint reasoning where the more we progress, the less possible values a variable can take. In ACDL, they chose to keep the same order direction than in abstract interpretation, thus the transformers are reductive functions—they go from $\top$ to $\bot$ in the lattice. In contrast, we chose to keep the intuition conveyed by domain theory [Sco82] in which $\bot$ represents the lack of information (the initial state), and as the computation progresses we go upwards in the lattice. By the duality principle of lattices, this does not change any theorems or results, hence it is only a matter of presentation. A second aspect is that ACDL transformers over-approximate the greatest fixed point of some concrete functions. Instead, we chose to explain approximations w.r.t. a concrete element instead of the function computing this concrete element, which would be defined as $f_\varphi \stackrel{\text{def}}{=} \lambda c.c \sqcup^\flat [\![\varphi]\!]^\flat$ in our framework. We believe it removes a small formal indirection which helps in making the definitions more understandable. In terms of fixed point, the refinement operators presented under- or over-approximate the least fixed point of f_φ (it is the greatest fixed point in ACDL due to the order reversal).

Linear programming solver. Abstract interpretation has relied on linear programming techniques since its inception with the polyhedron abstract domain [CH78]. However, there are at least two key differences with linear programming solvers. Firstly, an abstract domain implements additional operators such as the join or meet, which are not commonly studied in linear programming. For this reason, a polyhedron usually relies on a double description method: a constraint representation from linear programming, and a generator representation to implement efficiently the join and order relation. Secondly, linear inequalities over integers are generally relaxed to rational numbers in abstract analyser for efficiency reason [JM09]. For these two reasons, a polyhedron is quite different from an abstract linear programming solver. We give in the next section several hints on using our abstract framework to design an abstract integer linear programming solver.

8.2 Prospective abstract constraint solvers

Mixed integer programming. Mixed integer programming is an optimization program containing linear inequalities with both continuous and discrete variables. We write $\vec{D}$ the set of discrete variables. A standard algorithm for this problem consists in two steps: relaxation and splitting.

First, we relax the entire problem to continuous variables, and compute a solution S of the problem. As the set of integer solutions under-approximates the set of continuous solutions, whenever the variables $\vec{D}$ in S are integer, then S is also a solution of the initial problem. Otherwise, suppose a variable $x \in \vec{D}$ equals to 2.5 in S, then we create two subproblems with the constraints $x \leq 2$ and $x \geq 3$ to force the integrality constraints. The problem is then solved recursively. The search tree abstract domain can be used as a basis for this method. Moreover, thanks to our generalization, the under-approximating BAB algorithm is readily available to find the optimal solution. There are numerous other methods such as cutting planes and column generation, that can be studied from the perspective of abstract domains. The main goal is to investigate if these solving methods are generalizable to abstract domains.

Multi-objective optimization. We often need to optimize several variables at once, for instance to minimize the cost and the total time of a manufacturing process. The set of best solutions in multi-objective optimization forms a partial order, in contrast to a total order in single objective optimization. Therefore, the best solution is not necessarily unique. For instance, consider $\{cost \mapsto 1, time \mapsto 2\}$ and $\{cost \mapsto 2, time \mapsto 1\}$, there is no solution that dominates the other. One of the main algorithms to solve multi-objective optimization problems is branch-and-bound, and it is usually studied over linear programs [PG17]. This multi-objective variant of BAB already fits in the abstract domain we presented in Section 7. The key is that we relied on partially ordered universe of discourse which also include bound sets instead of single bounds. Lifting multi-objective optimization techniques, in particular finding better bounds set, to a lattice-theoretic framework might be valuable to various abstract domains.

Multilevel programming. Multilevel programming is a class of problems where a constraint of the problem is itself a satisfaction or optimization problem. It was first described for two layers, namely bilevel programming, in [BM73]. Recently, multilevel problems were studied in the context of constraint programming [CS14], where a sub-problem is captured as a propagator. These problems can be modelled by nesting Smyth lattices for each layer of the problem: the refinement operator of a layer can call the refinement operator of the layers below to obtain the solutions of the sub-problems. This class of problems also echoes with the concept of deep guards and encapsulated search in constraint logic programming and Oz [SSW94]. Further investigation is required to connect the semantics of these language features to Smyth lattices.

Local search. Local search is a class of incomplete optimization algorithms as they cannot prove optimality or unsatisfiability. Nevertheless, local search is useful for solving large problems where complete methods are not efficient. In a nutshell, a local search algorithm moves from one solution to another, by modifying the value of one or more variables, in order to improve an objective function. In this sense, they can be seen as functions under-approximating the space of a formula. Although the state space explored might be viewed as a lattice, it is difficult to view local search algorithms as extensive functions. One of the reasons is that local search allows us to move to a solution worst than the current one, in order to escape local minima. Moreover, they might reexplore previously encountered states which contradicts the extensive property. For these reasons, it seems challenging to come up with an ordering structure capturing the essence of local search. Nevertheless, some variants of local search such as *large neighbourhood search* [Sha98] relies on complete methods to solve smaller parts of the problem. Alternatively, we could view a local search algorithm as a black box function for computing under-approximations useful to accelerate the convergence of a complete algorithm, with a role similar to the upper-bounding procedure of BAB.

8.3 Combination of abstract constraint solvers

Cooperation among constraint solvers can also be captured as abstract domains. The Cartesian product allows us to compose different abstract domains, but it does not allow to exchange information among domains. The *reduced product* is a Cartesian product equipped with a reduction operator allowing the exchange of information [CC79]. It was recently shown that theories in SMT solvers could be combined using the framework of abstract interpretation with such a reduced product [CCM13]. In the context of our framework, we described how to create various cooperation schemes elsewhere [TMT20].

8.4 Concurrent constraint programming

Concurrent constraint programming (CCP) is a paradigm in which concurrent processes communicate through a shared global store of constraints [SR89, Sar93]. The particularity of this paradigm is to compute with *partial information* expressed as constraints. The write and read primitives are then replaced by the `tell` and `ask` primitives over a constraint store. We have `tell(`c`)` for adding a constraint c into the store and `ask(`c`)` for asking if c can be deduced from the store. Concurrency is treated by requiring the store to grow *monotonically*, *i.e.*, removal of information is not permitted. For example, if the store initially contains $x > 5$, adding $x > 2$ will not change the store and $x < 2$ will fail the whole computation due to contradictory information.

This computational model is deceptively simple. Indeed, CCP and its nondeterministic extension subsume a wide array of paradigms including logic programming, concurrent logic programming, constraint logic programming, and dataflow languages. This generality stems from the "global store of constraints" which is an element of a *constraint system*. CCP is generic in a constraint system, and therefore the paradigm mentionned above can be embedded in CCP by using the right constraint system, *e.g.*, Herbrand constraint system for (concurrent) logic programming, or Kahn constraint system for dataflow programming [SRP91]. One interesting trait of CCP is that the theorems established on CCP programs are valid for every constraint system.

We can view a constraint system as a lattice, and the processes as closure operators (extensive, monotone and idempotent functions) over this lattice. Any CCP programs is guaranteed by syntactic construction to be a closure operator. This implies that the closure property is preserved under composition of CCP programs "for free". In the context of this paper, it is tempting to investigate CCP to write correct by construction under- and over-approximating refinement operators. However, although CCP is a powerful paradigm from the mathematical perspective, it suffers from two practical shortcomings that can be addressed in conjunction with our framework.

Firstly, from our study of refinement operators, we understand that monotonicity is important to compose under-approximating refinement operators, see, *e.g.*, Proposition 5.11. Idempotence is important to guarantee the existence of a fixed point because the image of an idempotent function is also its set of fixed points. However, idempotence and monotonicity are not always *practical*. For idempotence, recall that the parallel composition of the fixed point of refinement operators is often not as efficient as their functional composition (Section 5.3). Therefore, it is important to observe and combine internal steps towards the fixed point. In addition, not all refinement operators are monotone as shown in [ST09] or by the refinement operators over the search tree.

Secondly, the semantics of CCP is formalized for an ideal mathematical representation of the constraint system (the concrete domain) over which it is not always realistic to compute. For instance, let $s^\flat$ be the current concrete store, the operator `ask(`φ`)` is defined as $s^\flat \subseteq [\![\varphi]\!]^\flat$. That is, we can deduce φ from the store if the current solutions of the store are included in the solutions of φ. In other words, adding φ to the store does not further constrain the solutions. Computing the set of solutions of a formula φ might be impossible or too expensive. Actually, in an implementation of

CCP over finite domains, namely cc(fd), they used a relaxed version of the ask operator [VHSD98]. Formally, this can be expressed generically in our framework by viewing the store as an abstract domain, and the ask operator as an under- or over-approximating operator.

The compositional conditions we established along this work might be a basis to extend CCP to abstract domains. We took a step in this direction by designing *spacetime programming* [Tal19], an extension of CCP to lattices, which relaxes idempotence and monotonicity properties of processes. However, the spacetime paradigm does not take into account under- and over-approximation properties, thus much remains to be done in this area.

9 CONCLUSION

We have lifted the components of a discrete and continuous constraint programming solvers to a very general lattice-theoretic framework. In this framework, the data structure of a constraint solver is viewed as a lattice, and the solving algorithm as an extensive function over this lattice. Lattices with extensive functions are packaged together in abstract domains, a concept widely studied in the field of abstract interpretation.

Based on this framework, we have built various constraint solvers in an incremental way. Each component of the solver is isolated in a suited abstract domain, and more expressive abstract domains are derived from more basic ones. This hierarchy of abstract domains captures the domain of variables, propagators, search trees and branch-and-bound optimization algorithms. For the first time, we obtain a framework in which the constraint model and the control aspects are unified.

Our long term project is to design a new programming language in which data structures are lattices and programs are under- or over-approximating refinement operators. We have studied compositional properties of refinement operators which will help us to carry on this task.

REFERENCES

[Apt99] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical computer science*, 221(1-2):179–210, 1999.

[Apt00] Krzysztof R. Apt. The role of commutativity in constraint propagation algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(6):1002–1036, November 2000.

[Apt03] Krzysztof Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[AR16] Ignacio Araya and Victor Reyes. Interval Branch-and-Bound algorithms for optimization and constraint satisfaction: a survey and prospects. *Journal of Global Optimization*, 65(4):837–866, August 2016.

[AW07] Krzysztof R. Apt and M Wallace. *Constraint logic programming using ECLiPSe*. Cambridge University Press, 2007.

[Aze07] Francisco Azevedo. Cardinal: A Finite Sets Constraint Solver. *Constraints*, 12(1):93–129, March 2007.

[Ben96] Frédéric Benhamou. Heterogeneous constraint solving. In *International Conference on Algebraic and Logic Programming*, pages 62–76. Springer, 1996.

[BGGP99] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. Revising hull and box consistency. In *Logic Programming: Proceedings of the 1999 International Conference on Logic Programming*, pages 230–244. MIT press, 1999. https://doi.org/10.7551/mitpress/4304.003.0024.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[Bir67] Garrett Birkhoff. *Lattice Theory*, volume XXV of *AMS Colloquium Publications*. American Mathematical Society, 3rd edition, 1967.

[BLP02] Fabrice Bouquet, Bruno Legeard, and Fabien Peureux. Clps-b—a constraint solver for b. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 188–204, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[BM73] Jerome Bracken and James T. McGill. Mathematical Programs with Optimization Problems in the Constraints. *Operations Research*, 21(1):37–44, 1973. Publisher: INFORMS.

[BTM11] Anicet Bart, Charlotte Truchet, and Eric Monfroy. Verifying a real-time language with constraints. pages 844–851. IEEE, 2015-11.

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN*

symposium on Principles of programming languages, pages 238–252. ACM, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, page 1–12, New York, NY, USA, 1977. Association for Computing Machinery.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CCM13] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. Theories, solvers and static analysis by abstract interpretation. *J. ACM*, 59(6), January 2013.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '78)*, pages 84–96, Tucson, Arizona, 1978. ACM Press.

[CJ09] Gilles Chabert and Luc Jaulin. Contractor programming. *Artificial Intelligence*, 173:1079–1100, 2009.

[CL01] Jason Crampton and George Loizou. The completion of a poset in a lattice of antichains. *International Mathematical Journal*, 1(3):223–238, 2001.

[Cou20] Patrick Cousot. The Symbolic Term Abstract Domain. In *The 14th International Symposium on Theoretical Aspects of Software Engineering*, page 8, 2020.

[CS14] Geoffrey Chu and Peter J. Stuckey. Nested constraint programs. In *International Conference on Principles and Practice of Constraint Programming*, pages 240–255. Springer, 2014.

[DAC12] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU prolog. *Theory and Practice of Logic Programming*, 12(1):253–282, 2012.

[DDD05] Gregoire Dooms, Yves Deville, and Pierre Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 211–225, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Dec03] Rina Dechter. *Constraint Processing*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, 1st edition, 2003.

[DHK13] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract Conflict Driven Learning. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 143–154, New York, NY, USA, 2013. ACM.

[DHK14] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract satisfaction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '14*, pages 139–150, San Diego, California, USA, 2014. ACM Press.

[DMP91] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[DPPR00] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931, September 2000.

[FH04] Antonio J. Fernández and Patricia M. Hill. An interval constraint system for lattice domains. *ACM Trans. Program. Lang. Syst.*, 26(1):1–46, January 2004.

[FPÅ04] Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation: 13th International Symposium, LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*, pages 214–232, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Ger94] Carmen Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. page 22, 1994.

[GJM06] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, volume 141, pages 98–102, 2006.

[GM03] Laurent Granvilliers and Eric Monfroy. Implementing Constraint Propagation by Composition of Reductions. In Catuscia Palamidessi, editor, *Logic Programming*, pages 300–314, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[GMN+18] Ian P. Gent, Ian Miguel, Peter Nightingale, Ciaran McCreesh, Patrick Prosser, Neil CA Moore, and Chris Unsworth. A review of literature on parallel constraint solving. *Theory and Practice of Logic Programming*, 18(5-6):725–758, 2018.

[GMS20] Arnaud Gotlieb, Dusica Marijan, and Helge Spieker. ITE: A Lightweight Implementation of Stratified Reasoning for Constructive Logical Operators. *International Journal on Artificial Intelligence Tools*, 29(03n04):2060006, June 2020.

[GNS+16] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. An Abstract Domain of Uninterpreted Functions. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification,*

Model Checking, and Abstract Interpretation, volume 9583, pages 85–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. Series Title: Lecture Notes in Computer Science.

[Gra78] George A. Gratzer. *General lattice theory*. Number 75 in Pure and applied mathematics : a series of monographs and textbooks. Academic Press, New York, 1978.

[GS04] Carla Gomes and Meinolf Sellmann. Streamlined Constraint Reasoning. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Mark Wallace, editors, *Principles and Practice of Constraint Programming – CP 2004*, volume 3258, pages 274–289. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science.

[Hni03] Brahim Hnich. Function Variables for Constraint Programming. page 158, 2003.

[HS97] Warwick Harvey and Peter James Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *ACSC'97*, volume 19, pages 102–111, 1997.

[IEE19] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*, pages 661–667. Springer, 2009.

[JMSY94] Joxan Jaffar, Michael J Maher, Peter J Stuckey, and Roland HC Yap. Beyond finite domains. In *Principles and Practice of Constraint Programming*, pages 86–94. Springer, 1994.

[Kea87] R. Baker Kearfott. Some tests of generalized bisection. *ACM Transactions on Mathematical Software (TOMS)*, 13(3):197–220, 1987.

[Kor93] Richard E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41 – 78, 1993.

[Kow79] Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.

[KVT20] Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. Deterministic parallel fixpoint computation. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–33, January 2020.

[Lau78] Jean-Louis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29 – 127, 1978.

[Lec09] Christophe Lecoutre. *Constraint networks: techniques and algorithms*. ISTE/John Wiley, Hoboken, NJ, 2009.

[LLLS11] Arnaud Lallouet, Yat Chiu Law, Jimmy HM Lee, and Charles FK Siu. Constraint programming on infinite data streams. In *International Joint Conference on Artificial Intelligence*, pages 597–604, 2011.

[Mes97] Pedro Meseguer. Interleaved depth-first search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, volume 2 of *IJCAI '97*, pages 1382–1387, 1997.

[Min06] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1):31–100, 2006.

[Min17] A. Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages (FnTPL)*, 4(3–4):120–372, 2017.

[Old93] William Older. Programming in CLP(BNR). In *In Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 239–249, 1993.

[PFL17] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[PG17] Anthony Przybylski and Xavier Gandibleux. Multi-objective branch and bound. *European Journal of Operational Research*, 260(3):856–872, August 2017.

[Plo70] Gordon D Plotkin. A note on inductive generalization. *Machine intelligence*, 5(1):153–163, 1970.

[Plo76] G. Plotkin. A Powerdomain Construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[PMTB13] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Verification, Model Checking, and Abstract Interpretation*, pages 434–454. Springer, 2013.

[PTB14] Marie Pelleau, Charlotte Truchet, and Frédéric Benhamou. The octagon abstract domain for continuous constraints. *Constraints*, 19(3):309–337, 2014.

[Raj94] Arcot Rajasekar. Applications in constraint logic programming with strings. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, pages 109–122, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[Rey70] John C. Reynolds. Transformational systems and the algebraic nature of atomic formulas. *Machine intelligence*, 5(1):135–151, 1970.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming*. ACM Doctoral dissertation awards. MIT Press, 1993.

[SBB+18] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Trans. Program. Lang. Syst.*, 41(1), December 2018.

[Sco82] Dana S. Scott. Domains for denotational semantics. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming*, pages 577–610, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[Sco16] Joseph Scott. *Other Things Besides Number: Abstraction, Constraint Propagation, and String Variable Types.* PhD thesis, Acta Universitatis Upsaliensis, Uppsala, 2016. OCLC: 943721122.

[Sha98] Paul Shaw. *Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems*, pages 417–431. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[SKH03] Axel Simon, Andy King, and Jacob M. Howe. Two variables per linear inequality as an abstract domain. In Michael Leuschel, editor, *Logic Based Program Synthesis and Transformation*, pages 71–89, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[Smy78] M. B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23 – 36, 1978.

[SR89] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM, 1989.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. The semantic foundations of concurrent constraint programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 333–352, New York, NY, USA, 1991. ACM.

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[SS08] Christian Schulte and Peter J. Stuckey. Efficient Constraint Propagation Engines. *ACM Trans. Program. Lang. Syst.*, 31(1):2:1–2:43, December 2008.

[SSW94] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in oz. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, PPCP '94, pages 134–150, London, UK, UK, 1994. Springer-Verlag.

[ST09] Christian Schulte and Guido Tack. Weakly monotonic propagators. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 723–730, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[STL14] Christian Schulte, Guido Tack, and Mikael Lagerkvist. *Modeling and Programming with Gecode*, 2014.

[Tac09] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation.* PhD thesis, Saarland University, 2009.

[Tal19] Pierre Talbot. Spacetime Programming: A Synchronous Language for Composable Search Strategies. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP 2019)*, pages 18:1–18:16, New York, NY, USA, 7–9 October 2019. ACM.

[Tar33] Alfred Tarski. Pojęcie prawdy w językach nauk dedukcyjnych. 1933.

[TCMT19] Pierre Talbot, David Cachera, Éric Monfroy, and Charlotte Truchet. Combining Constraint Languages via Abstract Interpretation. In *31st IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2019)*, pages 50–58, Portland, USA, 4–6 November 2019.

[TMT20] Pierre Talbot, Éric Monfroy, and Charlotte Truchet. Modular Constraint Solver Cooperation via Abstract Interpretation. *Theory and Practice of Logic Programming*, 20(6):848–863, 2020.

[VHM95] Pascal Van Hentenryck and Laurent Michel. Newton: Constraint programming over nonlinear real constraints. *Brown University, Providence, RI*, 1995.

[VHMD97] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: a modeling language for global optimization.* MIT press, 1997.

[VHSD91] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Technical report, Brown University, 1991.

[VHSD98] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*, 37(1–3):139–164, 1998.

[ZMM+19] Ghiles Ziat, Alexandre Maréchal, Pelleau Marie, Antoine Miné, and Charlotte Truchet. Combination of Boxes and Polyhedra Abstractions for Constraint Solving. In *The 8th International Workshop on Numerical and Symbolic Abstract Domains (NSAD 2019)*, Porto, Portugal, October 2019.

Received December 2021