Abstract Conflict Driven Learning

Vijay D'Silva
Department of Computer Science
University of California, Berkeley
vijayd@eecs.berkeley.edu

Leopold Haller
Department of Computer Science
Oxford University
leopold.haller@cs.ox.ac.uk

Daniel Kroening
Department of Computer Science
Oxford University
daniel.kroening@cs.ox.ac.uk

Abstract

Modern satisfiability solvers implement an algorithm, called Conflict Driven Clause Learning, which combines search for a model with analysis of conflicts. We show that this algorithm can be generalised to solve the lattice-theoretic problem of determining if an additive transformer on a Boolean lattice is always bottom. Our generalised procedure combines overapproximations of greatest fixed points with underapproximations of least fixed points to obtain more precise results than computing fixed points in isolation. We generalise implication graphs used in satisfiability solvers to derive underapproximate transformers from overapproximate ones. Our generalisation provides a new method for static analyzers that operate over non-distributive lattices to reason about properties that require disjunction.

Categories and Subject Descriptors F.3 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Satisfiability, Conflict Driven Clause Learning, Lattices

1. The Algebraic Essence of Satisfiability Solvers

The performance of solvers for the Boolean satisfiability problem (SAT) has improved at an exponential rate in the last decade. Several factors contribute to these improvements including and elegant algorithm, efficient, architecture-aware implementations of data structures, and heuristics that exploit the non-adversarial nature of practical problem instances. A recent survey of these developments by Malik and Zhang [21] concludes with the following question:

> *"Given its theoretical hardness, the practical success of* SAT *has come as a surprise to many in the computer science community. [...] Can we take these lessons to other problems and domains?"*

This paper presents an approach to instantiating the Conflict Driven Clause Learning algorithm (CDCL) in SAT solvers on new problems. We introduce a lattice-theoretic generalisation of Boolean satisfiability. The *bottom-everywhere problem* is to determine if a function on a lattice maps all elements of the lattice to bottom. Instances of the bottom-everywhere problem include satisfiability for formulae in propositional logic and quantifier-free first order theories, source to target reachability in transition systems, language emptiness for automata, and assertion violation in programs. We show that CDCL solves an instance of the bottom-everywhere problem for certain functions on finite, powerset lattices.

The contribution of this paper is *Abstract Conflict Driven Clause Learning* (ACDCL), a strict, mathematical generalisation of CDCL to lattice-based abstractions. We show that CDCL is a specific technique to combine overapproximations of a greatest fixed point with underapproximations of a least fixed point. We provide a correctness argument for ACDCL under general, lattice-theoretic conditions. These conditions are satisfied by abstract domains used in practice and lead to a new family of analysis procedures. Our work also enables a new understanding of CDCL, discussed next.

Abstract Interpretation Perspectives of CDCL

Many existing lattices used in static analysis lack negation, have meet operations that precisely model conjunction, and join operations that overapproximate disjunction. Precision loss due to joins is often eliminated by enriching a domain or analysis with disjunction. Such enrichment may suffer from *case explosion*, meaning that the number of disjunctive cases to be considered grows infeasibly large as the analysis progresses.

We show that the main data structure in a SAT solver, called a partial assignment, represents elements of a well known abstract domain, and that constraint propagation in SAT solvers is fixed point iteration in this domain. SAT solvers compute fixed points in non-Boolean abstract domains. The conceptual insight of this paper is that learning techniques used by SAT solvers can be viewed as synthesising an abstract transformer for negation. The combination of precise conjunction in the partial assignments domain with imprecise negation provided by learning allow a solver to reason indirectly about disjunction without enumerating cases. We show that abstract domains used in practice satisfy the conditions required to support learning.

A second insight of this paper is that the implication graph construction in SAT solvers is a technique for constructing an underapproximate transformer starting from an overapproximation. An abstract transformer can be viewed as a directed graph in which edges represent transformer application. The inverted graph represents a dual transformer and sets of vertices approximate applications of this dual transformer. In a SAT solver, the graph represents a deduction transformer and its inverse represents an abduction transformer. When lifted to programs, we can start with an overapproximate postcondition transformer and derive an underapproximate precondition transformer.

The insights above have practical ramifications, which we have demonstrated with two practical instantiations of ACDCL. We have instantiated ACDCL with the interval abstract domain to analyse C programs that manipulate floating point variables [14]. Our analyser is more precise that a standard static analyser and more efficient than an SMT solver, if either is run in isolation. We have also instantiated ACDCL in the MathSAT framework to derive an SMT solver for

POPL'13, January 23–25, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1832-7/13/01...$10.00

floating point logic [17]. In both cases, our instantiations avoid case explosion experienced by several competing tools.

Contribution and Content

This paper presents *Abstract Conflict Driven Learning* (ACDL), a procedure for reasoning about functions on Boolean lattices by operating on non-Boolean lattices. Our goal is to present an account of ACDCL that reveals its generality and to present correctness arguments that apply to all instantiations of ACDCL. Towards this end, we make the contributions below.

1. The bottom-everywhere problem, a lattice-theoretic problem which encompasses satisfiability of logical formulae, reachability in transition systems, and assertion violation in programs.
2. A view of CDCL as a procedure for solving a specific instance of the bottom-everywhere problem.
3. A generalisation of CDCL to solve the bottom-everywhere problem by combining greatest fixed point and least fixed point computation.
4. A novel technique for deriving underapproximate transformers from overapproximate transformers, based on a generalisation of implication graphs to abstract domains.
5. Lattice-theoretic soundness and completeness arguments that apply to all instances of ACDCL.

The paper is organised as follows: We introduce the bottom-everywhere problem in Section 2 and discuss several instances of this problem. We apply abstract interpretation to derive over- and underapproximate solutions to the bottom-everywhere problem in Section 3, and combine these approximations in Sections 4 and 5. The generalisation of implication graphs to abstract domains is discussed and illustrated in Section 6.

2. The Bottom-Everywhere Problem

We introduce the bottom-everywhere problem and show that satisfiability of formulae and error reachability in transition systems are instances of this problem. Subsequent sections lift CDCL to solve this problem.

Lattice-Theoretic Terminology A lattice is called *bounded* if it has a greatest element, called *top* and denoted $\top$, and has a least element called *bottom* and denoted $\bot$. A function f on a complete lattice L is *additive* if $f(x \sqcup y) = f(x) \sqcup f(y)$ and is *completely additive* if $f(\bigsqcup X) = \bigsqcup f(X)$. A completely additive function maps $\bot$ to $\bot$. The dual notions are called multiplicative and completely multiplicative. The function f is *reductive* if $f(x) \sqsubseteq x$ for all x and is *extensive* if $f(x) \sqsupseteq x$ for all x. A function is *idempotent* if $f(f(x)) = f(x)$ for all x. A *transformer* is a monotone function on a lattice. An *upper closure* is an idempotent and extensive transformer, and a *lower closure* is an idempotent and reductive transformer. The *pointwise order* $f \sqsubseteq g$ between functions from a set to a poset holds if $f(x) \sqsubseteq g(x)$ holds for all x. The *pointwise meet* of f and g, denoted $f \sqcap g$, where both functions map into a lattice is defined as $\lambda x.\ f(x) \sqcap g(x)$. The pointwise join is similarly defined. The set of transformers on a complete lattice form a complete lattice under the pointwise order. The *De Morgan dual* of a function f on a Boolean lattice is $\tilde{f} \mathrel{\hat{=}} \neg \circ f \circ \neg$. We require the following property of De Morgan duals.

Proposition 1. *If f is a completely additive, reductive function on a powerset lattice, $\tilde{f}$ is a completely multiplicative, extensive function.*

The least fixed point of a transformer f on a complete lattice is denoted $\mathsf{lfp}(f)$ and the greatest fixed point is denoted $\mathsf{gfp}(f)$.

2.1 Bottom and Top Everywhere

The material below is new.

Definition 2. *A function f on a bounded lattice is* bottom-everywhere *if $f(x) = \bot$ for all x. The bottom-everywhere problem is to determine if a function on a lattice is bottom-everywhere.*

A *non-bottom witness* is an element a such that $f(a)$ is not bottom. A non-bottom witness a is minimal if no $b \sqsubset a$ is a non-bottom witness. The *top-everywhere* property and *top-everywhere problem* are similarly defined. The dual notions for the top-everywhere problem are a non-top witness and a maximal non-top witness.

In this paper, we consider the bottom-everywhere problem for completely additive, reductive functions on powerset lattices.

Lemma 3. *A completely additive, reductive function on a powerset lattice is a lower closure.*

Proof. Consider a function $f : \mathscr{P}(S) \to \mathscr{P}(S)$. For each x in S, $f(\{x\})$ is either $\emptyset$ or is $\{x\}$, because f is reductive. Since f is completely additive, $f(X)$ equals $\bigcup f(X)$ for every subset X of S. In particular, $f(X)$ equals $\{x \in X \mid f(\{x\}) \neq \emptyset\}$. It follows that $f(f(X))$ equals $f(X)$. $\square$

Due to Lemma 3 we abbreviate 'completely additive, reductive function' to 'additive closure' for the rest of the paper. Theorem 4 below is a consequence of Lemma 3. It is straightforward to prove but the fixed point view is valuable because an abstraction of a lower closure may not be idempotent, in which case iterating a transformer in the abstract yields strictly more precision than applying it once.

Theorem 4. *The following statements are equivalent for a completely additive, reductive function f on a powerset lattice.*

1. *f is bottom-everywhere.*
2. *$\mathsf{gfp}(f)$ is bottom.*
3. *$\tilde{f}$ is top-everywhere.*
4. *$\mathsf{lfp}(\tilde{f})$ is top.*

The implication from 1 to 2 is straightforward, while the implication 2 to 1 holds because f is a lower closure. The equivalence of 1 and 3 follows by negation and of 3 and 4 due to the closure property and complementation of fixed points.

Next, we show that two well-known problems, satisfiability and error reachability, can be reduced to the bottom-everywhere problem. Neither reduction is mathematically surprising but allows us to think of decision problems rather than function problems.

2.2 Unsatisfiability as Bottom Everywhere

Let *Struct* be a set of structures and *Form* be a set of formulae, and $\models \subseteq$ *Struct* $\times$ *Form* be a satisfaction relation between structures and formulae. If (σ, φ) is in $\models$, we write $\sigma \models \varphi$, and say σ *satisfies* φ, or that σ *is a model of* φ. The details of the structures and the formulae are not relevant for the formalisation. A formula φ is *satisfiable over Struct* if some σ in *Struct* satisfies φ, and is *unsatisfiable* over *Struct* otherwise. We drop the qualifier "over *Struct*". The *satisfiability problem* is to determine if a given formula φ is satisfiable.

We formulate satisfiability in terms of a transformer. The *domain* of structures is $(\mathscr{P}(\mathit{Struct}), \subseteq, \cup, \cap)$. The *model transformer* $\mathit{mod}_\varphi : \mathscr{P}(\mathit{Struct}) \to \mathscr{P}(\mathit{Struct})$ maps a set of structures to the subset containing only models of φ.

$$\mathit{mod}_\varphi(S) \mathrel{\hat{=}} \{\sigma \in S \mid \sigma \models \varphi\}$$

Properties of φ can be expressed as properties of mod_φ. The set of models of φ is $\mathit{mod}_\varphi(\mathit{Struct})$ and φ is unsatisfiable exactly if $\mathit{mod}_\varphi(\mathit{Struct})$ is empty. Observe that mod_φ is completely additive and reductive, hence it is in the scope of problems we consider.

We denote the function $(\neg \circ \mathit{mod}_\varphi \circ \neg)$ as ucmod_φ and call ucmod_φ the *conflict transformer*. The conflict transformer ucmod_φ adds to any set of structures all countermodels of φ. The set of countermodels of φ is $ucmod_\varphi(\emptyset)$ and φ is unsatisfiable exactly if $ucmod_\varphi(\emptyset)$ contains all structures. Observe that $ucmod_\varphi$ is completely multiplicative and extensive.

Theorem 5. *A formula φ is unsatisfiable exactly if mod_φ is bottom-everywhere.*

A non-bottom witness for mod_φ is a set of structures that contains a model of φ. A minimal non-bottom witness is a singleton set containing a model of φ. A non-top witness for $ucmod_\varphi$ is a set of structures strictly contained in $Struct$ that excludes some models of φ. A maximal non-top witness for $ucmod_\varphi$ is a set that contains all structures except one model of φ.

We define two logics for use in examples.

Propositional Logic Let $Prop$ be a finite set of propositional variables. The set of *literals* is $Lit \mathrel{\hat{=}} \{p, \neg p \mid p \in Prop\}$, containing a variable or its negation. A *clause* is a disjunction of literals and a CNF *formula* is a conjunction of clauses. As is common in the SAT literature, we view clauses as sets of literals and formulae as sets of clauses. The set of truth values is $\mathbb{B} \mathrel{\hat{=}} \{\mathsf{t}, \mathsf{f}\}$. Literals, formulae and clauses are interpreted over functions $Asg \mathrel{\hat{=}} Prop \to \mathbb{B}$, from variables to truth values, and are called *propositional assignments*. The entailment relation $\models$ is defined as follows. For each assignment σ and literal l, $\sigma \models l$ exactly if $\sigma(l) = \mathsf{t}$ if l is a variable, and $\sigma(l) = \mathsf{f}$ if l is the negation of a variable. For each clause C, $\sigma \models C$ exactly if $\sigma \models l$ for some literal l in C. For a CNF formula φ, we say $\sigma \models \varphi$ if $\sigma \models C$ for every clause C in φ.

Inequality Logic Inequality logic can express order between variables. Let $Vars$ be a set of first-order variables. The set of difference predicates is $\{x < y \mid x, y \in Vars\}$. The set of inequality literals contains all difference predicates and their negations. We interpret difference formulae over integers. The set of structures $Struct \mathrel{\hat{=}} Vars \to \mathbb{Z}$ consists of functions from variables to integers. A structure σ satisfies a predicate $x < y$ if the inequality $\sigma(x) < \sigma(y)$ holds. For formulae, the entailment relation $\models$ is defined as expected.

2.3 Feasible Traces via Bottom Everywhere

Let $M = (S, T)$ be a transition system where S is a set of states and $T \subseteq S \times S$ is a transition relation. Let S^+ be the set of non-empty sequences of states. A trace is a sequence $\tau = \tau_0, \ldots, \tau_{n-1}$ satisfying that every (τ_i, τ_{i+1}) is a valid transition. A state t is reachable from s if there exists a trace starting in s and ending in t. Given sets of states P and Q, the *feasible trace problem* is to determine if there exists a trace from a state in P to a state in Q.

We formulate the feasible trace problem as a bottom-everywhere problem. The *domain of sequences* is $\mathscr{P}(S^+)$. The *feasible trace transformer* $trace_{P,Q} : \mathscr{P}(S^+) \to \mathscr{P}(S^+)$ maps a set of non-empty sequences of states to the subset containing only traces that start from a state in P and reach a state in Q. We assume τ_n is the last state of τ below.

$$trace_{P,Q}(X) \mathrel{\hat{=}} \{\tau \in X \mid \tau \text{ is a trace}, \tau_0 \in P, \tau_n \in Q\}$$

The set of traces from P to Q has several fixed point characterisations, consolidated by Cousot [6]. Once again, the function $trace_{P,Q}$ is completely additive and reductive.

We denote the function $(\neg \circ trace_{P,Q} \circ \neg)$ as $uctrace_{P,Q}$, and call it the *countertrace transformer*. The set $uctrace_{P,Q}(X)$ contains the set X, as well as all sequences that are not traces, all sequences that start in P but do not lead to Q and all sequences that end in Q but do not start in P.

Theorem 6. *Given a transition system, and sets of states P and Q, there is no trace from P to Q exactly if $trace_{P,Q}$ is bottom-everywhere.*

A non-bottom witness for $trace_{P,Q}$ is a set of sequences containing a trace from P to Q. A minimal non-bottom witness is a singleton set containing a feasible trace from P to Q. A non-top witness for $uctrace_{P,Q}$ is a set of sequences that excludes a trace from P to Q. A maximal non-top witness contains all sequences except one trace from P to Q.

The view of feasible traces as bottom-everywhere lifts to reachability and assertion checking in programs. The definition of the set of traces of a program is well-known and is not recalled here. We directly apply the procedures developed in this paper to reason about programs.

2.4 Reachable States via Bottom Everywhere

We consider reachability problems defined in terms of states rather than over traces. We make such a distinction because the details of lifting CDCL are different for trace-based abstractions and for state-based abstractions. The material recalled here is also required to distinguish CDCL from Cousot's forward-backward iteration [7].

Let $M = (S, T)$ be a transition system where S is a set of states and $T \subseteq S \times S$ is a transition relation. The concrete lattice of states is $\mathscr{P}(S)$. Recall that a transition system defines the two transformers below.

$$post(X) \mathrel{\hat{=}} \{t \in S \mid (s,t) \text{ is in } T \text{ and } s \text{ is in } X\}$$
$$pre(X) \mathrel{\hat{=}} \{s \in S \mid (s,t) \text{ is in } T \text{ and } t \text{ is in } X\}$$

The sets of forward and backward reachable states have standard fixed point characterisations, recalled below. Forward-backward reachable states consist of pairs (X, Y) such that every state in X reaches some state in Y and vice-versa. When applied in an abstract domain, this kind of iteration yields strictly more information than forward analysis or backward analysis in isolation.

$$freach_{P,Q}(X) \mathrel{\hat{=}} X \cap [\mathsf{lfp}\ x.\ (P \cup post(x))] \cap Q$$
$$breach_{P,Q}(X) \mathrel{\hat{=}} X \cap [\mathsf{lfp}\ x.\ (Q \cup pre(x))] \cap P$$
$$fbreach_{P,Q}(X, Y) \mathrel{\hat{=}} [\mathsf{lfp}\ x.\ (P \cap Y) \cup post(x),\ \mathsf{lfp}\ x.\ (Q \cap X) \cup pre(x)]$$

The functions $freach_{P,Q}$ and $breach_{P,Q}$ above are completely additive and reductive. The function $fbreach_{P,Q}$ is completely additive (see Chapter 22 of Cousot's notes [7] for a proof). To see that $fbreach_{P,Q}$ is not reductive, consider $fbreach_{P,Q}(P, Q)$, which will contain the states reachable from P and Q respectively. Component-wise intersection can be used to make the function $fbreach_{P,Q}$ reductive.

Theorem 7. *The following are equivalent, given a transition system and sets of states P and Q.*
1. *No state in Q is reachable from a state in P.*
2. *$freach_{P,Q}$ is bottom-everywhere.*
3. *$breach_{P,Q}$ is bottom-everywhere.*
4. *$fbreach_{P,Q}$ is bottom-everywhere.*

The proof of the first three statements is straightforward. The proof of equivalence to 4 is due to Cousot [7]. A minimal non-bottom witness for $freach_{P,Q}$ is a state in Q that is reachable from P. A minimal non-bottom witness for $breach_{P,Q}$ is a state in P from which a state in Q is reachable.

The function $(\neg \circ freach_{P,Q} \circ \neg)$ maps a set of states X to a set containing X and states that are not reachable from P and states not in Q. Observe that such a function can directly be computed by computing a fixed point using the function $\neg \circ post \circ \neg$. The same applies for $breach_{P,Q}$ and the function $\neg \circ pre \circ \neg$. These dual functions have been implemented in model checkers and used to compute fixed points. Henzinger et al. [19] discuss temporal properties that can be checked with these fixed points, and Cousot and Cousot [11] combine these fixed points with abstraction.

Note that the functions $\mathit{fbreach}_{P,Q}$ and $(\neg \circ \mathit{fbreach}_{P,Q} \circ \neg)$ are different. Intuitively, forward-backward iteration exploits a temporal duality between forwards and backwards analysis. In contrast, ACDCL will exploit a different duality between functions and their De Morgan duals.

3. Abstract Procedures for Bottom-Everywhere

In this section, we apply abstraction to the bottom-everywhere problem. If an overapproximation of a function f on $\mathscr{P}(S)$ is bottom-everywhere, the function f is also bottom-everywhere. If an underapproximation of the dual function $\tilde{f}$ is top-everywhere, the function $\tilde{f}$ is also top-everywhere. We now highlight the overapproximate and underapproximate analysis present in SAT solvers and model checkers. We begin by recalling abstract interpretation.

Abstract Interpretation The key idea of abstract interpretation is to characterise solutions to a problem by a fixed point and derive approximate solutions by fixed point approximation. For convenience, we work in the Galois connection framework. Cousot and Cousot [10] extensively discuss generalisations. These generalisations are required to analyse theories such as linear arithmetic because the lattice of polyhedra is not complete, and for analysis of automata, and trace-based abstractions.

A *Galois connection* between posets $(C, \preccurlyeq)$ and $(A, \sqsubseteq)$, written $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, is a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ satisfying that $\alpha(x) \sqsubseteq y$ holds exactly if $x \preccurlyeq \gamma(y)$ does. The lattice C is called the *concrete domain* and A is called the *abstract domain*. Monotone functions on C are called *concrete transformers* and those on A are called *abstract transformers*. An abstract transformer $\mathit{fo} : A \to A$ *soundly approximates* $f : C \to C$ if the pointwise order $f \circ \gamma \preccurlyeq \gamma \circ \mathit{fo}$ holds.

If $(C, \subseteq)$ is a powerset lattice, an abstract domain $(A, \sqsubseteq)$ is called an overapproximation because it satisfies $x \subseteq \gamma(\alpha(x))$ for all x. If $(A, \sqsupseteq)$ is an abstract domain of $(C, \supseteq)$ it is called an underapproximation of $(C, \subseteq)$ because it satisfies $x \supseteq \gamma(\alpha(x))$. A sound abstract transformer on an overapproximation is called a sound overapproximation and one on an underapproximation is called a sound underapproximation.

The next two notions formalise precision of an approximation. A Galois connection implies there is a maximally precise approximation $\alpha \circ f \circ \gamma$, called the *best abstract transformer*. An abstract transformer fo approximating f is *γ-complete at a* if $f(\gamma(a)) = \gamma(\mathit{fo}(a))$, and is *$\gamma$-complete* if it is γ-complete at every a. If an abstract transformer fo is γ-complete at a, no precision is lost. For example, if post is a concrete successor transformer and apost is a sound abstraction that is γ-complete at an abstract state a, every abstract transition from a also exists in the concrete. In other words, there are no spurious transitions. Giacobazzi and Quintarelli [16] discuss γ-completeness in detail.

Parametric Fixed Points We require the notion of a fixed point above (or below) an element. Let $f : L \to L$ be a monotone function on a complete lattice and a be an element of L. The *greatest fixed point below a* denoted $\mathsf{gfp}_a(f)$ is the greatest fixed point of the function $\lambda x. f(x \sqcap a)$. The *least fixed point above a* denoted $\mathsf{lfp}_a(f)$ is the least fixed point of the function $\lambda x. f(x \sqcup a)$. The *parametric fixed point* functions below map an element x of L to the least fixed point above x and greatest fixed point below x, respectively.

$$\mathsf{plfp}(f) \mathrel{\hat{=}} \lambda x.\mathsf{lfp}_x(f) \qquad \mathsf{pgfp}(f) \mathrel{\hat{=}} \lambda x.\mathsf{gfp}_x(f)$$

Extrapolation and Interpolation We use the terms extrapolation and interpolation for techniques used to accelerate fixed point computation. Figuratively, extrapolation operators move upwards or downwards from an element in a lattice, as illustrated in Figure 1. Interpolation operators move between elements.

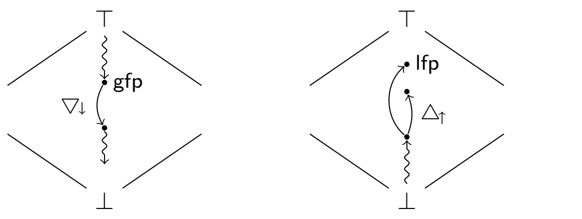

Figure 1. Downwards extrapolation to underapproximate a greatest fixed point, and upwards interpolation to underapproximate a least fixed point. The two operations are not dual.

Let L be a lattice. An *upwards extrapolation* $\triangledown_{\uparrow} : L \to L$ is a unary *extensive* function. We write $\triangledown_{\uparrow} a$ for the application $\triangledown_{\uparrow}(a)$. The dual notion of a *downwards extrapolation* $\triangledown_{\downarrow} : L \to L$ is a unary *reductive* function. The definitions above are based on (but not identical to) those for widening operators without a well-foundedness requirement [10].

A *downwards interpolation* is a binary function $\triangle_{\downarrow} : L \times L \to L$ satisfying that for all x and y in L, the inequality $x \sqsupseteq y$ implies the inequality $x \sqsupseteq \triangle_{\downarrow}(x, y) \sqsupseteq y$. We write interpolation application as $x \triangle_{\downarrow} y$. The standard definition of a *narrowing operator* extends downwards interpolation with a well-foundedness condition. The notions of *upwards interpolation* and dual narrowing, both denoted $\triangle_{\uparrow}$, are dually defined. Note that extrapolation and interpolation are not dual operations, just as widening and narrowing are not dual. Though interpolation is self-dual, we distinguish between upwards and downwards interpolation based on whether the operator is used in a least or greatest fixed point computation.

Downset Completion Downset completion is an operation that enriches an abstraction with disjunction [9]. Consider an abstraction $(A, \sqsubseteq, \sqcup, \sqcap)$ of a powerset lattice $\mathscr{P}(S)$ with concretisation γ. The abstraction A *is disjunctive* if $\gamma(a \sqcup b) = \gamma(a) \cup \gamma(b)$.

A subset Q of A is *downwards closed* if for every x in Q and y in A, $y \sqsubseteq x$ implies that y is in Q. A downwards-closed set is called a *downset*. The smallest downset containing Q is denoted $Q{\downarrow}$, and the downset of a singleton set $\{x\}$ is denoted $x{\downarrow}$. In examples, we denote a downset as the set of its maximal elements. We omit a general discussion of when such a representation is possible. The *downset lattice* over A, written $(\mathscr{D}(A), \subseteq, \cap, \cup)$, is the set of downsets of A ordered by inclusion. Downsets strictly generalise powersets because the downset lattice with respect to the identity relation is isomorphic to $\mathscr{P}(S)$.

The *downset completion* of A is the lattice $\mathscr{D}(A)$ with the abstraction and concretisation functions below. In contrast to the standard treatment, we use downsets as underapproximating abstractions.

$$\begin{aligned} &\gamma_{\mathscr{D}(A)} : \mathscr{D}(A) \to \mathscr{P}(S) \quad && \gamma_{\mathscr{D}(A)}(Q) \mathrel{\hat{=}} \bigcup \{\gamma(x) \mid x \in Q\} \\ &\alpha_{\mathscr{D}(A)} : \mathscr{P}(S) \to \mathscr{D}(A) \quad && \alpha_{\mathscr{D}(A)}(P) \mathrel{\hat{=}} \{x \mid \gamma(x) \subseteq P\} \end{aligned}$$

Consult [9] for proofs that the pairs of functions above form Galois connections and that the domains are disjunctive.

3.1 Abstract Bottom-Everywhere

We apply abstract interpretation to determine if a function is bottom-everywhere. Let $(O, \sqsubseteq, \sqcup, \sqcap)$ be an abstract domain in a Galois connection with $(\mathscr{P}(S), \subseteq, \cup, \cap)$. We denote the abstraction and concretisation functions as α_O and γ_O. We assume that $\gamma_O(\bot)$ is the empty set.

Let $\mathit{fo} : O \to O$ be a sound abstraction of a completely additive, reductive function f. If $\mathsf{gfp}(\mathit{fo})$ concretises to $\bot$, the function f is bottom-everywhere. If $\mathsf{gfp}(\mathit{fo})$ does not concretise to $\bot$, we do not know if f is bottom-everywhere, due to imprecision in the transformer. This intuition is stated below and the proof follows from the basic soundness results of abstract interpretation.

Theorem 8. *If f a completely additive, reductive function on a powerset lattice, $fo : O \to O$ is a sound overapproximation of f, and $\mathsf{gfp}(fo)$ is $\bot$, the function f is bottom-everywhere.*

If we know that *fo* is γ-complete at some element a, we can determine, despite working in an abstraction, that f is not bottom-everywhere. Since *fo* only has to be γ-complete at a single element, the lattice and transformer may still be imprecise.

Proposition 9. *If fo is γ-complete at a and $\gamma(fo(a))$ is not $\bot$, f is not bottom-everywhere.*

The procedure Abstract-non-$\bot$ below takes as input an overapproximate transformer *fo*, an abstract element o and a downwards extrapolation operator $\triangledown\!\downarrow$. In addition, a procedure to check emptiness $\gamma(o) = \emptyset$ is required together with a sufficient criterion for γ-completeness of *fo*. The procedure attempts to determine if *fo* is bottom on all elements below o in the lattice. The output is a pair with the first element being either $\bot$, not $\bot$ or unknown, and the second element representing the last lattice element obtained.

If $\mathsf{pgfp}(fo)(o)$ concretises to $\bot$, the function f is bottom on elements below $\gamma_O(o)$. If *fo* is not bottom on the fixed point o' and is γ-complete at o', we know f is not bottom-everywhere. Neither condition above may hold due to imprecision in the transformer or domain. In this case, downwards extrapolation is used to check if *fo* is also bottom on elements below the fixed point. Unlike standard applications of widening, downwards extrapolation is applied here to improve precision.

Algorithm 1: Abstract Search for a Non-$\bot$ Witness

```
Abstract-non-⊥($fo : O \to O$, $\triangledown\!\downarrow : O \to O$, $o : O$)
    repeat
        $o' \leftarrow o$
        $o \leftarrow o \sqcap fo(o)$
    until $o = o'$ or $\gamma(o) = \emptyset$
    if $\gamma(o) = \emptyset$ then return $(\bot, o')$
    if fo satisfies $\gamma$-completeness criterion at $o$ then
        return (not $\bot$, $o$)
    $d \leftarrow \triangledown\!\downarrow o$
    if $d = o$ then return (unknown, $d$)
    return Abstract-non-⊥($fo$, $\triangledown\!\downarrow$, $d$)
```

Model Search in SAT Solvers

We show that the procedure in Algorithm 1 generalises *model search* in SAT solvers. The abstract lattice contains *partial assignments*, the abstract transformer is called the *unit rule*, the greatest fixed point computation $\mathsf{pgfp}(fo)$ is *Boolean Constraint Propagation* (BCP) and downwards extrapolation is implemented by *decisions*. We briefly elaborate on these points. Further details are in [13].

A partial assignment maps each variable to true, false, or unknown. The set of partial assignments $PAsg \mathrel{\hat{=}} (Vars \to \{\mathsf{true}, \mathsf{false}, \top\}) \cup \{\bot\}$ consists of partial assignments and a unique element $\bot$. In implementations of SAT solvers, the state $\bot$ is indicated by a special conflict flag. Partial assignments form a lattice with respect to the natural pointwise order and are equivalent to the constants lattice or the Cartesian abstraction for Boolean valued variables. The element $\top$ represents a partial assignment in which all variables have undefined values, and $\bot$ represents the empty set. The concretisation function $\gamma : PAsg \to \mathscr{P}(Asg)$ maps $\bot$ to $\emptyset$, and $\pi \neq \bot$ to the set defined below.

$$\gamma(\pi) \mathrel{\hat{=}} \{\sigma \mid \text{ for all } x \in Vars, \pi(x) \neq \top \text{ implies } \pi(x) = \sigma(x)\}$$

Propositional solvers deduce properties about a formula using the unit rule. The unit rule asserts that if a partial assignment is defined on all but one literals in a clause and does not satisfy those literals, it must satisfy the remaining literal to satisfy the formula. For a propositional literal l, we write $\pi \models l$ if $l = x$ and $\pi(x) = \mathsf{t}$ or if $l = \neg x$ and $\pi(x) = \mathsf{f}$. Lattice-theoretically, the unit rule for a clause is a decreasing transformer on the lattice of partial assignments.

$$Unit_C(\pi) \mathrel{\hat{=}} \begin{cases} \bot & \text{for all } l \in C.\ \pi \models \neg l \\ \pi \sqcap \{x \mapsto \mathsf{t}\} & C = C' \cup \{x\}, \pi(x) = \top \text{ and} \\ & \text{for all } l \in C'.\ \pi \models \neg l \\ \pi \sqcap \{x \mapsto \mathsf{f}\} & C = C' \cup \{\neg x\}, \pi(x) = \top \text{ and} \\ & \text{for all } l \in C'.\ \pi \models \neg l \\ \pi & \text{otherwise} \end{cases}$$

Moreover, $Unit_C$ can be characterised as the best abstract transformer for the model transformer mod_C. The unit rule extends to formulae in CNF by taking the pointwise meet of unit rules for each clause in the formula.

$$Unit_\varphi \mathrel{\hat{=}} \bigsqcap_{C \in \varphi} Unit_C$$

Boolean Constraint Propagation (BCP) repeatedly applies the unit rule and computes the fixed point $\mathsf{gfp}(Unit_\varphi)$. A solver makes a *decision* by assuming that some variable, which is unknown in a partial assignment, has a definite value. Mathematically, a decision maps a partial assignment π to $\pi[x \mapsto v]$, where x is unknown in π and v is a truth value. Observe that decisions are applications of a downward extrapolation operator $\triangledown_\downarrow : PAsg \to PAsg$. If all variables are assigned to t or f in π, then $\triangledown_\downarrow \pi = \pi$. Otherwise, $\triangledown_\downarrow \pi$ is $\pi[x \mapsto \mathsf{t}]$ or $\pi[x \mapsto \mathsf{f}]$ for some x such that $\pi(x) = \top$.

Consider the conditionals in Abstract-non-$\bot$. If BCP starts from $\top$ and leads to $\bot$, the formula is unsatisfiable, as returned by the first conditional. A SAT solver terminates if it finds a satisfying assignment. Since assignments are partial assignments, the unit rule applies to them. Observe that $Unit_\varphi(\pi)$ for an assignment is $\bot$ if π does not satisfy φ and is π otherwise. More generally, $\gamma(Unit_\varphi(\pi))$ equals $mod_\varphi(\gamma(\pi))$, so finding a satisfying assignment is a sufficient condition for γ-completeness. Another sufficient condition is that at least one literal in each clause is satisfied by the current partial assignment.

To summarise, the algorithmic content of model search in a SAT solver is an instance of Abstract-non-$\bot$ where $O = PAsg$, $\triangledown_\downarrow$ is propositional decision making, *fo* is given by $Unit_\varphi$ and the γ-completeness check tests whether all variables have been assigned to t or f. Several heuristics and carefully implemented data structures are required to achieve high performance with BCP, but these aspects correspond to optimising the fixed point iteration and abstract domain implementation.

Model Search for Inequalities

We instantiate Abstract-non-$\bot$ for reasoning about inequality formulae. Let Lit be the set of inequality literals. We introduce an inequality abstract domain, which contains only conjunctions of inequalities. Specifically, each element of $(Ineq, \supseteq, \cup, \cap)$, where $Ineq = \mathscr{P}(Lit)$ represents a conjunction of inequalities. We use the superset order because a larger set of constraints represents more constraints and has fewer models. The join is intersection of sets of constraints and meet is union. The empty set represents true and the set of all inequalities is one representation of false. Observe that false has multiple representations because every set containing a predicate and its negation is equivalent to false.

Recall that the concrete domain of structures was $\mathscr{P}(Vars \to \mathbb{Z})$. The concretisation function $\gamma_{Ineq} : Ineq \to \mathscr{P}(Vars \to \mathbb{Z})$ maps a set of constraints to their models.

$$\gamma_{Ineq}(\pi) \mathrel{\hat{=}} \{\sigma \mid \sigma \models P, \text{ for every } P \text{ in } \pi\}$$

We use an example due to McMillan et al. [23] to illustrate Abstract-non-$\bot$ on an inequality formula.

$$\varphi \mathrel{\hat{=}} (a < b) \wedge (a < c) \wedge (b < d \vee c < d) \wedge (d < a)$$

As with propositional logic, we will use the best abstract transformer for a clause, but construct the transformer for a formula in the abstract.

$$amod_{\varphi} \mathrel{\hat{=}} \bigsqcap_{C \in \varphi} \alpha_{Ineq} \circ mod_C \circ \gamma_{Ineq}$$

We apply Abstract-non-$\bot$ to check satisfiability of φ. The sets below represent the evaluation of each clause in the abstract domain.

$$\begin{aligned} amod_{\varphi}(\emptyset) &= \{a < b\} \cup \{a < c\} \cup \emptyset \cup \{d < a\} \\ &= \{a < b, a < c, d < a\} = \pi \\ amod_{\varphi}(\pi) &= \pi \cup \{d < b, d < c\} \\ &\quad \cup (amod_{b<d}(\pi) \cap amod_{c<d}(\pi)) \\ &= Lit \end{aligned}$$

The second step above requires explanation. The best abstract transformer for the singleton clauses when applied to π has the effect of computing the transitive closure of constraints in π. Due to $(a < b)$ and $(d < a)$, the inequality $(d < b)$ is added to π. The conjunction of π with $b < d$, represented by $amod_{b<d}(\pi)$, is unsatisfiable, as is the conjunction of π with $c < d$. The best representation of an unsatisfiable conjunction is the set of all constraints.

Compare the calculation above to solving the same formula with DPLL(T). In DPLL(T), a propositional variable would have been introduced for each predicate, and two solvers, one for propositional logic and one for inequalities would have been required to solve the formula. By instantiating Abstract-non-$\bot$, we can solve the formula using only the abstract domain, which plays the role of a theory solver, and without introducing extra variables. This difference becomes important for large formulae and is the motivation for techniques like *natural domain* SMT [5].

The generalised DPLL technique of McMillan et al. [23], referred to as GDPLL, also solves this formula without introducing propositional variables. Unlike ACDCL, GDPLL makes decisions and uses the shadow rule for conflict analysis to solve the formula.

An instantiation of Abstract-non-$\bot$ over $Ineq$ may use $amod$ as defined above for fo, a decision operator $\triangledown_{\downarrow} : Ineq \to Ineq$ such that $\triangledown_{\downarrow} Ineq = Ineq$ and otherwise $\triangledown_{\downarrow}\pi = \pi \cup \{x < y\}$ such that $x < y$ is not in π. The γ-completeness check for an element π can be performed by determining whether for every clause $C \in \varphi$, the intersection $\pi \cap C$ is non-empty. The check for $\gamma(o)$ being empty amounts to detecting a conjunction of unsatisfiable constraints. If constraints are represented as directed graphs, the emptiness check is implemented by cycle detection.

Feasible Traces The automata-theoretic approach to the feasible traces problem is to construct an automaton representing the traces of a system and the negation of a correctness property, and check if the language of this automaton is empty. This approach can be viewed as indirectly computing the greatest fixed point of $trace_{P,Q}$, where P represents the set of initial states of a system, and Q represents states that violate a property.

3.2 Abstract Top-Everywhere

CDCL is distinguished from several procedures that combine dual reasoning because the procedure used to reason about the top-everywhere problem *is not dual* to Abstract-non-$\bot$. Let $(U, \geqslant, \vee, \wedge)$ be an abstract domain that underapproximates $(\mathscr{P}(S), \subseteq, \cup, \cap)$ with functions α_U and γ_U. We assume that $\gamma_U(\top)$ is S. Let $\tilde{fu} : U \to U$ be a sound abstraction of a completely multiplicative, extensive function $\tilde{f}$, where f is completely additive and reductive. If $\mathsf{lfp}(\tilde{fu})$ concretises to $\top$, then the function f is bottom-everywhere.

The procedure Abstract-non-$\top$ computes an underapproximation of a least fixed point using the upwards interpolation operator $\triangle_{\uparrow}$. Correctness follows directly from known theorems in abstract interpretation and it suffices for us to discuss instantiations of this procedure.

Algorithm 2: Abstract non-$\top$

```
Abstract-non-$\top$($\tilde{fu} : U \to U$, $\triangle_{\uparrow} : U \times U \to U$, $u : U$)
    $c \leftarrow u \triangle_{\uparrow} (u \sqcup \tilde{fu}(u))$
    if $\gamma(c) = \top$ then return ($\top$, $c$)
    if $\tilde{fu}$ satisfies $\gamma$-completeness criterion at $c$ then
        return (not $\top$, $c$)
    if $c = u$ then return (unknown, $c$)
    return Abstract-non-$\top$($\tilde{fu}$, $\triangle_{\uparrow}$, $c$)
```

Conflict Analysis in SAT Solvers

We now show that conflict analysis in SAT solvers is an instance of Abstract-non-$\top$. Consider the downset completion $(\mathscr{D}(PAsg), \subseteq)$ of partial assignments treated as an underapproximation. That is, every set of truth assignments is underapproximated by a set of partial assignments. Conflict minimisation [26] is a technique used by SAT solvers to generalise the reason behind a partial assignment leading to a conflict. Minimisation techniques replace a partial assignment π with partial assignments from which π can be derived by the unit rule. There may not be a unique partial assignment from which π is derived, so downsets of partial assignments have to be considered. We define a transformer for conflict minimisation below.

$$minimise_{\varphi} : \mathscr{D}(PAsg) \to \mathscr{D}(PAsg)$$
$$minimise_{\varphi}(Q) \mathrel{\hat{=}} Q \cup \{\pi \mid Unit_C(\pi) \in Q \text{ for some } C \in \varphi\}$$

Conflict minimisation can dually be viewed as restricted application of resolution.

Example 1. Consider a formula $\varphi \mathrel{\hat{=}} \theta \wedge (x \vee \neg y) \wedge (\neg x \vee y)$ and a partial assignment $\pi = \{x \mapsto \mathsf{true}, y \mapsto \mathsf{true}\}$ such that $\mathsf{pgfp}(Unit_{\varphi})(\pi)$ is $\bot$. Since y can be derived by the unit rule if x is true and x can be derived if y is true, we have $minimise_{\varphi}(\pi) = \{\{x \mapsto \mathsf{true}\}, \{y \mapsto \mathsf{true}\}\}$. $\lhd$

There may be many ways to minimise a conflict but generating all minimisations exhausts solver memory. To avoid representing sets of conflicts, a SAT solver usually chooses one conflict. In Example 1, each set in $\Pi = \{\{x \mapsto \mathsf{true}\}, \{y \mapsto \mathsf{true}\}\}$ can be used. We model this step with a choice function $choice$: $\mathscr{D}(PAsg) \times \mathscr{D}(PAsg) \to \mathscr{D}(PAsg)$. The choice function ensures that the effort of generalising a conflict is not lost, and satisfies $\Pi \sqsubseteq choice(\Pi, \Pi') \sqsubseteq \Pi'$, where Π and Π' are downsets. Observe that $choice$ is an upwards interpolation.

We emphasise that the soundness of the procedures in this section follows from standard results in abstract interpretation.

Over programs, instances of Abstract-non-$\top$ are procedures that underapproximate least fixed point computations. For example, the set of all counterexamples leading to an error can be defined by a least fixed point. Counterexample analysis techniques usually underapproximate this set by heuristically choosing one, or some subset of counterexamples.

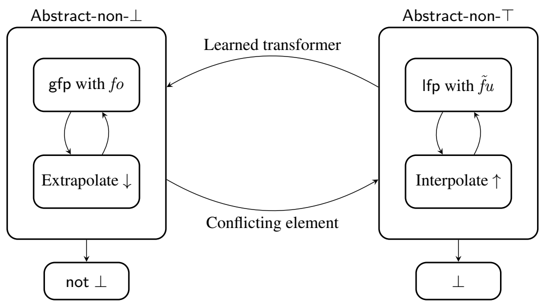

Figure 2. Abstract Conflict Driven Learning

4. Abstract Conflict Driven Learning

The procedures in the previous section were treated separately. The CDCL algorithm in modern solvers combines Abstract-non-$\bot$ and Abstract-non-$\top$ as summarised in Figure 2. Rather than return unknown from Abstract-non-$\bot$, information from the fixed point computation drives Abstract-non-$\top$. If Abstract-non-$\top$ produces inconclusive results, Abstract-non-$\bot$ can still "learn" information about the conflict for the greatest fixed point computation. This section makes this combination precise.

The *Abstract Conflict Driven Learning* procedure (ACDL) is shown in Algorithm 3. The procedures alternates runs of Abstract-non-$\bot$ and Abstract-non-$\top$. Communication between the two procedures is achieved using two functions, $\tilde{fou}$ and $learn$. Conflicting elements are transferred from Abstract-non-$\bot$ to Abstract-non-$\top$ using a function $\tilde{fou}$. We require that this function soundly underapproximates $\tilde{f}$, i.e., that $\tilde{f} \circ \gamma_O \subseteq \gamma_U \circ \tilde{fou}$. A natural choice for this transformer is to compute the composition $\alpha_U \circ \gamma_O$, which maps an abstract element in O to its best underappproximation in U. In the other direction, a transformer is learnt, as discussed below.

Algorithm 3: Abstract Conflict Driven Learning

```
ACDL($fo : O \to O$, $\triangledown_\downarrow : O \to O$,
     $\tilde{fu} : U \to U$, $\triangle_\uparrow : U \times U \to U$,
     $\tilde{fou} : O \to U$, $learn : U \to (O \to O)$)
  loop
    $(s, o) \leftarrow$ Abstract-non-$\bot$($fo$, $\top$, $\triangledown_\downarrow$)
    if $s \neq \bot$ then return $(s, o)$
    $u \leftarrow \tilde{fou}(o)$
    $(s, u) \leftarrow$ Abstract-non-$\top$($\tilde{fu}$, $u$, $\triangle_\uparrow$)
    if $s = \top$ then return $(\bot, u)$
    $fo \leftarrow fo \sqcap learn(u)$
```

Best Learning Transformer If Abstract-non-$\bot$ derives $\bot$ starting from o, to determine if f is bottom at a concrete element c, it suffices to check if f is bottom at $c \cap \neg\gamma_O(o)$, because f is reductive. If Abstract-non-$\bot$ is invoked on an abstract element a, it suffices to check if fo is bottom on elements below $\alpha_O(\neg\gamma_O(o) \cap \gamma_O(a))$. The procedure Abstract-non-$\top$ is used to generalise an element that leads to a conflict but it operates in an underapproximating domain. Thus, we need to learn information about overapproximate elements after generalising within an underapproximation.

The *best learning transformer* is a function that takes elements u and o and subtracts u from o.

$$learn : U \times O \to O \qquad learn_u(o) \hat{=} \alpha_O(\gamma_O(o) \cap \neg\gamma_U(u))$$

Theorem 10. *Let fo be a sound overapproximation of a completely additive, reductive transformer f and let $\tilde{fu}$ be a sound underapproximation of the De Morgan dual $\tilde{f}$ of f. If* $\mathsf{pgfp}(fo)(o)$ *represents the empty set, then the transformer*

$$fo \sqcap learn_u, \text{ where } u \text{ is } \mathsf{plfp}(\tilde{fu})(\alpha_U(\gamma_O(o)))$$

is a sound overapproximation of f.

Theorem 10 strictly generalises the notion of learning from SAT solvers to every instance of the bottom-everywhere problem.

We give an example of a simple, sound learning procedure supported by all lattices. An element that leads to $\bot$ is *tabu*, in the sense of tabu search. Tabu learning defines a sound learning transformer $Tabu : U \times O \to O$.

$$Tabu_u(o) \hat{=} \begin{cases} \bot & \text{if } \gamma(o) \subseteq \gamma(u) \\ o & \text{otherwise} \end{cases}$$

The tabu rule checks whether search has entered a region that is known to map to $\bot$.

Theorem 11 (Soundness)**.** *If* ACDL *returns* not $\bot$*, then f is not bottom everywhere. If it returns $\bot$, then f is bottom everywhere.*

Proof. Assume the algorithm returns not $\bot$. Then Abstract-non-$\bot$ returned (not $\bot, o$). Then fo is γ-complete at o, $\gamma_O(o) \neq \bot$ and $fo(o) = o$. By γ-completeness, we then conclude that f is not bottom at $\gamma_O(o)$.

Assume the algorithm returns $\bot$. Then Abstract-non-$\top$ returned $\top$. We denote by u the initial value of u in Abstract-non-$\top$, and by u' the return value with $\gamma_U(u') = \top$. It is an invariant of the algorithm that fo is a sound overapproximation of f. Hence, whenever Abstract-non-$\bot$ returns $(\bot, o)$ it holds that $f(\gamma_O(o)) = \bot$. It holds that $u = \alpha_U \circ \gamma_O(o)$ for such an o. By duality, we have that $\tilde{f}(\bot) \supseteq \gamma_O(o) \supseteq \gamma_U(u)$. Abstract-non-$\top$ returns $u' \sqsupseteq u$ such that $\gamma_U(u') = \top$. By soundness of abstract interpretation it holds that $\gamma_U(u') = \tilde{f}(\gamma_U(u)) = \top$. Since $\tilde{f}$ is an upper closure operator with $\tilde{f}(\bot) \supseteq \gamma_U(u)$ and $\tilde{f}(\gamma_U(u)) = \top$, it follows by idempotence that $\tilde{f}(\bot) = \top$. Dually, it follows that $f(\top) = \bot$. Therefore, f is bottom everywhere. ☐

5. Abstract Conflict Driven Clause Learning

The previous section showed that the notion of learning is very general. Clause learning is a specific form of learning in which abstract transformers are implicitly represented by clauses. Since model search in SAT solvers is driven by the unit rule, clause learning can be viewed as learning unit rule transformers. We present a generalised unit rule that lifts clause learning to richer lattices. Abstract Conflict Driven Clause Learning (ACDCL) is a strict generalisation of the CDCL algorithm in SAT solvers. In § 3, we demonstrate that propositional solvers operates over the overapproximate partial assignments domain $PAsg$ and its underapproximate downset completion $\mathscr{D}(PAsg)$. We present ACDCL as a variant of the ACDL procedure presented in the previous section operating over an overapproximation O and its downset completion $\mathscr{D}(O)$.

5.1 Generalising Clause Learning

To understand the lattice-theoretic essence of clause learning, it is useful to compare the unit rule with tabu learning. Consider a clause $C = p \vee \neg q$ and the partial assignment $\pi = \{p \mapsto \mathsf{f}, q \mapsto \mathsf{t}\}$, which contains no satisfying assignment to C. Consider π' strictly greater than π. We have that $Tabu_\pi(\pi') = \pi'$. However, if π' is the partial assignment $\{p \mapsto \mathsf{false}\}$, by unit rule application we have $Unit_C(\pi') = \{p \mapsto \mathsf{false}, q \mapsto \mathsf{false}\}$. The tabu rule only drives search away from elements where f is bottom. In contrast, if f is

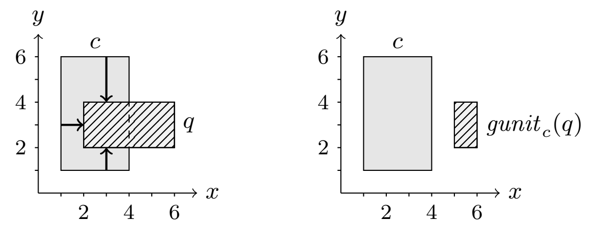

Figure 3. Generalised unit rule $gunit_c$ for intervals

"almost bottom" on π', the unit rule drives the search away from the part of π' that leads to bottom.

Example 2. We illustrate a generalised unit rule for the abstract domain of intervals over integers. Assume an abstract transformer *fo* is $\bot$ on the interval c in Figure 3. We design a generalised unit rule that maps the shaded interval q in the figure to the interval $gunit_c(q)$, which drives the search to the portion of q not known to lead to $\bot$. In what follows, we write an interval constraint such as $5 \leq x \leq 7 \wedge -\infty \leq y \leq \infty$ as $\{x \mapsto [5,7]\}$.

The complement of the interval $c = \{x \mapsto [1,4], y \mapsto [1,6]\}$ is not an interval. However, c is the intersection of the one-way infinite intervals $\{x \mapsto [1,\infty]\}$, $\{x \mapsto [-\infty,4]\}$, $\{y \mapsto [1,\infty]\}$ and $\{y \mapsto [-\infty,6]\}$. The complement of each of these intervals is an interval, and the set of complements can be viewed as a clause containing $\{x \mapsto [-\infty,0]\}$, $\{x \mapsto [5,\infty]\}$, $\{y \mapsto [-\infty,0]\}$ and $\{y \mapsto [7,\infty]\}$. The meet of $q = \{x \mapsto [2,6], y \mapsto [2,4]\}$ with each element of this generalised clause is bottom for all elements except $\{x \mapsto [5,\infty]\}$, so we only consider $\{x \mapsto [5,6]\}$. In this way, we generalise the propositional unit rule to new domains. $\lhd$

We point out the similarities to the propositional unit rule. Every interval is the intersection of one-way infinite intervals, just as a partial assignment is the conjunction of literals. One-way infinite intervals, like propositional literals, have complements. The complement of a partial assignment is a clause, and the complement of an interval can be represented as the disjunction of one-way infinite intervals. The rest of this section lifts the unit rule to new domains.

Complementable Decompositions This section deals with decompositions of lattice elements into elements which cannot be further decomposed, called *irreducibles*. An element m is *meet irreducible* if $x \sqcap y = m$ implies that $x = m$ or $y = m$. The set of meet irreducibles of a lattice A is $\mathcal{M}_A$. A function $mdc : A \to \mathscr{P}(\mathcal{M}_A)$ is a *meet decomposition* if, for all a, the set $mdc(a)$ is finite, and $\bigsqcap mdc(a) = a$. A meet decomposition is *irredundant*, if for any $a \in A$ and $b \in mdc(a)$ it holds that $\bigsqcap mdc(a) \setminus \{a\} \neq a$. Most abstract domains used in practice have an obvious unique irredundant meet decomposition (e.g., partial assignments decompose into partial assignments with only one variable taking a Boolean value, intervals and octagons decompose into sets of half-spaces). A number of examples of possible abstract domains with complementable decompositions are given in Figure 4. Complementable decompositions are not limited to numeric abstractions. For example, the depicted control-flow abstraction abstractly represents a set of traces in terms of control-flow branches.

An element a of an abstract domain A has a *precise complement* if there exists an element $\overline{a}$ such that $\gamma(a) = \neg\gamma(\overline{a})$. A has *complementable meet irreducibles* if every element $m \in \mathcal{M}_A$ has a precise complement $\overline{m} \in \mathcal{M}_A$. A domain A admits *complementable meet decompositions* if A has complementable meet irreducibles and a meet decomposition.

Generalised Unit Rule An abstraction A with complementable meet decompositions admits a generalised unit rule $gunit : A \times A \to A$. In contrast to the propositional unit rule, which is defined with respect to a clause, we define the generalisation with respect to elements that lead to $\bot$. This is only a difference of presentation, since the negation of a clause is a partial assignment on which $Unit_\varphi$ is bottom.

$$gunit_c(a) \mathrel{\hat{=}} \begin{cases} \bot & \text{for all } m \in mdc(c).\ a \sqsubseteq m \\ a \sqcap \overline{n} & mdc(c) = M \cup \{n\} \\ & \text{ and for all } m \in M.\ a \sqsubseteq m \\ a & \text{otherwise} \end{cases}$$

If $gunit_c(a)$ returns $a \sqcap \overline{n}$ as above, we call $\overline{n}$ a *unit irreducible*. The transformer *gunit* can be lifted to a learning transformer $gunit : O \times U \to O$, where $U = \mathscr{D}(O)$. For a downwards-closed set S, we define $gunit_S \mathrel{\hat{=}} \bigsqcap_{c \in S} gunit_c$. We skip the proof that *gunit* is a sound learning transformer. We instantiate abstract conflict driven learning with *gunit* to derive *Abstract Conflict Driven Clause Learning*.

5.2 An Abstract Backjumping Algorithm

Abstract conflict driven learning is sound. To achieve completeness, we need to ensure that learning drives the Abstract-non-$\bot$ procedure to explore new regions of the abstract lattice. The abstract backjumping algorithm appearing next generalises the backjumping search of propositional CDCL. The abstract procedure in Algorithm 4 operates over an overapproximate domain O with complementable decompositions, and its underapproximate downset completion $\mathscr{D}(O)$. Learning refines the transformer *fo* with the generalised unit rule *gunit*.

Algorithm 4: Backjumping and Clause Learning

```
ACDCL($fo$: $O \to O$,
      $\triangledown\!\downarrow$: $O \to O$,
      $\tilde{fu}$: $\mathscr{D}(O) \to \mathscr{D}(O)$,
      $\triangle\!\uparrow$: $\mathscr{D}(O) \times \mathscr{D}(O) \to \mathscr{D}(O)$)
    Initialize $\rho \in O^*$ to singleton sequence $\top_O$
    repeat
        $(s, \rho) \leftarrow$ Abstract-non-$\bot^*$($fo$, $\rho$, $\triangledown\!\downarrow$)
        if $s \neq \bot$ then return $(s, \bigsqcap \rho)$
        $u \leftarrow \{\bigsqcap \rho\}$
        $(s, \{c\}) \leftarrow$ Abstract-non-$\top$($\tilde{fu}$, $u$, $\triangle\!\uparrow$)
        $l \leftarrow gunit_c$
        $\rho \leftarrow$ backjump($l$, $fo$, $\rho$)
        $fo \leftarrow fo \sqcap l$
    until $\rho$ is empty
    return not $\bot$

Abstract-non-$\bot^*$($fo : O \to O$, $\triangledown\!\downarrow : O \to O$, $\rho : O^*$)
    $o \leftarrow \bigsqcap \rho$
    repeat
        $o' \leftarrow o$; $o \leftarrow fo(o) \sqcap o$
    until $o = o'$ or $\gamma(o) = \emptyset$
    if $\gamma(o) = \emptyset$ then return $(\bot, \rho)$
    $\rho \leftarrow \rho \cdot o$
    if $fo$ satisfies $\gamma$-completeness criterion at $o$ then
        return (not $\bot$, $\rho$)
    $d \leftarrow \triangledown\!\downarrow o$
    if $d = o$ then return (unknown,$o$)
    return Abstract-non-$\bot^*$($fo$, $\rho \cdot d$, $\triangledown\!\downarrow$)
```

The algorithm uses a modification of Abstract-non-$\bot$ to record transformer application on a stack ρ. Concatenation of two stacks

	INTERVALS	OCTAGONS	EQUALITY
Concrete Domain	$\mathscr{P}(V \to \mathbb{Z})$	$\mathscr{P}(V \to \mathbb{Z})$	$\mathscr{P}(V \to \mathbb{Z})$
Abstract Elements	$(l \leq x \leq u, \ldots)$	$(\pm x \pm y < c, \ldots)$	$(x = y, w \neq z, \ldots)$
Meet Irreducibles	$(x \leq 4)$	$(x + y < 1)$	$(x \neq y)$
Complemented Irreducible	$(5 \leq x)$	$(-x - y < 0)$	(x = y)

	ARRAY ABSTRACTION	SET ABSTRACTION	CONTROL-FLOW ABSTRACTION
Concrete	$\mathscr{P}(V \to (\mathbb{N} \to \mathbb{Z}))$	$\mathscr{P}((V \to D) \times (SV \to \mathscr{P}(D)))$	$\mathscr{P}(\mathit{Traces})$
Abstract Elements	$(x \preceq \lambda i.c_x, y \npreceq \lambda i.c_y, \ldots)$	$(x \in S, y \notin R, \ldots)$	$(l_i \to \texttt{else}, \ldots)$
Meet Irreducibles	$x \preceq \lambda i.4$	$x \in Q$	$(l_1 \to \texttt{else})$
Complemented Irreducibles	$x \npreceq \lambda i.4$	$x \notin Q$	$(l_1 \to \texttt{if})$

Figure 4. Complementable Decompositions. The set V represents program variables.

ρ and ρ' is denoted by $\rho \cdot \rho'$. The difference to Abstract-non-$\bot$ is that after learning backjumping resets the procedure to a state before the element that is negated by the learning transformer. A call to backjump(l, fo, ρ) returns a non-empty prefix ρ' of ρ such that $\gamma(\mathit{fo}'(\bigsqcap \rho')) \neq \bot$ where $\mathit{fo}' = l \sqcap \mathit{fo}$. If no such prefix exists, the empty sequence is returned, indicating that *fo* is bottom-everywhere.

Progress and Precision If we disregard extensions such as restarts and forgets, propositional CDCL never generates the same clause twice. The number of clauses over a fixed set of literals is finite, so a CDCL solver makes constant progress and necessarily terminates. The procedure ACDCL makes *progress* if each iteration of the outer loop decreases some well-founded order. CDCL solvers do not return unknown. Decisions refine partial assignments until either $\bot$ or a satisfying assignment is obtained.

In propositional solvers, decisions are made by computing a greatest lower bound between the current element and a meet irreducible, that is, a partial assignment in which only one variable is not $\top$. Decisions never directly cause a conflict. We generalise these conditions below. A *meet irreducible extrapolation* is a downwards extrapolation operator $\triangledown_{\downarrow} : O \to O$ satisfying two conditions.

1. $\triangledown_{\downarrow} o$ equals $o \sqcap m$ for some meet-irreducible m.
2. If $\gamma_O(o)$ is not $\bot$, $\gamma_O(\triangledown_{\downarrow} o)$ is not $\bot$.

The precision requirement given next ensures that a sequence of decisions eventually leads to an abstract element that can be analysed without loss of precision. A meet-irreducible extrapolation is *precise* if whenever o is equal to $\triangledown_{\downarrow} o$, the function *fo* is γ-complete at o with respect to f.

The progress condition in a propositional solver is that learning causes model search to deduce new information. The generalised notion we use is that of *asserting backjumps*, which ensures that the procedure navigates away from the element that most recently led to $\bot$. A backjump function is *asserting* if whenever backjump(l, fo, ρ) returns ρ', then ρ' is the empty sequence or $l(\bigsqcap \rho')$ is strictly smaller than $\bigsqcap \rho'$. This condition ensures that backjumping drives the search into a new part of the search space.

Lemma 12. *In some non-final iteration of the* ACDCL *main loop, let ρ and ρ' be, respectively, the stack before and after backjumping. If the backjumping function is asserting, then after learning, it holds that $\mathit{fo}(\bigsqcap \rho')$ and $\bigsqcap \rho$ are incomparable.*

Proof. Let o and o' be, respectively, $\bigsqcap \rho$ and $\bigsqcap \rho'$, and let $\{c\}$ be the element returned by Abstract-non-$\top$. It holds that $c \sqsupseteq o$. Let $\mathit{gunit}_c(o') = o' \sqcap \overline{m}$ where $\overline{m}$ is the unit irreducible and therefore after learning $\mathit{fo}(o') \sqsubseteq \overline{m}$. It holds that $m \sqsupseteq c$ and therefore also $m \sqsupseteq o$.

We now show that $\mathit{fo}(o')$ and o are not ordered. Assume for a contradiction that $\mathit{fo}(o') \sqsubseteq o$. Then it holds that $\mathit{fo}(o') \sqsubseteq m$. Since we also know that $\mathit{fo}(o') \sqsubseteq \overline{m}$ it must hold that $\gamma(\mathit{fo}(o')) = \emptyset$. This violates the condition that the backjumping function returns a non-conflicting prefix ρ'. Therefore $\mathit{fo}(o') \not\sqsubseteq o$. Now assume for a contradiction that $o \sqsubseteq \mathit{fo}(o')$. Then $o \sqsubseteq \overline{m}$, and from $o \sqsubseteq c$ we can derive $o \sqsubseteq m$ and consequently $\gamma(o) = \bot$. This is not possible because Abstract-non-$\bot^*$ never returns a trail representing the empty set. We have shown that o and $\mathit{fo}(o')$ are incomparable. $\square$

Analogous to the case of propositional SAT, asserting backjumps can be implemented using a meet-irreducible extrapolation operator. We informally sketch the reason: Let ρ be the stack $\rho_1 \ldots \rho_k (\rho_k \sqcap m)$ during a run of the procedure such that the last element $(\rho_k \sqcap m)$ is derived by applying a meet-irreducible extrapolation operator. If we can determine that $f(\gamma_O(\bigsqcap \rho)) = \emptyset$, we may refine *fo* by learning the transformer $\mathit{gunit}_{\bigsqcap \rho}$. Backjumping to $\rho_1 \ldots \rho_k$ and then applying $\mathit{gunit}_{\bigsqcap \rho}$ yields the element $\rho_k \sqcap \overline{m}$, which drives the search to a new region.

Theorem 13 (Relative Completeness)**.** *If the downwards extrapolation $\triangledown_{\downarrow}$ is precise, and* ACDCL$(\mathit{fo}, \triangledown_{\downarrow}, \tilde{\mathit{fu}}, \triangle_{\uparrow})$ *terminates, the result is $\bot$ or* not $\bot$.

Proof. We prove by contradiction. If unknown is returned, Abstract-non-$\bot$ returned (unknown, o), which is only possible if $\triangledown_{\downarrow} o$ equals o. Since $\triangledown_{\downarrow}$ is precise, we know that *fo* is γ-complete at o. This is impossible because not $\bot$ would have been returned. $\square$

Theorem 14 (Termination)**.** *If O is finite and backjumps are asserting, then* ACDCL *makes progress.*

Proof. The procedures Abstract-non-$\bot$ and Abstract-non-$\top$ terminate over finite lattices. To prove the theorem, we assume that the outer loop of ACDCL is non-terminating and derive a contradiction.

Let r be the last element of the sequence ρ at the beginning of an arbitrary iteration of the main loop of ACDCL. Since the lattice is finite, we can reason using well-founded induction. We show that backjumping eventually resets the stack to a prefix whose last element is $b \sqsupset r$.

Base step: Assume r is an atom, meaning every r' that satisfies $r' \sqsubset r$ also satisfies $r' = \bot$. Then the Abstract-non-$\top$ step will be initialised with $\{r\}$, and will return an element $c \sqsupseteq r$. Therefore, it holds that after learning, $\mathit{fo}(r) = \bot$. Backjumping therefore necessarily returns to an element $b \sqsupset r$ in order to satisfy the condition that after a backjump $\mathit{fo}(b) \neq \bot$.

Induction step: Assume that for all $r' \sqsubset r$, the induction hypothesis holds. Assume for a contradiction that backjumping never jumps to an element greater than r. Then r is the target of a backjump infinitely often. Then it must be the case that in two subsequent iterations i_1 and i_2, Abstract-non-$\bot^*$ returns a stack with the same final conflicting element c such that r is the target of the corresponding backjump in both cases. After the first such backjump, we have by Lemma 12 that *fo* immediately infers an element $a \sqsubset r$ such that a and c are unordered. Specifically, a is not greater than c. Therefore, any subsequent call to Abstract-non-$\bot^*$ returns a stack with a last element that is smaller than a, and therefore different from c. This contradicts the above, which states that i_2 backtracks from c.

This completes the proof. It therefore holds that, in an infinite run, backtracking will eventually return an element strictly greater than $\top$. This is impossible, hence all runs of ACDCL terminate. $\square$

Though we only show termination for finite lattices, the termination of CDCL is usually non-trivial to argue. When applied to decidable logics that involve infinite lattices (such as linear arithmetic), specific details of the theory, the transformers and acceleration techniques must be used to prove termination. It is not clear that there are general termination arguments for infinite lattices. Ascending and descending chain conditions are not enough because of the alternation of the two procedures and the use of decisions and choice, which also operate on infinite sets.

6. Abstract Implication Graphs

The design of underapproximate transformers, such as $\tilde{fu}$, is a challenging problem that has received less attention than the design of overapproximate transformers. SAT solvers use *fo* applications to construct an implication graph, and use this graph to derive $\tilde{fu}$. We now generalise implication graphs to other domains.

Implication Graphs as Transformer Abstractions Consider a proof rule that when applied to antecedents $a_1, \ldots, a_n$ yields the consequent c. If we view antecedents and consequents as elements of an abstract domain, we can formalise a proof rule as a transformer *fo* that satisfies $fo(\bigsqcap\{a_1, \ldots, a_n\}) \sqsubseteq c$. Implication graphs provide a means to derive antecedents from consequents thereby implementing an abduction transformer.

Recall that $\mathcal{M}_O$ is the set of meet irreducibles of O. An *implication edge* is a directed hyperedge in $E_\mathcal{I} \mathrel{\hat{=}} \mathscr{P}(\mathcal{M}_O) \times \mathcal{M}_O$. We order implication edges so that $E_1 \preceq E_2$ means E_1 requires weaker antecedents than E_2 to derive stronger consequents. Define $(M_1, m_1) \preceq (M_2, m_2)$ to hold if $m_1 \sqsubseteq m_2$ and for every m_1' in M_1 there exists m_2' in M_2 satisfying $m_1' \sqsupseteq m_2'$. An *implication graph* G is an upwards-closed set of implication edges and the *domain of implication graphs* $\mathcal{I} \mathrel{\hat{=}} (\mathscr{U}(E_\mathcal{I}), \supseteq, \cup, \cap)$ contains implication graphs with the supserset order. In practice, an upwards-closed set is represented by its minimal elements.

Figure 5 contains examples of implication graphs. All incoming edges to a node represent one hyperedge and each hyperedge represents its upwards-closure. A hyperedge of the form $(\{\}, m)$ is depicted by m.

An implication graph represents an abstraction of a transformer, as shown below. Assume below that *fo* is a reductive transformer. Let $(Red_O, \sqsubseteq)$ be the lattice of reductive transformers on O with the pointwise order. Define two functions $\alpha_\mathcal{I} : Red_O \to \mathcal{I}$ and $\gamma_\mathcal{I} : \mathcal{I} \to Red_O$ below.

$$\alpha_\mathcal{I}(f) \mathrel{\hat{=}} \{(M, m) \in E_\mathcal{I} \mid f(\bigsqcap M) \sqsubseteq m\}$$

$$\gamma_\mathcal{I}(I) \mathrel{\hat{=}} \lambda o. \bigsqcap\{m \mid \exists(M, m) \in I. \bigsqcap M \sqsupseteq o\}$$

Proposition 15. *The functions $\alpha_\mathcal{I}$ and $\gamma_\mathcal{I}$ form a Galois connection.*

Example 3. We illustrate the application graph construction for reachability analysis of a program with the interval abstract domain. The problem is to determine if the location $\lightning$ is reachable in the flow graph in Figure 6. An implication graph derived by applying abstract successor transformers is shown alongside.

The label DL0, denotes *decision level 0*, which contains meet irreducibles that are deduced without making assumptions. These represent facts obtained by a run of a static analyser. Due to imprecision, the error is reachable in the abstract.

Suppose we apply downwards extrapolation to force the constraint $a \leq -42$ at the location n_1. This constraint is shaded in the figure. If we continue static analysis with this assumption, we conclude that $\lightning$ is unreachable. The facts required to arrive at this conclusion are shown in the implication graph in the figure. $\triangleleft$

Extracting Dual Transformers We now show how conflict analysis can be viewed as construction of a dual transformer from an implication graph. The dual transformer maps a set of consequences C to the set of antecedents A from which we could have derived C. The *dual implication transformer* defined by an implication graph I is $\tilde{fu}_I : \mathscr{D}(O) \to \mathscr{D}(O)$ below.

$$\tilde{fu}_I(C) \mathrel{\hat{=}} C \cup \{q \sqcap (\bigsqcap M) \mid q \sqcap m \in C \wedge (M, m) \in I\}$$

Theorem 16. *If the implication graph I represents an overapproximation of fo, the transformer $\tilde{fu}_I$ underapproximates $\tilde{f}$.*

Proof. Consider an implication graph I that overapproximates the transformer *fo* and a downwards closed set $C \subseteq O$. We prove the inequality $\gamma_{\mathscr{D}(O)} \circ \tilde{fu}_I(C) \subseteq \tilde{f} \circ \gamma_{\mathscr{D}(O)}(C)$.

We show that $fo(c)$ is in C for all c in $\tilde{fu}_I(C)$. Consider c in $\tilde{fu}_I(C)$. If c is in C, we have that $fo(c)$ is in C because *fo* is reductive and C is downwards-closed. If c is in $\tilde{fu}_I(C) \setminus C$, c is, by definition of the form $c = q \sqcap (\bigsqcap M)$, where $q \sqcap m$ is in C for some (M, m) in I. Since I soundly approximates *fo*, we have that $fo(\bigsqcap M) \sqsubseteq m$, and as a consequence also that $fo(c) = fo(q \sqcap (\bigsqcap M)) \sqsubseteq q \sqcap m$. Since C is downwards closed and $q \sqcap m$ is in C it holds that $c \in C$. We have shown that $\{fo(c) \mid c \in \tilde{fu}_I(C)\}$ is contained in C.

The soundness of *fo* implies that the downwards-closure of $\bigcup\left\{f(\gamma_O(c)) \mid c \in \tilde{fu}_I(C)\right\}$ is contained in the downwards closure of $\left\{\gamma_O(c) \mid c \in \tilde{fu}_I(C)\right\}$. We can rewrite the above condition as $f \circ \gamma_{\mathscr{D}(O)} \circ \tilde{fu}_I \subseteq \gamma_{\mathscr{D}(O)}$, where $\subseteq$ is the pointwise lifting of the subset order. The functions $(f, \tilde{f})$ form a Galois connection, due to which, the inequality $f \circ \gamma_{\mathscr{D}(O)} \circ \tilde{fu}_I \subseteq \gamma_{\mathscr{D}(O)}$ is equivalent to $\gamma_{\mathscr{D}(O)} \circ \tilde{fu}_I \subseteq \tilde{f} \circ \gamma_{\mathscr{D}(O)}$. Thus, $\tilde{fu}_I$ underapproximates $\tilde{f}$. $\square$

Example 4. Consider the interval implication graph I shown in Figure 5, consisting of the following hyperedges.

$$\{(\emptyset, y \geq 5), (\emptyset, x \leq 2), (\{y \geq 5\}, x \geq 6), (\{x \geq 6, x \leq 2\}, \bot)\}$$

We compute a least fixed point over $\tilde{fu}_I$, and represent downwards-closed sets by their maximal elements.

$$C_0 = \tilde{fu}_I(\bot) = \{(x \geq 6, x \leq 2)\}$$
$$C_1 = \tilde{fu}_I(C_0) = C_0 \cup \{\{(y \geq 5, x \leq 2), (x \geq 6)\}\}$$
$$C_2 = \tilde{fu}_I(C_1) = C_1 \cup \{\top\} = \top$$

It follows that the formula is unsatisfiable. This fixed point computation, in which meet irreducibles are replaced by their explanations, is similar to conflict analysis in SAT solvers. $\triangleleft$

Generalising Implication Graphs We briefly discuss implication graph generalisation. See [17] for a more algorithmic discussion. In propositional CDCL, meet irreducibles are of the form $\{x \mapsto v\}$ where v is $\mathtt{t}$ or $\mathtt{f}$. Meet irreducible partial assignments are incomparable. In more general lattices, such as intervals, there is an order on meet irreducibles. The order on meet irreducibles represents implications that follow from theory axioms. This order

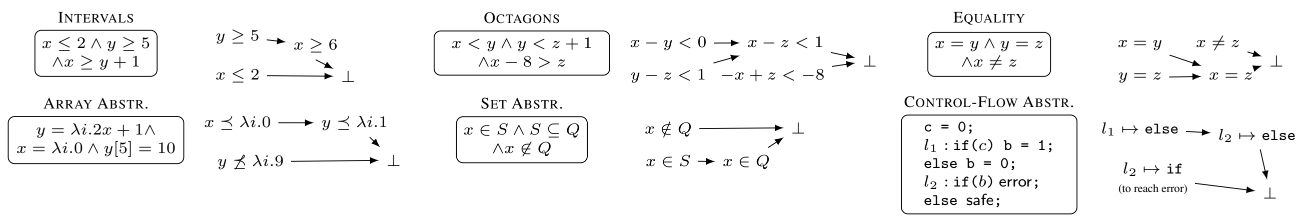

Figure 5. Abstract Implication Graphs

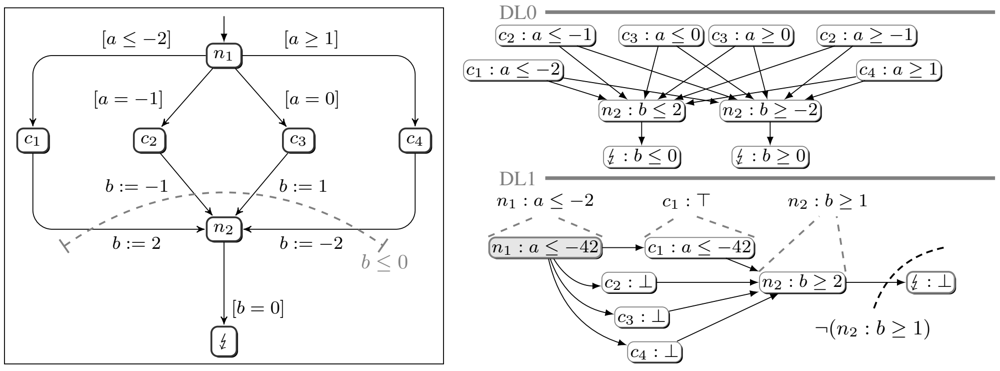

Figure 6. Implication Graph for Reachability of the Program Location Marked with $\lightning$

can be used to generalise implication graphs to derive more useful dual transformers.

Example 5. We illustrate implication graph generalisation on the example in Figure 6. Computing $\tilde{fu}_I$ for the graph I depicted leads to a sound transformer. Using this transformer one can find, for example, that $n_2 : b \ge 2$ is sufficient to deduce a conflict.

There is a more general graph that still allows deduction of unreachability of $\lightning$: The constraint on the node $n_2 : b \ge 2$ can be weakened to obtain $n_2 : b \ge 1$. Likewise, $c_1 : a \le -42$ and $n_1 : a \le -42$ can be generalised, respectively, to $c_1 : \top$ and $n_1 : a \le -2$. Computing $\tilde{fu}_{I'}$ over the resulting graph I' allows one to find that $n_2 : b \ge 1$ is sufficient to deduce a conflict. By generalising the graph, we can derive more general conditions under which a location is unreachable.

Learning allows us to negate this condition and discover that the flow graph always satisfies $n_2 : b \le 0$ because the variable b is in the interval $[-1, -2]$ at the location n_2. We can now prove that $\lightning$ is unreachable without any assumptions. We note that ACDCL instantiated in this way over programs is a method to dynamically discover a trace partitioning [25]. ◁

7. Related Work

A survey of the literature on combining logical solvers and abstract interpreters is beyond the scope of this paper. We limit ourselves to discussing extensions of CDCL solvers to richer problems.

The DPLL(T) architecture, also called *lazy* SMT combines a SAT solver with a solver for the conjunctive fragment of a theory, to determine satisfiability of quantifier-free formulae in a theory [15]. In the original lazy SMT architecture, decisions and learning take place in the propositional solver, which may cause theory facts to be enumerated. *Splitting on demand* allows new theory facts to be encoded by propositional variables and added to a formula [2]. ACDCL is an alternative to lazy SMT in which all reasoning happens directly in a fragment of the theory, and no propositional solver is required. See [3] for a theoretical and empirical comparison of ACDCL and DPLL(T).

Other approaches that jettison a propositional solver in favour of making decisions and learning directly in a theory include *natural-domain* SMT [5], *generalised* DPLL [23] and theory specific solutions for equality [1], and integer linear arithmetic [20] using the cutting-planes proof system. ACDCL is one recipe for generalising CDCL to new domains. The details of the decision and learning operations have to be designed anew for each theory, but the algorithm used by the solver, and its soundness and relative-completeness follow from our work. In separate work, we have instantiated ACDCL to derive an SMT solver for floating-point logic and shown that this solver performs better than using a propositional encoding [17], and better than DPLL(T) [17].

SMT solvers use the *Nelson-Oppen* and *Delayed Theory Combination* methods to combine solvers for different theories. Abstract interpretation is a modular framework, which allows abstract domains to be combined in several different ways [8]. The *Nelson-Oppen* method and DPLL(T) have recently been shown to be instances of more general product constructions in abstract interpretation [3, 12]. These product constructions lift to instantiations of ACDCL: if ACDCL can be instantiated over domains O_1 and O_2, it can be instantiated over the cardinal product $O_1 \times O_2$.

Our work allows CDCL to be lifted to verification problems. We have instantiated ACDCL with the interval abstract domain to analyse the bounds of variables in programs with floating-point variables [14]. *Lazy annotation with interpolants* [22] lifts CDCL to programs, and uses an SMT solver to implement constraint propagation, and Craig interpolation for learning. We work with abstract domains, and do not assume the domain is closed under Boolean operations, or that it supports interpolation. *Satisfiability Modulo Path Programs* [18] lifts DPLL(T) to programs by combining a SAT solver and an abstract interpreter, and Stålmarck's method has been lifted to programs in [28].

Our work provides an abstract interpretation view of CDCL, which we extended to other procedures in [13], and to DPLL(T) in [3]. An account of Stålmarck's method is given in [27], and [24] contains a transition system view of CDCL.

A natural question is whether ACDCL is a form of Counter-Example Guided Abstraction Refinement (CEGAR) [4]. CEGAR uses non-$\perp$ witnesses to construct a new domain and new transformers. ACDCL implements proof-guided transformer refinement and makes progress when no non-$\perp$ witness is found. ACDCL never changes the domain, and this immutability is crucial for efficiency, because the implementations of the abstract domain and transformers can be highly optimised.

8. Conclusion

In this paper, we applied abstract interpretation to study the Conflict Driven Clause Learning algorithm (CDCL) implemented by contemporary SAT solvers. We showed that CDCL can be understood as a lattice-theoretic approximation algorithm that combines overapproximations of greatest fixed points and underapproximations of least fixed points to determine properties of a function on a Boolean lattice. Our generalised Abstract Conflict Driven Clause Learning (ACDCL) procedure and its correctness proofs rely on properties enjoyed by the data structures in SAT solvers as well as by abstract domains used in practice.

In separate work, we have instantiated ACDCL to derive an SMT solver and a program analyser and obtained positive results. Problems to investigate in future instantiations include satisfiability in theories of weak arithmetic, assertion checking with relational abstract domains, and nullness of pointers. A second family of problems is to design non-Boolean abstractions of states and traces so that ACDCL can be applied to model checking problems, and to determine emptiness of non-deterministic automata over finite and infinite words.

We opened the paper by recalling a question posed by Malik and Zhang about whether the lessons from the success of SAT solvers lift to other domains. We believe that our work contributes a mathematical answer to this question that applies to algorithmic issues. Our work does not explain or provide a means to lift the heuristics used by SAT solvers to new problems. We conjecture that heuristics to boost efficiency of constraint propagation are closely related to those for exploiting sparsity in program analysis. It is our hope that future work will provide a framework for understanding these connections and for lifting engineering techniques in SAT solvers to new problem domains.

Acknowledgments

This work was supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/H017585/1, and the FP7 STREP PINCETTE. Vijay D'Silva was supported by a Microsoft Research PhD Scholarship.

We thank the abstract interpretation community, in particular Patrick and Radhia Cousot, for their support and encouragement of this work. Vijay D'Silva thanks Heejong Lee and Wonchan Lee for their diligence in an independent project, which allowed him time for this one.

References

[1] B. Badban, J. van de Pol, O. Tveretina, and H. Zantema. Generalizing DPLL and satisfiability for equalities. *Information and Computation*, 205(8):1188–1211, 2007.

[2] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In *Proc. of Logic for Programming, Artificial Intelligence, and Reasoning*, pages 512–526, 2006.

[3] M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening. An abstract interpretation of DPLL(T). In *Proc. of Verification, Model Checking and Abstract Interpretation*, 2013. To appear.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. of the ACM*, 50:752–794, 2003.

[5] S. Cotton. Natural domain SMT: A preliminary assessment. In *Proc. of Formal Modeling and Analysis of Timed Systems*, pages 77–91, 2010.

[6] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1-2):47–103, Apr. 2002.

[7] P. Cousot. Abstract interpretation. MIT course 16.399, Feb.–May 2005.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Principles of Programming Languages*, pages 269–282, 1979.

[9] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[10] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[11] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[12] P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *Proc. of Foundations of Software Science and Computational Structures*, pages 456–472, 2011.

[13] V. D'Silva, L. Haller, and D. Kroening. Satisfiability solvers are static analysers. In *Proc. of Static Analysis Symposium*, pages 317–333, 2012.

[14] V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric bounds analysis with conflict-driven learning. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, pages 48–63, 2012.

[15] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proc. of Computer Aided Verification*, pages 175–188, 2004.

[16] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In *Proc. of Static Analysis Symposium*, pages 356–373, 2001.

[17] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *Proc. of Formal Methods in Computer-Aided Design*, pages 131–140, 2012.

[18] W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis via satisfiability modulo path programs. In *Proc. of Principles of Programming Languages*, pages 71–82, 2010.

[19] T. A. Henzinger, O. Kupferman, and S. Qadeer. From pre-historic to post-modern symbolic model checking. *Formal Methods in Systems Design*, 23(3):303–327, Nov. 2003.

[20] D. Jovanovic and L. M. de Moura. Cutting to the chase - solving linear integer arithmetic. In *Proc. of Automated Deduction*, pages 338–353, 2011.

[21] S. Malik and L. Zhang. Boolean satisfiability: From theoretical hardness to practical success. *Communications of the ACM*, 52:76–82, Aug. 2009.

[22] K. L. McMillan. Lazy annotation for program testing and verification. In *Proc. of Computer Aided Verification*, pages 104–118, 2010.

[23] K. L. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to richer logics. In *Proc. of Computer Aided Verification*, pages 462–476, 2009.

[24] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *JACM*, 53:937–977, 2006.

[25] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems*, 29, 2007.

[26] N. Sörensson and A. Biere. Minimizing learned clauses. In *Proc. of Theory and Applications of Satisfiability Testing*, pages 237–243, 2009.

[27] A. Thakur and T. Reps. A Generalization of Stålmarck's Method. In *Proc. of Static Analysis Symposium*, pages 334–351, 2012.

[28] A. Thakur and T. Reps. A method for symbolic computation of abstract operations. In *Proc. of Computer Aided Verification*. Springer, 2012.