Abstract
This article introduces an abstract interpretation framework that codifies the operations in SAT and SMT solvers in terms of lattices, transformers and fixed points. We develop the idea that a formula denotes a set of models in a universe of structures. This set of models has characterizations as fixed points of deduction, abduction and quantification transformers. A wide range of satisfiability procedures can be understood as computing and refining approximations of such fixed points. These include procedures in the DPLL family, those for preprocessing and inprocessing in SAT solvers, decision procedures for equality logics, weak arithmetics, and procedures for approximate quantification. Our framework provides a unified, mathematical basis for studying and combining program analysis and satisfiability procedures. A practical benefit of our work is a new, logic-agnostic architecture for implementing solvers.

Categories and Subject Descriptors F.4.1 [Mathematical Logic]: Mechanical Theorem Proving; I.2.3 [Deduction and Theorem Proving]: Deduction

Keywords Abstract interpretation; Logic; Decision Procedures

1. Reasoning and Abstraction
Static analyzers and satisfiability solvers represent practical triumphs of computer science in the face of theoretical hardness results. Static analysis problems are typically undecidable yet analyzers compute information that is indispensable in compiler optimization and program verification. The satisfiability problem for several logics and theories is NP-hard but SAT and SMT solvers handle large problem instances arising in practice. In this paper, we introduce an abstract interpretation framework that makes explicit some fundamental similarities between the way undecidable and NP-hard problems are solved in practice. This framework has several applications including lattice-theoretic characterizations of satisfiability algorithms [18, 19], the development of SMT solvers based on abstract interpretation [26], and the generalization of satisfiability algorithms to static analysis [3, 20].

Abstract interpretation is a lattice-theoretic framework for reasoning about fixed points [10, 13]. The idiomatic approach to applying abstract interpretation to a problem is to characterise solutions to the problem by fixed points, identify a space of fixed point approximations, and design an algorithm to compute these approximations. The application of abstract interpretation to static analysis can be understood in terms of the schema below. The box on the left is called an abstract domain. It consists of a lattice \((A, \sqcap, \sqcup, \bot, \top, T)\) with each element \(a\) of \(A\) representing a set of program states. Each statement \(s\) in the programming language defines four transformers. The predecessor transformer \(\text{pre}_s\) maps \(a\) to states the program may have come from before executing \(s\), while the successor transformer \(\text{post}_s\) maps \(a\) to states the program may reach after executing \(s\). The transformers \(\text{pre}_s\) and \(\text{post}_s\) capture must behaviour. Properties of programs are specified as fixed points of such transformers.

The box on the right represents procedures that use components of the abstract domain to reason about fixed points. Iterative procedures are used to compute fixed points. These procedures may use a widening (\(\triangledown\)) or dual widening (\(\triangleleft\)) operator to accelerate convergence. If the result is not precise enough, a narrowing (\(\bowtie\)) or dual narrowing (\(\bowtie\)) operator is used to refine the result. The architecture above achieves a valuable separation of concerns by allowing the design and implementation of an abstract domain to be independent of the fixed point approximation procedures.

Abstract satisfaction is a framework for applying fixed point approximation to logical reasoning in the same manner that abstract interpretation was first applied to static analysis [10]. Consider the satisfiability problem for a logic. An SMT solver typically works in a fragment \(T\) of the logic. Elements of \(T\) are represented using data structures such as sets, partial functions, graphs, or matrices. These elements are manipulated using techniques called constraint propagation, decisions, learning and subsumption.

We show that a solver can be understood in terms of the schema below, which closely resembles the structure of a static analyzer. Elements of \(T\), ordered by implication, form a lattice of approximations \((T, \Rightarrow, \sqcap, \sqcup, \bot, \top, \text{false}, \text{true})\). A solver can use deduction to compute facts implied by \(\varphi\), or use abduction to compute facts that imply \(\varphi\). These operations define deduction and abduction transformers \((\text{ded}_\varphi\) and \(\text{abd}_\varphi\)), and their counterparts \((\text{cdc}_\varphi\) and \(\text{cabd}_\varphi\)) for contrapositive reasoning.

\[
T \Rightarrow \sqcap \text{false} \text{true} \\
\text{ded}_\varphi \text{abd}_\varphi \text{cdc}_\varphi \text{cabd}_\varphi \\
\text{ext}_\top \text{ext}_\bot \text{sep}
\]

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM 978-1-4503-2544-8/14/01.
http://dx.doi.org/10.1145/2535838.2535868

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

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

Daniel Kroening
Department of Computer Science
University of Oxford
daniel.kroening@cs.ox.ac.uk
Propagation and learning in solvers can be viewed as the application of these transformers. Techniques like decisions, used by a solver to improve precision, correspond to a relaxation of widening called extrapolation (denoted ext₁ and ext₂), while techniques like subsumption and clause minimization correspond to a relaxation of narrowing, which we call interpolation (denoted int₁ and int₂). Thus, despite external differences, there are fundamental similarities between the internals of SAT and SMT solvers and static analyzers. We believe that making these similarities explicit has several consequences that we discuss below.

Abstract Interpretation to SMT One consequence is a transfer of techniques from abstract interpretation to SMT solvers. Solvers have been extended with abstractions [4, 30] and joins [1] to improve time and memory efficiency, and with widening [33] to aid in guessing loop invariants. We show that the internal data structures of SMT solvers are lattices, which means that these data structures also support joins and widening. Crucially, we show that quantifiers are transformers, which means that abstract quantifiers and best abstract quantifiers are well defined notions. Thus, an abstract interpretation perspective suggests a new, approximate approach to deduction and quantifier elimination.

SMT to Static Analysis The main conclusion of our work is that SMT solvers, like static analyzers, operate on imprecise abstractions. However, SMT solvers return precise results, which means their algorithms can be understood as techniques for refining an imprecise analysis. These techniques are based on properties of lattices and can be used to refine static analyses. SMT solvers for decidable logics implement refinement procedures that are guaranteed to terminate (though termination proofs can be non-trivial). The fundamental undecidability of static analysis problems precludes the existence of terminating refinement procedures.

An insight we present in Section 4 is that deduction and abduction in a logic coincide with reasoning about the postconditions and preconditions of conditional statements. Improvements in deduction should lead to an improved handling of conditionals in static analyzers. Moreover, preprocessing and inprocessing techniques, which are responsible for recent performance improvements in solvers can be lifted to static analysis constraints.

A Grand Unification A lofty goal, towards which this work is an early step, is to achieve a uniform theoretical and practical treatment of static analysis and SMT solving. Specifically, if both technologies can be understood in terms of lattices and transformers, their similarities and differences can be studied using lattice theoretic techniques. The three different problems of combining static analyzers, combining SMT solvers, and combining a static analyzer with an SMT solver can be reduced to the single problem of combining fixed point approximation procedures. Existing procedures such as the reduced product, Nelson-Oppen, and DPLL(T), which are now understood to be combination procedures [2, 11, 14], can all be applied to the same task.

We anticipate practical benefits from carrying out a unification programme. Decomposing a complex piece of software like a static analyzer or an SMT solver into smaller blocks consisting of lattice elements, transformers, and an iteration engine leads to a mathematically justified modular design. We expect this modularity to contribute to the development of extensible and programmable solvers and analyzers, and reduce the performance and development overheads involved in integrating different technologies.

2. Mathematical Preliminaries

We denote the complement of a set \( S \) as \( \neg S \), and the set of all subsets of \( S \), as the powerset \( \mathcal{P}(S) \). The function from \( x \) to \( f(g(x)) \) is denoted \( f \circ g \), and a function \( f \) is treated as a set \( \{ a \mapsto f(a), \ldots \} \) when convenient.

Sequences We use sequences to simplify presentation. An intermediate A-sequence is a function \( S : \{0, \ldots, m - 1\} \rightarrow A \), whose length \( m \) is denoted \( \text{len}(S) \). We write \( f(S) \) for the application \( f(a_0, \ldots, a_{n-1}) \) and leave implicit that \( S \) has length \( n \). Given a function \( g : A \rightarrow C \), we write \( g[a \mapsto c] \) for the function that maps \( a \) to \( c \) and \( x \) distinct from \( a \) to \( g(x) \). We write a sequence of substitutions \( g[a_0 \mapsto c_0][a_1 \mapsto c_1] \cdots \) with pairwise distinct elements \( a_i \) as \( g[a \mapsto c] \).

Lattices A transformer is a monotone function on a lattice. A lattice is 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 lattice is reductive if \( f(x) \subseteq x \) for all \( x \) and is extensive if \( f(x) \supseteq x \) for all \( x \). A function is idempotent if \( f(f(x)) = f(x) \) for all \( x \). 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 \( f(x) \subseteq g(x) \) holds for all \( x \). The pointwise meet of \( f \) and \( g \), denoted \( f \land g \), where both functions map into a lattice is defined as \( x \mapsto f(x) \land g(x) \). The pointwise join is similarly defined. The set of transformers on a complete lattice form a complete lattice under the pointwise order.

A lattice is distributive if every \( x, y \) and \( z \) satisfy \( x \lor (y \land z) = (x \lor y) \land (x \lor z) \), which is equivalent to the identity obtained by interchanging meets and joins. An element \( y \) on a bounded lattice is the complement of \( x \) if \( x \land y = \bot \) and \( x \lor y = \top \). Complements may not exist and when they do, may not be unique. We use the notation \( \neg x \) or \( \sim x \) for unique complements. Complements in a distributive lattice are unique. A Boolean lattice is complemented and distributive. The De Morgan dual of a function \( f \) on a Boolean lattice is \( f \equiv \neg f \circ \neg \sim \). The powerset lifting of \( f : A \rightarrow B \) is the function \( f : \mathcal{P}(A) \rightarrow \mathcal{P}(B) \) that maps a set to its image under \( f \).

The least and greatest fixed points of a monotone function \( f \) on a complete lattice are denoted \( \mathit{gfp}(f) \) and \( \mathit{hp}(f) \).

Galois Connections Let \( (L, \sqsubseteq) \) and \( (M, \preceq) \) be posets. Two functions \( \alpha : L \rightarrow M \) and \( \gamma : M \rightarrow L \) form a Galois connection if for all \( x \in L \) and \( y \in M \), \( \alpha(x) \preceq y \) if and only if \( x \sqsubseteq \gamma(y) \). A Galois connection is written as \( \alpha \dashv \dot\vdash \gamma \) or \( (L, \alpha, \gamma, M) \). The function \( \alpha \) is called the left adjoint and \( \gamma \) is called the right adjoint of the Galois connection. In a Galois insertion, \( \alpha \) is a surjection.

3. A Collecting Semantics for First-Order Logic

The phrase collecting semantics in abstract interpretation refers to associating meaning to an object in terms of its properties. For example, a trace property is a set of sequences of states and the collecting trace semantics of a program, which is also the strongest trace property the program satisfies, is the set of all program executions. In this section, we introduce a compositional, collecting semantics for quantified first-order formulae. Our semantics allows us to interpret a formula as an element of a lattice, so that abstract interpretation of formulae, and of properties of formulae, is well defined. The collecting semantics of a term is defined by lifting the standard evaluation semantics of terms to sets of environments. The collecting semantics of a formula is the set of models of the formula. The Boolean operations of conjunction, disjunction and negation have their standard interpretation as intersection, union and set complement. Quantifiers are interpreted as transformers between structures over different sets of variables.

Of several lattice-theoretic semantics for quantified first-order logics [38], we use a category-theoretic treatment due to Pitts [40]. A key feature of this treatment is to make the set of free variables part of the syntax of a formula. The structure of the lattices over
which a formula is interpreted are then determined by the syntax of formula. Quantifiers define transformers between lattices over different sets of free variables. See [38] for a discussion of the challenges in giving a lattice-theoretic semantics to quantifiers.

**Structural Rules** We recall the structural rules for forming terms and formulae. The signature of a first-order logic $\langle \text{Sig}, \text{ar} \rangle$ consists of disjoint sets $\text{Sig} = \text{Pred} \cup \text{Fun}$ of predicate and function symbols whose arity is $\text{ar} : \text{Sig} \to \mathbb{N}$. A nullary function symbol is called a constant. We use $P, Q, R$ to range over predicate symbols and $F, G, H$ to range over function symbols.

Let $\text{Vars}$ be a set of variables and $x, y, z$ range over variables. A **first-order context** $\Gamma$ is a finite sequence of variables in which each variable occurs exactly once. We write $\emptyset$ for the empty sequence. $\Gamma, \Gamma'$ for sequence concatenation, and $\text{var}(\Gamma)$ for the set of variables in a context. In the case of many-sorted logic, a context is a sequence of pairs, where each pair consists of a variable and a sort. A context $\Gamma'$ is a **subcontext** of $\Gamma$ if $\text{var}(\Gamma') \subseteq \text{var}(\Gamma)$.

The rules from [40] for forming terms-in-context are given below. Henceforth, we abbreviate ‘terms-in-context’ to ‘terms’ for convenience. In addition to standard rules for variable introduction $(\text{var})$ and function composition $(\text{Fun})$, we use a rule $(\text{Seq})$ for forming a sequence of terms. The rule for function composition has the side condition that $\text{ar}(F)$ is $\text{len}(t)$.

**Semantic Structures** We now introduce a lattice-theoretic structure in which to interpret formulae. This structure consists of a lattice and transformers and provides a template for implementing SMT solvers based on abstract interpretation. Recall that the classical semantics of first-order logic is given by a $\text{Sig}$-interpretation $\mathcal{M} = (\text{Val}, \text{int})$, which consists of a universe $\text{Val}$ and an interpretation that maps each function symbol $F$ to a function $\text{int}(F) : \text{Val}^{\text{ar}(F)} \to \text{Val}$ and each predicate symbol $P$ to a relation $\text{int}(P) \subseteq \text{Val}^{\text{int}(P)}$.

An environment over $\Gamma$ maps variables in $\Gamma$ to values. Let $\text{Env}_\Gamma : \text{var}(\Gamma) \to \text{Val}$ be the set of environments over $\Gamma$. The classical semantics of first-order logic is given by a relation $\mathcal{M}, \varepsilon \models \varphi$ specifying when an environment satisfies a formula.

Categorical logic does away with environments by observing that $\text{Env}_\Gamma \to \text{Val}$ is isomorphic to $\text{Val}^{\text{len}(\Gamma)}$. First-order hyperdoctrines significantly extend this observation to provide a category-theoretic semantics for first-order logic [32, 40]. In Definition 1 below, we adapt the definition of a first-order hyperdoctrine to posets of environments. The reader may rightly baulk at the length of the definition. One can view the classical and algebraic definitions of semantics as making different tradeoffs. In the classical semantics, first-order structures have a succinct definition but the definition of $|=\text{v}$ is verbose. In algebraic semantics, the definition of a structure may appear involved, but (in our opinion) leads to a succinctly defined semantics.

Each item in Definition 1 is required to provide semantics for some aspect of first-order logic. The lattices of tuples $\text{P}(\text{Val}^n)$ represent the domains over which function symbols are interpreted. The lattices of environments $\text{P}(\text{Env}_\Gamma)$ represent the domains over which terms-in-context are interpreted.

We use the function $\nu$-to-$\text{Val}$ that maps values to environments, and the function $\nu$-to-$\text{Env}$ to deal with substitution into a term, and the function $e$-to-$\text{Val}$ to deal with substitution into a predicate. The existential and universal projection functions $(\text{epr}$ and $\text{upr})$ give a concrete transformer semantics to quantifiers.

**Definition 1.** A collecting $\text{Sig}$-structure defined by a first-order structure $\mathcal{M} = (\text{Val}, \text{int})$ consists of the items below.

1. The lattices of tuples $\langle \{\text{P}(\text{Val}^n), \subseteq \} \mid n \in \mathbb{N} \rangle$.
2. The lattices of environments $\langle \{\text{P}(\text{Env}_\Gamma), \subseteq \} \mid \Gamma \rangle$ for every context $\Gamma$.
3. For every context $\Gamma = (x_0, \ldots, x_n)$ of length $n$, there are two translation functions for mapping between tuples and environments.

- $\nu$-to-$\text{Val}$ : $\nu \mapsto \nu$ $\to \text{P}(\text{Val}^\nu)$. Each $\nu$-to-$\text{Val}$ takes a word to a tuple.
- $\nu$-to-$\text{Env}$ : $\nu$ $\to \text{P}(\text{Env}_\Gamma)$. Each $\nu$-to-$\text{Env}$ maps a word to an environment.
- $e$-to-$\text{Val}$ : $e \mapsto e$ $\to \text{P}(\text{Val}^e)$. Each $e$-to-$\text{Val}$ maps an environment to a tuple.
- $e$-to-$\text{Env}$ : $e \mapsto e$ $\to \text{P}(\text{Env}_\Gamma)$. Each $e$-to-$\text{Env}$ maps an environment to an environment.

4. There is a weakening function for mapping environments from a subcontext $\Gamma'$ to environments over $\Gamma$. $\text{upr}_{\Gamma'} : \text{P}(\text{Env}_{\Gamma'}) \to \text{P}(\text{Env}_\Gamma$).

5. For every subcontext $\Gamma'$ of $\Gamma$ an existential projection $\text{epr}_{\Gamma', \Gamma}$ and a universal projection $\text{upr}_{\Gamma', \Gamma}$, in $\text{P}(\text{Env}_\Gamma)$. $\text{epr}_{\Gamma', \Gamma} : \text{P}(\text{Val}^{\text{int}(\Gamma')}) \to \text{P}(\text{Val}^{\text{int}(\Gamma)})$.

6. A lifting of the interpretation of function symbols to sets.

- $\text{int}(F) : \text{P}(\text{Val}^{\text{int}(F)}) \to \text{P}(\text{Val})$.
- $\text{int}(F) : \text{P}(\text{Val}^{\text{int}(F)}) \to \text{P}(\text{Val})$.

7. The relation $\text{int}(\cdot)$ for each predicate symbol.

The existential and universal projection functions as defined above eliminate arbitrary subcontexts. We only use them to eliminate single variables and use the following abbreviations. We write $\text{epr}_x$ and $\text{upr}_x$ for projections that extract a single variable by mapping environments over the context $\Gamma, x$ to those over $x$. Conversely, we write some$_x$ and all$_x$ for projections that eliminate
a single variable by mapping environments over $\Gamma$, $x$ to environments over $\Gamma$. We also write $\text{wk}_x$ for the weakening function from environments over $\Gamma$ to those over $\Gamma, x$.

**Collecting Semantics** The collecting semantics of terms and sequences of terms with respect to a classical interpretation $M$ follows. We drop the subscript $M$ when no ambiguity arises.

$\begin{align*}
\llbracket : \text{Term}_1^0 \rightarrow (\mathcal{P}(\text{Env}_N) \rightarrow \mathcal{P}(\text{Val}_N))
\end{align*}$

The semantics function is defined inductively below. We interpret variables as natural numbers and have to be rewritten before being evaluated. Observe that the term does not have to be rewritten before being evaluated.

Example 1. Consider the term $t \equiv x + 2y[y \mapsto 2x]$ derived below.

$\begin{align*}
\text{VAR} & \quad x, y \\
\text{WEAK} & \quad x, x : y \\
\text{FUN} & \quad x, x, y \\
\text{CSUB} & \quad x + 2y : x, y \\
\text{FUN} & \quad x, x
\end{align*}$

We interpret variables as natural numbers and $+$ as addition. The semantics of the term above is a function in $\mathcal{P}(\text{Env}_N) \rightarrow \mathcal{P}(\mathbb{N})$.

$\begin{align*}
[t] : \Gamma & \equiv \{x \mid \epsilon(x) \in E\} \\
\llbracket x & \mapsto \bar{r} \rrbracket : \Gamma \\
\llbracket \bar{r} & \mapsto \bar{r} \rrbracket : \Gamma \\
\llbracket \rho & \mapsto \bar{r} \rrbracket : \Gamma \\
\llbracket \rho & \mapsto \bar{r} \rrbracket
\end{align*}$

Example 2. We extend Example 1 to illustrate the semantics of quantification. Consider the formula

$\phi \equiv \exists x. x + 2y[y \mapsto 2x] = z$

with $=$ interpreted as equality over the natural numbers. The relation $\text{cint}(\epsilon) = \{(n, n) \mid n \in \mathbb{N}\}$ and the semantics of the formula is a set of environments over $z$.

$\begin{align*}
\llbracket \exists x. x + 2y[y \mapsto 2x] = z \mapsto \bar{r} \rrbracket & \equiv \text{some}_x\llbracket x + 2y \mapsto 2x \mapsto \bar{r} \rrbracket
\end{align*}$

As expected, $z$ maps to multiples of 5. This example shows that the semantics of a quantified formula can be calculated mechanically by applying the appropriate transformers. If the concrete transformers are replaced with abstract transformers, we can similarly calculate an abstract semantics.

The collecting semantics we have defined is consistent with the classical semantics of first-order logic.

**Theorem 2.** For each formula $\phi \in \text{Env}_N$, classical, first-order interpretation $M$ and environment $\epsilon \in \text{Env}_N$, $M, \epsilon \models \phi$ exactly if $\epsilon \in \llbracket \phi \rrbracket_M$.

We also refer to environments as structures. Let $\Gamma$ be a non-empty context. A structure $\epsilon$ is a model of $\phi$ if $\epsilon \in \llbracket \phi \rrbracket$. A formula $\phi$ is unsatisfiable in $M$ if $\llbracket \phi \rrbracket_M$ is the empty set and is satisfiable in $M$ otherwise. We refer to satisfiability in a structure as satisfiability for the rest of the paper.

A sentence is a formula in an empty context. The set of environments over the empty context is the empty set. If $\llbracket \phi \rrbracket = \emptyset$, we say that $\phi$ is true in $M$, and otherwise, $\phi$ is false in $M$.

### 4. Concrete Reasoning

The basic operations in logical reasoning can be viewed as giving a dynamic interpretation to an implication $\phi \Rightarrow \psi$. Deduction is the process of deriving $\psi$ from $\phi$. Abduction is the process of deriving $\psi$ from $\phi$. In classical logic, these processes have contrapositive formulations: we can start with $\neg \phi$ and attempt to deduce $\neg \psi$, a process we call contraduction, or start with $\neg \psi$ and attempt to abduce $\neg \phi$, a process we call contraduction. In this section, we model these processes using transformers and characterize properties of formulae as fixed points of these transformers.

The set of formulae that can be derived from a set of formulae $\Phi$ using a set of rules $R$ forms the deductive closure of $\Phi$ with respect to $R$. Deductive closure and automated reasoning procedures have characterizations in terms of Tarski’s consequence operator, or as the transitive closure of a set of rewrite rules. An important difference between these characterizations and ours is that we operate on sets of structures, so our notions of deduction and abduction are semantic. Existing characterizations can be derived from ours by abstract interpretation but we can also derive abstractions of deduction that operate on objects other than formulae.

### 4.1 Structure Transformers

A structure transformer for formulae in $\text{Form}_\Phi$ is a function $T_\phi : \mathcal{P}(\text{Env}_N) \rightarrow \mathcal{P}(\text{Env}_N)$. Structure transformers encode reasoning about the models and countermodels of a formula. The deduction transformer $\text{ded}_\phi$ which encodes reasoning about models of $\phi$. In
the definition below, we assume that the set-theoretic operations are lifted pointwise to functions.

\[ \text{ded}_{P(i)}(X) = X \cap \{ P(i) : \Gamma \} \]
\[ \text{ded}_{\varphi \land \psi} = \text{ded}_{\varphi} \cap \text{ded}_{\psi} \]
\[ \text{ded}_{\neg \varphi} = \neg \{ \varphi : \Gamma \} \]
\[ \text{ded}_{\varphi \lor \psi} = \text{ded}_{\varphi} \cup \text{ded}_{\psi} \]

The use of negation in defining \( \text{ded}_{\neg \varphi} \) is problematic in general because lifting such a definition to abstractions requires a structure that supports Boolean reasoning. We discuss this issue in greater detail shortly.

The contraduction transformer \( \text{cded}_{\varphi} \) encodes contrapositive reasoning and manipulates countermodels of \( \varphi \). If \( \text{cded}_{\varphi} \) is used for satisfiability checking, \( \text{cded}_{\varphi} \) can be used for validity checking.

\[ \text{cded}_{P(i)}(X) = X \cap \neg \{ P(i) : \Gamma \} \]
\[ \text{cded}_{\varphi \land \psi} = \text{cded}_{\varphi} \cap \text{cded}_{\psi} \]
\[ \text{cded}_{\neg \varphi} = \neg \text{cded}_{\varphi} \]
\[ \text{cded}_{\varphi \lor \psi} = \text{cded}_{\varphi} \cup \text{cded}_{\psi} \]

The two transformers above reason forwards in that they start from hypotheses and attempt to derive conclusions. The dual notion to deduction is abduction, where we start from a conclusion and derive the hypotheses under which that conclusion holds. For example if we can abduce true from a formula \( \varphi \), we know that \( \varphi \) is valid with respect to a set of structures.

\[ \text{abd}_{P(i)}(X) = X \cup \{ P(i) : \Gamma \} \]
\[ \text{abd}_{\varphi \land \psi} = \text{abd}_{\varphi} \cup \text{abd}_{\psi} \]
\[ \text{abd}_{\neg \varphi} = \neg \text{abd}_{\varphi} \]
\[ \text{abd}_{\varphi \lor \psi} = \text{abd}_{\varphi} \cap \text{abd}_{\psi} \]

Finally, we have a contraabduction transformer which models starting from a conclusion and deriving the fallacies: hypotheses from which that conclusion surely does not follow. A contraabduction transformer can be used to prune the space of abductions.

\[ \text{cabd}_{P(i)}(X) = X \cup \{ P(i) : \Gamma \} \]
\[ \text{cabd}_{\varphi \land \psi} = \text{cabd}_{\varphi} \cap \text{cabd}_{\psi} \]
\[ \text{cabd}_{\neg \varphi} = \neg \text{cabd}_{\varphi} \]
\[ \text{cabd}_{\varphi \lor \psi} = \text{cabd}_{\varphi} \cup \text{cabd}_{\psi} \]

In addition to deduction and abduction, quantifier elimination is fundamental to logical reasoning. The transformers \( \text{sone}_{\varphi} \) and \( \text{all}_{\varphi} \) model quantifier elimination and will henceforth be called quantification transformers.

There are several symmetries between deduction and abduction, which are preserved in our formulation. We make these properties explicit in Theorem 3. Below, the set-theoretic identities are not necessarily satisfied by abstract transformers, which is why they are not used as a definition. The characterizations of deduction and abduction as closures extend the existing characterization of logical consequence as a closure. The characterization of existential quantification \( \text{wk}_{\varphi} \circ \text{sone}_{\varphi} \) as an upper closure, and of universal quantification \( \text{wk}_{\varphi} \circ \text{all}_{\varphi} \) as a lower closure, when combined with the view of closure operators as abstractions [11], reiterates the connection between quantification and abstraction used in model checking [5]. The Galois connection between weakening and quantification was first observed by Lawvere [32] and indicates that even domains that do not support negation will support both existential and universal quantification.

**Theorem 3.** Structure transformers have the following properties.
1. The transformers satisfy the identities below:
   \[ \text{ded}_{\varphi}(X) = (X \cap \{ \varphi : \Gamma \}) \]
   \[ \text{cded}_{\varphi}(X) = (X \cap \neg \{ \varphi : \Gamma \}) \]
   \[ \text{abd}_{\varphi}(X) = (X \cup \{ \varphi : \Gamma \}) \]
   \[ \text{cabd}_{\varphi}(X) = (X \cup \neg \{ \varphi : \Gamma \}) \]
2. The transformers \( \text{ded}_{\varphi} \) and \( \text{cded}_{\varphi} \) are lower closures.
3. The transformers \( \text{abd}_{\varphi} \) and \( \text{cabd}_{\varphi} \) are upper closures.
4. The composition \( \text{wk}_{\varphi} \circ \text{sone}_{\varphi} \) is an upper closure and \( \text{wk}_{\varphi} \circ \text{all}_{\varphi} \) is a lower closure.
5. The pairs of transformers \( \text{ded}_{\varphi}, \text{abd}_{\varphi} \), \( \text{cded}_{\varphi}, \text{cabd}_{\varphi} \) and \( \text{all}_{\varphi}, \text{sone}_{\varphi} \) are De Morgan duals.
6. The pairs of transformers \( \text{ded}_{\varphi}, \text{abd}_{\varphi} \), \( \text{cded}_{\varphi}, \text{cabd}_{\varphi} \) form a Galois connection on \( \mathcal{P}(\text{Env}), \subseteq \).

**Figure 1.** The vertices represent deduction or abduction procedures for checking satisfiability of \( \varphi \) or validity of \( \neg \varphi \). The edges represent combinations of deduction and abduction procedures.

7. The quantification transformers are related to weakening by the following Galois connections.
   \[ \langle \mathcal{P}(\text{Env}), \subseteq \rangle \xleftarrow{\text{all}_{\varphi}} \langle \mathcal{P}(\text{Env}_{\varphi}), \subseteq \rangle \]
   \[ \langle \mathcal{P}(\text{Env}_{\varphi}), \subseteq \rangle \xleftarrow{\text{wk}_{\varphi}} \langle \mathcal{P}(\text{Env}), \subseteq \rangle \]

The Galois connections are useful for deriving equivalent formulations of properties of a formula. For example, the models of \( \varphi \) are \( \text{ded}_{\varphi}(\text{Env}) \). A formula is unsatisfiable exactly if \( \text{ded}_{\varphi}(\text{Env}) \subseteq \emptyset \), which by the Galois connection, is equivalent to \( \text{Env} \subseteq \text{cded}_{\varphi}(\emptyset) \). In words, we can determine if a formula is unsatisfiable by trying to deduce false from \( \varphi \) or trying to abduce \( \varphi \) from false. Similarly, a formula is valid exactly if \( \text{cded}_{\varphi}(\text{Env}) \subseteq \emptyset \), or equivalently \( \text{Env} \subseteq \text{cabd}_{\varphi}(\emptyset) \).

Since satisfiability corresponds to existence of a model, we can equivalently define it in terms of existential quantification. Treating quantifiers as transformers allows us to formalise techniques that combine variable elimination and deduction.

### 4.2 Fixed Points for Satisfiability

We now show that properties of a formula can be characterized by fixed points of structure transformers. Consider the process of computing consequences of \( \varphi \). We initially know nothing about \( \varphi \), which we can express as \( \varphi \Rightarrow \text{true} \). A single step of a reasoning algorithm may indicate that \( \varphi \Rightarrow \psi_1 \). After \( k \) steps, the algorithm may deduce that \( \varphi \land \cdots \land \psi_k \land \psi_{k+1} \Rightarrow \psi_{k+1} \). We know that the models of \( \varphi \) can be represented by the sequence \( [\psi_1], [\psi_1 \land \cdots \land \psi_k] \). The process of deduction can thus be viewed as a greatest fixed point computation, whose limit expresses the maximal information we can derive about models of \( \varphi \).

Fixed points of structure transformers represent brute force algorithms. The greatest fixed point \( \text{gfp}(\text{ded}_{\varphi}) \) represents the semantics of a solver that initially assumes that every structure is a model of \( \varphi \) and then eliminates countermodels of \( \varphi \). The fixed point \( \text{gfp}(\text{cded}_{\varphi}) \) represents a procedure that progresses by eliminating countermodels of \( \varphi \). The fixed point \( \text{gfp}(\text{abd}_{\varphi}) \) does the same for models of \( \neg \varphi \). The characterisation below first appeared in [18] and is included here to include validity as the dual of satisfiability.

**Theorem 4.** The following statements are equivalent.
1. The formula \( \varphi \) is unsatisfiable.
2. The greatest fixed point \( \text{gfp}(\text{ded}_{\varphi}) \) is empty.
3. The least fixed point \( \text{lfp}(\text{abd}_{\varphi}) \) has all structures.
4. The formula \( \neg \varphi \) is valid.
5. The least fixed point \( \text{lfp}(\text{cabd}_{\varphi}) \) has all structures.
6. The greatest fixed point \( \text{gfp}(\text{cded}_{\varphi}) \) is empty.

In program analysis and model checking, combinations of forward and backward analysis have advantages over a single method [9, 28]. In logical reasoning, we can similarly combine
the benefits of deduction and abduction. For instance, as shown in [19], Conflict Driven Clause Learning (CDCL) combines deduction and abduction. The original Davis and Putnam algorithm [16] combined deduction via ordered resolution and variable elimination with the pure literal rule. CDCL solvers that use the pure literal rule for pre-/in-processing combine all three [21].

We define combinations of deduction and abduction transformers below. The transformers are defined on $\mathcal{P}(\text{Env}) \times \mathcal{P}(\text{Env})$ with the lattice order shown alongside. For intuition about these combinations, consider a greatest fixed point iteration with the transformer $da_\varphi$, which combines deduction and abduction. The first element is $(\text{Env}, \emptyset)$, representing that every structure is a potential model and no structure is a potential countermodel. A single application of $da_\varphi$ yields $(\text{ded}_\varphi(\text{Env}), \text{abd}_\varphi(\emptyset))$, which is a fixed point. If $\varphi$ is unsatisfiable, this fixed point is $(\emptyset, \text{Env})$, representing that there are no models and every structure is a countermodel.

When using abstract transformers, a fixed point may not be reached in a single step so iteration allows information to be transferred between deduction and abduction.

$$
dcd_\varphi(X, Y) = (\text{ded}_\varphi(X \cap Y), \text{cde}_\varphi(\text{ded}_\varphi(X \cap Y))) \subseteq \subseteq
daa_\varphi(X, Y) = (\text{ded}_\varphi(X \cap \lnot Y), \text{abd}_\varphi(\lnot X \cup Y)) \subseteq \times \odot
dca_\varphi(X, Y) = (\text{ded}_\varphi(X \cup Y), \text{cabd}_\varphi(\text{ded}_\varphi(X \cup Y))) \subseteq \subseteq
cda_\varphi(X, Y) = (\text{ded}_\varphi(X \cap \lnot Y), \text{abdd}_\varphi(\lnot X \cup Y)) \subseteq \times \odot
caca_\varphi(X, Y) = (\text{abd}_\varphi(X \cup Y), \text{cabd}_\varphi(\text{adcd}_\varphi(X \cup Y))) \subseteq \times \odot
\$$

The $dcd_\varphi$ transformer is based on Cousot’s forward-backward iteration [9], but the other combinations are, to the best of our knowledge, new. An additional possibility for reasoning about satisfiability is to combine deduction with existential quantification $(ds_\varphi)$ and abduction with universal quantification $(aa_\varphi)$. These are only two of many possible combinations.

$$
ds_\varphi(X, Y) = (\text{ded}_\varphi(X \cap Y), \text{wk}_a \circ \text{some}_a(\text{X}(X \cap Y))) \subseteq \subseteq
aa_\varphi(X, Y) = (\text{abd}_\varphi(X \cup Y), \text{wk}_a \circ \text{all}_a(\text{X}(X \cap Y))) \times \odot
\$$

The application of these transformers to satisfiability is below.

**Theorem 5.** The following statements are equivalent.

1. The formula $\varphi$ is unsatisfiable.
2. The fixed point $\text{gfp}(da_\varphi)$ is $(\emptyset, \text{Env})$.
3. The fixed point $\text{gfp}(dcd_\varphi)$ is $(\emptyset, \emptyset)$.
4. The fixed point $\text{gfp}(dca_\varphi)$ is $(\emptyset, \text{Env})$.
5. The fixed point $\text{gfp}(dca_\varphi)$ is $(\emptyset, \emptyset)$.
6. The fixed point $\text{gfp}(dca_\varphi)$ is $(\emptyset, \emptyset)$.
7. The fixed point $\text{gfp}(dca_\varphi)$ is $(\emptyset, \text{Env}, \text{Env})$.
8. The fixed point $\text{gfp}(dca_\varphi)$ is $(\emptyset, \emptyset)$.
9. The fixed point $\text{gfp}(aaf_\varphi)$ is $(\text{Env}, \text{Env})$.

### 4.3 Connection to Programs

We now relate deduction and abduction transformers to transformers generated by programs. Assume a first-order signature $\text{Sig}$ and variables $\text{Vars}$ as before. We write $\text{assum}[b]$, abbreviated to $[b]$, for an assumption statement with a quantifier-free formula $b$. The operational semantics of the statement is below.

$$
\text{rel}([b]) \doteq \{ (\varepsilon, \varepsilon) \mid \varepsilon \in [b] \}
$$

The context for a program is the set of variables in the program. The operational semantics defines the four transformers below, which are related to deduction and abduction in Theorem 6.

$$
\text{post}_[b] \doteq X \rightarrow X \cap [b] \quad \text{post}_[b] \doteq \lnot \circ \text{post}_[b] \circ \lnot \\
\text{pre}_[b] \doteq X \rightarrow X \cap [b] \quad \text{pre}_[b] \doteq \lnot \circ \text{pre}_[b] \circ \lnot
$$

**Theorem 6.** For a quantifier-free test $\varphi$ we have that $\text{ded}_\varphi = \text{post}_[\varphi]$ and $\text{abd}_\varphi = \text{pre}_[\varphi]$.

The consequence of Theorem 6 is that the same transformers can be used for deduction and abduction in an SMT solver or for reasoning about conditionals in program analysis. Improvements in solvers lead to improved reasoning about conditionals and vice versa. Moreover, the Galois connection between deduction and abduction is a special case of the classic Galois connection between postcondition and precondition transformers [7].

What of assignments? A beautiful result of categorical logic shows that substitution defines a transformer that has two adjoints, which generalise universal and existential quantification [32]. Transformers for assignments are closely related to transformers for quantification. Consequently, assignment transformers, which are ubiquitous in program analysis, define approximate quantification procedures. Improvements in quantifier elimination procedures should lead to better transformers for assignments. Due to space restrictions, we do discuss this connection further.

### 5. Abstract Reasoning

This section presents three ideas. The first is a standard application of abstract interpretation to the collecting semantics of formulae: if concrete transformers are replaced by abstract transformers, we obtain sound but incomplete conclusions about the properties of a formula. The second is the notion of an abstract reasoning domain, which provides the building blocks for SMT solvers based on abstract interpretation in the same way that traditional abstract domains are building blocks of program analyzers. The third idea is that standard logical notions such as definability or completeness are properties of a Galois connection between lattices of structures and lattice of formulae.

**Abstract Interpretation** We recall essential notions of abstract interpretation. Assume two posets $C$ and $A$ related by a Galois connection between an abstraction function $\alpha : C \rightarrow A$ and a concretisation function $\gamma : A \rightarrow C$. The fundamental fixed point approximation theorem of abstract interpretation is below, with $aF$ representing the abstract transformer corresponding to $F$.

**Theorem 7** ([7]). Let $(C, \alpha, \gamma, A)$ be a Galois connection between two complete lattices and $F : C \rightarrow C$ and $aF : A \rightarrow A$ be transformers satisfying $\alpha \circ F \subseteq aF \circ \alpha$. Then, $(\alpha(\text{lfp}(F))) \subseteq \text{lfp}(aF)$ and $(\alpha(\text{gfp}(F))) \subseteq \text{gfp}(aF)$.

In the case that $C$ is a powerset lattice $\mathcal{P}(S)$, we call $A$ overapproximating if $X \subseteq \gamma(\alpha(X))$ for all $X \subseteq S$ and underapproximating if $\gamma(\alpha(X)) \subseteq X$ for all $X \subseteq S$. An abstract transformer $aF : A \rightarrow A$ is a sound overapproximation of a concrete transformer $F : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ if $F \circ \gamma \subseteq \gamma \circ aF$ and is a sound underapproximation if $F \circ \gamma \supseteq \gamma \circ aF$.

An abstract transformer $aF$ is $\alpha$-complete for $F$ at $c$ if it satisfies $\alpha(F(c)) = aF(\alpha(c))$ and is $\alpha$-complete if $\alpha \circ F = aF \circ \alpha$. We say that $F$ is $\alpha$-fixed point complete if $\alpha(\text{lfp}(F)) = \text{lfp}(aF)$. An abstract transformer $aF$ is $\gamma$-complete for $F$ at $a$ if $\gamma(\text{gfp}(F)) = \text{gfp}(aF)$ and is $\gamma$-complete if $\gamma \circ aF = F \circ \gamma$.

### 5.1 Abstract Interpretation of Formulae

We introduce abstract $\text{Sig}$-structures which are derived from collecting $\text{Sig}$-structures by replacing concrete transformers with abstract transformers. Evaluating a formula with respect to an abstract $\text{Sig}$-structure allows us to approximate the semantics of a formula in a predictable manner.

**Definition 8.** An abstract $\text{Sig}$-structure $A$ consists of the following complete lattices and transformers.

1. Abstract value lattices $(a\text{Val}, \subseteq, \sqcap, \sqcup)$ for each $n \in \mathbb{N}$.
2. Abstract environment lattices $(a\text{Env}, \subseteq, \sqcap, \sqcup)$ for each $\Gamma$. 

3. Abstract translation functions $\text{av-to-}\Gamma: a\text{Val}_n \rightarrow a\text{Env}_\Gamma$ and $\text{ae-to-}\Gamma: a\text{Env}_\Gamma \rightarrow a\text{Val}_n$ whenever $\text{len}(\Gamma) = n$.
4. An abstract weakening function $\text{awk}_\Gamma: a\text{Env}_\Gamma \rightarrow a\text{Env}_\Gamma$, for each context $\Gamma$ and subcontext $\Gamma'$.
5. Projection functions $\text{acp}_\Gamma: a\text{Env}_\Gamma \rightarrow a\text{Env}_{\Gamma'}$, for every subcontext $\Gamma'$ of $\Gamma$.
6. An abstract interpretation $\text{aint}(F): a\text{Val}_{ar}(F) \rightarrow a\text{Val}_I$ and its inverse $\text{ainv}(F): a\text{Val}_I \rightarrow a\text{Val}_{ar}(F)$ for every function symbol $F$ in $\text{Sig}$.
7. An abstract interpretation $\text{aint}(P) \in a\text{Val}_{ar}(P)$ of every predicate symbol $P \in \text{Sig}$.

Every abstract lattice above is related to the corresponding concrete lattice by a Galois connection $(C, \alpha, \gamma, A)$. We extend the convention for collecting structures and write $\text{asome}_x$ and $\text{aall}_x$ for elimination of a single variable.

The abstract semantics of terms and of formulae with respect to an abstract $\text{Sig}$-structure is given by the functions below.

$$\begin{align*}
\mathcal{L}_A: \text{Term}_\Gamma^n & \rightarrow a\text{Env}_\Gamma \rightarrow a\text{Val}_n \quad \mathcal{L}_A: \text{Form}_\Gamma & \rightarrow a\text{Env}_\Gamma
\end{align*}$$

We obtain these by replacing concrete transformers and semantics by their abstract counterparts in the definition $\mathcal{L}_\Gamma$ given earlier. Existing abstract domains already implement several components of Definition 8. These domains have abstract value and environment domains. The abstract semantics of terms is called ‘forward abstract interpretation of expressions’ and the abstract semantics of a quantifier-free formula is called ‘backward abstract interpretation of expressions’ in [8].

The abstract semantics of a quantifier-free $\varphi$ overapproximates the concrete semantics if $[\varphi: \Gamma]_C \subseteq \gamma([\varphi: \Gamma]_A)$. Underapproximation is dually defined. Abstract domains in program analysis need not have a representation of the empty set. For example, the one-element abstraction is a sound overapproximation [10]. When dealing with satisfiability, a representation of the empty set is necessary to express that a formula is unsatisfiable. Soundness, as defined below, includes this condition. The one-element abstraction is not necessarily sound in the sense below.

**Definition 9.** An abstract $\text{Sig}$-structure $A$ soundly overapproximates a concrete $\text{Sig}$-structure $C$ if every lattice in $A$ satisfies $\gamma(\bot) = \emptyset$ and concrete and abstract transformers satisfy $F \circ \gamma \subseteq \gamma \circ aF$. An abstract transformer $aF$ is the best abstract transformer if $aF = \alpha \circ F \circ \gamma$.

When dealing with underapproximations, these definitions are dualised. The soundness of abstraction for negation-free formulae is given below. We discuss negation more detail shortly.

**Theorem 10.** If $A$ is a sound overapproximation of $C$ and $\varphi$ is negation-free $[\varphi: \Gamma]_C \subseteq \gamma([\varphi: \Gamma]_A)$.

**The Interval Domain** We present an extended example of abstractly interpreting quantified formulae over intervals. It is known that interval propagation can be used reason about terms, but the example we present shows that it also supports sound but incomplete reasoning about quantifiers. The example also exhibits a difference between abstract quantifiers and best abstract quantifiers.

The complete lattice of intervals $\{\text{Intv}, \subseteq, \cap, \cup, \neg\}$ is defined over the set $\text{Intv}^n = \{[a, b] | a \leq b, a, b \in \mathbb{Z} \cup \{-\infty, \infty\}\}$, with $\bot$ representing the empty interval and $\top$ being $[-\infty, \infty]$. The interval structure $\text{Intv}$ contains the items below.

1. The value lattices $a\text{Val}_n$ are product lattices $\text{Intv}^n$.
2. The interval environments $a\text{Env}_\Gamma$ are $\text{var}(\Gamma)$.
3. The weakening function extends an interval with the new variable going to $\top$.

$$\text{awk}_\Gamma: \varepsilon \mapsto \varepsilon \cup \{y \mapsto \top | y \in \text{var}(\Gamma) \setminus \text{var}(\Gamma')\}$$

4. The projection functions are given below.

$$\begin{align*}
\text{acp}_\Gamma: \varepsilon \mapsto \varepsilon \cup \{x \mapsto \varepsilon(x) | x \in \text{var}(\Gamma')\}
\text{ainv}(F): a\text{Val}_{ar}(F) \rightarrow a\text{Val}_I
\end{align*}$$

Existential projection drops certain variables and universal projection is not $\bot$ only if the variables being dropped are all $\top$. The interpretation of functions and predicates depends on the signature of the theory considered.

**Example 3.** We compute the abstract semantics $[\varphi]_{\text{Intv}}$ of the formula $\varphi = x \geq 5 \land x \leq 10 \land y = 2$ using the interval domain.

$$[x \geq 5] \land [x \leq 10] \land [y = 2] = \{x \mapsto [5, 10], y \mapsto \{2\}\}$$

The domain also supports abstract quantification.

$$\begin{align*}
[\exists y. \varphi] = \text{asome}_y([\varphi]) & = \{x \mapsto [5, 10]\}
[\forall y. \varphi] = \text{aall}_y([\varphi]) & = \bot
\end{align*}$$

Now consider $\psi = \varphi \land y = 3$ whose semantics is $[\psi]_{\text{Intv}} = \{x \mapsto [5, 10], y \mapsto \bot\}$. We have that $\text{asome}_y([\psi]_{\text{Intv}}) = \{x \mapsto [5, 10]\}$, but $\text{aall}_y([\psi]_{\text{Intv}}) = \bot$.

showing that the existential projection transformer we have defined is not the best possible.

**Negation** Theorem 10 only applies to negation-free formulae. Example 4 illustrates the problem of formulae with negation.

**Example 4.** Consider a logic with unary predicates of the form $x = k$ interpreted over the integers. Consider the parity lattice $\text{Par} = (\{\bot, \emptyset, \text{O}, \top\}, \subseteq)$, which represents even and odd numbers. Let $[x = k]_{\text{Par}}$ be $\text{E}$ if $k$ is even and be $\text{O}$ otherwise. The complement $\neg \psi$ in this lattice cannot be used to approximate negation. To see why, consider evaluating $\neg(x = 2)$ as $\neg[x = 2]_{\text{Par}}$, which is $\neg \text{E} = \text{O}$ but $\text{O}(\text{O})$ does not include all models of $\neg(x = 2)$. The only sound abstraction of negation is to map every element to $\top$.

More generally, in domains that allow for strict overapproximation of the semantics of a formula, the only sound abstract negation will map every element to $\top$. This problem only exists if the syntax of formulae have arbitrary negation. If negation is limited to atomic predicates, Definition 8 can be extended to include an abstract interpretation $\text{aint}(-P)$ of the negation of every predicate, Definition 9 can be extended to require sound overapproximation of negation of predicates, and Theorem 10 can be extended to formulae in negation normal form. A similar restriction is applied when model checking is combined with abstraction [5].

**5.2 Abstract Reasoning Domains**

We now identify the structure of an abstract domain for logical reasoning. In addition to a lattice that provides approximate semantics of formulae and abstract transformers that provide approximate semantics for logical operators, the domain includes operators for inductive reasoning (in the sense of philosophic logic). Unlike deductive and abductive reasoning, which are both sound with respect to implication, inductive reasoning models generalization or specialization which are not necessarily sound.

Widening and dual-widening operators implement inductive reasoning in program analyzers. Craig interpolation is another technique used for generalisation [34]. Narrowing is a restricted form of interpolation in a lattice because it maps a pair satisfying $A \subseteq B$ to an element $I$ satisfying $A \subseteq I$ and $I \subseteq B$. Narrowing does not have the syntactic constraints of Craig interpolation because the lattice is not necessarily constructed from formulae. McMillan’s
notion of interpolation, which we call separation, maps a pair satisfying $A \land B \subseteq \bot$ to an $I$ satisfying $A \subseteq I$ and $I \land B \subseteq \bot$. While interpolation and separation are inter-derivable on Boolean lattices they are not in general, hence have distinct definitions.

**Definition 11.** A reasoning domain is an extension of a abstract Sig structure with the following operations.

1. Abstract transformers $\text{aded}_\varphi$, $\text{aced}_\varphi$, $\text{aad}_\varphi$, and $\text{acabd}_\varphi$ for deduction and abduction.
2. A unary, extensive function $\text{ext} : L \rightarrow L$ called upwards extrapolation and its dual $\text{ext} : L \rightarrow L$ called downwards extrapolation.
3. A partial, interpolation function $\text{tp} : L \times L \rightarrow L$, satisfying $x \subseteq \text{tp}(x, y) \subseteq y$ whenever $x \subseteq y$. The element $\text{tp}(x, y)$ is called an interpolant.
4. A partial, separation function $\text{sep} : L \times L \rightarrow L$, satisfying $x \subseteq \text{sep}(x, y) \subseteq \text{sep}(x, y) \cap y \subseteq \bot$ whenever $x \cap y \subseteq \bot$. The element $\text{sep}(x, y)$ is called a separator.

We use the word ‘function’ and not ‘transformers’ above because the operations need not be monotone, as with widening [9].

**Example 5.** We apply abstract, interval deduction by computing $\text{gfp}(\text{aded}_\varphi)$ with $\varphi = \{x \geq 5 \land x \leq 10 \land y = 2x\} \land E_0 = \top$.

- $E_1 = \text{aded}_\varphi(E_0)$
  
  $= \{x \geq 5\}(\top) \cap \{x \leq 10\}(\top) \cap \{y = 2x\}(\top)$
  
  $= \{x \in [5, 10], y \rightarrow [-\infty, \infty]\}$

- $E_2 = \text{aded}_\varphi(E_1)$
  
  $= \{x \geq 5\}(E_1) \cap \{x \leq 10\}(E_1) \cap \{y = 2x\}(E_1)$
  
  $= \{x \in [5, 10], y \rightarrow [10, 20]\}$

The fixed point is superfluous in the concrete but yields more information in the abstract than one transformer application.

The overapproximation and underapproximation conditions given earlier provide the following abstract satisfaction theorem.

**Theorem 12.** Let $\mathcal{O}$ be an overapproximate reasoning domain and $\mathcal{U}$ be an underapproximate one.

1. If $\gamma(\text{gfp}(\text{aded}_\varphi)) = \emptyset$ in $\mathcal{O}$ then $\varphi$ is unsatisfiable.
2. If $\gamma(\text{lp}(\text{acabd}_\varphi)) = \mathcal{U}$ then $\varphi$ is unsatisfiable.

5.3 Galois Connection of Syntax and Semantics

The first abstraction we identify is between syntax and semantics. The Galois connection below has been observed in different set-tings in the literature [32, 42]. We are not aware of this connection being identified in the setting that we use.

**Theories and Definability** A theory is a set of formulae, each called an axiom. A complete theory is a set of formulae closed under implication. The theory of a set of structures $S$ consists of formulae which are true in every structure in $S$. A set of structures $S$ is definable if there exists a formula $\varphi$ such that $S = \{\varphi\}$. These notions lead to a Galois connection.

$$
\begin{align*}
\text{th} : \mathcal{P}(\text{Env}_I) & \rightarrow \mathcal{P}(\text{Form}_I) & \text{st} : \mathcal{P}(\text{Form}_I) & \rightarrow \mathcal{P}(\text{Env}_I) \\
\text{th} & \models S \rightarrow \{\varphi \mid S \subseteq \{\varphi\}\} & \text{st} & \models \Phi \rightarrow \bigcap_{\varphi \in \Phi} \{\varphi\}
\end{align*}
$$

**Theorem 13.** There is a Galois connection $(\mathcal{P}(\text{Env}) \subseteq (\mathcal{P}(\text{Form}), \supseteq)$ between structures and formulae.

The superset order is chosen because we interpret a set of formulae as their conjunction. The Galois connection allows us to view formulae as abstractions of structures and is appropriate for our goal of studying satisfiability. In a proof-theoretic investigation, one may prefer to accord priority to formulae and view structures as abstractions. By the Galois connection, a complete theory $\Phi$ is one satisfying $\text{th}(\text{st}(\Phi))$ and a set of structures $S$ is definable if $\text{st}(\text{th}(S)) = S$.

**Proof Systems** The Galois connection of syntax and semantics allows us to view proof rules as transformers on an abstract domain of formulae and logical soundness and completeness as soundness and completeness in the sense of abstract interpretation.

A proof rule is a relation between formulae. In a rule containing $(\varphi_0, \ldots, \varphi_{n-1}, \varphi_n)$ the formulae $\varphi_0, \ldots, \varphi_{n-1}$ are called antecedents and $\varphi_n$ is the consequent. A proof system is a collection of proof rules of possibly different arities. A unary rule is a set of formulae, which are also called the hypotheses.

A formula $\psi$ is derived from a set of formulae $\Phi$ using a proof system, written $\Phi \vdash \psi$, if $\psi$ occurs in $\Phi$ or if $\psi$ is derived from $\varphi_0, \ldots, \varphi_{n-1}$ by applying an $n$-ary inference rule and the formulae $\varphi_1, \ldots, \varphi_n$ are derived from $\Phi$. The deductive closure of a set of axioms $\Phi$ with respect to a proof system $\vdash$ is the set of all formulae that can be derived from $\Phi$ with $\vdash$. A proof system is deductively sound if every $\psi$ derived from $\Phi$ is in the theory of $\Phi$. A proof system is deductively complete if every $\psi$ in the theory of $\Phi$ can be derived from $\Phi$. A refutation is a derivation of false.

A proof system $\vdash$ defines a transformer $\text{aded}_\varphi : \mathcal{P}(\text{Form}) \rightarrow \mathcal{P}(\text{Form})$ from a set of formulae to its immediate consequences.

$$
\text{aded}_\varphi \models \Phi \rightarrow \{\psi \mid \text{there exist } \varphi_i \in \Phi \cup \{\varphi\} \text{ such that } \varphi_0, \ldots, \varphi_{n-1}, \psi \text{ is in a proof rule } \}
$$

The fixed point $\text{gfp}(\text{aded}_\varphi)$ (with respect to the superset order) contains all formulae derivable from $\varphi$. Since multiple applications of proof rules are required to derive a conclusion, $\text{aded}_\varphi$ is not usually a closure operator. Properties of proof systems become properties of the abstract transformer.

**Lemma 14.** A proof system $\vdash$ is deductively sound exactly if $\text{ded}_\varphi \circ \text{st} \supseteq \text{aded}_\varphi \circ \text{th}$ and is deductively complete exactly if $\text{th} \circ \text{ded}_\varphi \circ \text{st} = \text{gfp}(\text{aded}_\varphi)$.

While deductive soundness corresponds to soundness in abstract interpretation, deductive completeness is fixed point completeness, which is only one of several completeness notions [24].

6. Abstract Satisfaction Procedures

The framework in the previous section has already been applied to characterize satisfiability procedures for propositional and first-order logics in terms of the lattices and transformers. The method of truth tables, propositional resolution, and Boolean Constraint Propagation were formalized as greatest fixed points in [18]. The classic DPLL algorithm was shown to be a fixed point refinement procedure in [18], and CDCL was formalized as a combination of deduction and abduction in [19]. The congruence closure algorithm for the theory of equality with uninterpreted functions is a fixed point in a lattice of partitions, and the application of the Bellman-Ford algorithm for deciding difference logic is a fixed point in a lattice of weighted graphs [2]. The DPLL(T) algorithm used in SMT solvers was characterized as an approximate, reduced product construction in [2]. Though different frameworks were used to give a lattice-theoretic characterization of the Nelson-Oppen procedure [14] and Stålmarck’s method [44], those procedures can also be formulated in the language of this paper.

This section introduces a simple and systematic formulation of the abstractions above. We introduce lattice-theoretic generalizations of CNF and DNF formulae. A wide range of lattices used in practice, such as partial assignments, equality graphs, the intervals, and binary implication graphs, admit such a representation. We
exhibit Galois connections showing that CNF and DNF and their
generalizations define semantic abstractions. Moreover, these lat-
tices support logical notions such as resolution, the pure literal rule,
and subsumption. We demonstrate that pre- and in-processing
structures used in SAT solvers can be understood as abstract quantifica-
tion procedures. Thus, these procedures have a semantic justifica-
tion, abstract interpretation-based soundness proofs, and generalize
to program analysis.

**Upwards and Downwards Closure** We will use the notions of
upward and downward closure to generalise CNF and DNF formulæ
to logics defined over posets.

A subset $Q$ of a poset $A$ is called a *downset* if for every $x$ in $Q$ and $y$ in $A$, $y \subseteq x$ implies that $y$ is in $Q$. A downwards-closed
set is called a *downset*. The smallest downset containing $Q$ is de-
noted $Q^\downarrow$, and the downset of a singleton set $\{x\}$ is denoted $x^\downarrow$. In examples, we denote a downset as the set of its maximal ele-
ments. The *downset lattice* over $A$, written $(D(A), \subseteq, \cap, \cup)$, is the set of downsets of $A$ ordered by inclusion. Downsets strictly gen-
eralise powersets because the downset lattice with respect to the
identity relation is isomorphic to $\mathcal{P}(S)$. The dual notion to down-
set is *up-sets*. The smallest up-set containing $Q$ is $Q^\uparrow$, and $(\cup(A), \supseteq)$ is the *up-set* lattice with intersection as join and union as meet.

Let $\min(Q)$ and $\max(Q)$ denote the minimal and maximal ele-
ments of a poset $Q$. These sets form antichains. When convenient,
we assume that up-sets are represented by minimal elements and
downsets by maximal elements.

An abstraction $A$ of a powerset lattice is *disjunctive* if $\gamma(a \cup
b) = \gamma(a) \cup \gamma(b)$. Downset completion is an operation that en-
riches an abstraction with disjunction [12]. The *downset completion* of $A$ is the lattice $\mathcal{D}(A)$ with the abstraction and concretisation functions
below. Unlike the standard treatment, we use downsets as underap-
proximating abstractions.

$$
\gamma_{D}(A) : \mathcal{D}(A) \rightarrow \mathcal{P}(S)
\gamma_{D}(A)(Q) \doteq \bigcup \{ \gamma(x) \mid x \in Q \}
\alpha_{D}(A) : \mathcal{P}(S) \rightarrow \mathcal{D}(A)
\alpha_{D}(A)(P) \doteq \{ x \mid \gamma(x) \subseteq P \}
\gamma_{U}(A) : \mathcal{U}(A) \rightarrow \mathcal{P}(S)
\gamma_{U}(A)(Q) \doteq \bigcap \{ \gamma(x) \mid x \in Q \}
\alpha_{U}(A) : \mathcal{P}(S) \rightarrow \mathcal{U}(A)
\alpha_{U}(A)(P) \doteq \{ x \mid \gamma(x) \supseteq P \}
$$

Consult [12] for proofs that the pairs of functions above form
Galois connections and that the domains are disjunctive. We can
dually define the up-set completion of a lattice. If a lattice $A$ overapproximates $\mathcal{P}(S)$, the downset completion overapproximates
$\mathcal{P}(S)$ and the up-set completion underapproximates $\mathcal{P}(S)$.

6.1 Generalised Cube Abstract Domains

There is some debate about whether the use of CNF representations
is beneficial or detrimental to solvers [43]. We believe that CNF is
advantageous for solvers because it leads to simple and efficient
data structures. We also believe there must be deeper, algebraic
properties of CNF that are advantageous to solvers. One reason is
because, as we observed earlier, CNF leads to a simpler treatment of
negation. Another is that several domains used in practice have
generalized CNF representations.

Consider a poset $(gLit, \subseteq)$ of *generalised literals*. One may
think of $gLit$ as a set of semantically distinct formulæ. We define
generalised cubes, clauses, CNF and DNF formulæ by using up-sets
to form conjunctions and downsets to form disjunctions.

$$
gClause \doteq (D(gLit), \subseteq) \quad gCube \doteq (U(gLit), \supseteq)
gCNF \doteq (U(gClause), \supseteq) \quad gDNF \doteq (U(gCube), \subseteq)
$$

Since up- and downsets become powersets if the identity relation is
used as the order, standard cubes and clauses are special cases of
this definition. If $gLit$ contains formulæ, the concretisation function
for the poset $A$ is the function $st : gLit \rightarrow \mathcal{P}(Env)$

![Figure 2. Refinement order between the generalised CNF domains.](image)

<table>
<thead>
<tr>
<th>$gLit$</th>
<th>$gCube$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$({ p, \neg p \mid p \in Prop }, =)$</td>
<td>Partial assignments</td>
</tr>
<tr>
<td>$({ x = y, x \neq y \mid x \inVars }, =)$</td>
<td>Equality graphs</td>
</tr>
<tr>
<td>$({ x &lt; y, x \neq y \mid x \inVars }, =)$</td>
<td>Difference graphs</td>
</tr>
<tr>
<td>$({ x \leq k, x \geq k \mid k \in \mathbb{N} }, \geq)$</td>
<td>Interval cubes</td>
</tr>
<tr>
<td>$({ m \lor n \mid m, n \in Lit }, =)$</td>
<td>Binary Implication Graphs</td>
</tr>
</tbody>
</table>

Table 1. Domains viewed as generalised clauses, cubes and CNF
from formulæ to models. If $a$ is a generalized literal, we write $\sim a$
for a literal that satisfies $\gamma(a) \cup \gamma(\sim a) = T$.

The relationship between these domains is depicted in the Hasse
diagram in Figure 2, where an upward line denotes an abstraction
relationship. Cubes constructed by powerset operations are over-
approximations of generalised cubes and CNF formulæ, which are
in turn overapproximations of generalised CNF, which are overap-
proximations of environments. On the other hand, the dual con-
structions give us underapproximating domains of clauses, gener-
alised clauses, DNF and generalised DNF respectively.

Another property of this construction is that all domains ob-
tained are distributive because up-set and downset lattices are distri-
butive [15]. As we show shortly, these domains are distributive
lattices but do not express disjunction. Table 1 lists domains that
have generalized cube descriptions.

6.2 Partial Assignments

Consider a set of Boolean variables $Prop$, a set of literals $Lit$ and the identity relation. The lattice $gCube$ is iso-
morphic to the lattice of partial assignments, the main data structure
in solvers based on DPLL. This domain was studied in [18].

**Equality Graphs** Consider generalized clauses over the literals
$\mathcal{ELit} = \{ x = y, x \neq y \}$ with the identity relation on literals.
(Caution! Identity between literals is different from the equality
between variables.) Cubes over equality literals define equality
graphs, which are used in several solvers [36, 47]. Figure 3 depicts
a graph with elements representing literals highlighted at the top.
Two important operations in equality logic decision procedures
are transitive closure and cycle detection. We formalize transitive
closure by the transformer $trans : U(\mathcal{ELit}) \rightarrow U(\mathcal{ELit})$.

$$
trans \doteq G \rightarrow G \cup \{ x = y \mid x = z, z = y \in G \}
$$

Recall that a reduction operator is an abstract transformer $\rho$ satis-
fying that $\gamma(a) = \gamma(\rho(a))$ for all $a$. Observe that transitive closure
is a reduction operator and the saturation of a graph with transitive
edges is a greatest fixed point.

A *conflicting cycle* in an equality graph contains only equalities
and exactly one disequality edge [36]. A graph with a conflicting

![Figure 3. A conflicting cycle in an equality graph.](image)
cycle satisfies $\gamma(G) = \emptyset$. The shaded elements at the bottom of
Figure 3 all contain conflicting cycles and concretise to the empty
set. A similar abstract domain and reduction operation appear in
the logic of order and directed graphs in [31].

**Interval Cubes** Consider the set of single variable inequalities
$\{x < k, x \geq k \mid k \in \mathbb{Z}\}$, which form a poset under
implication. The elements of BLit are light grey in Figure 4. By representing intervals as cubes instead of pairs, we obtain a set-
based representation that supports proof rules such as resolution
and other algorithms in solvers that target clauses and cubes. Example 6 illustrates how intervals can be manipulated using up-sets.

Example 6. The interval domain as defined in Section 5 is not
distributive because identities such as the one below fail.

$$(0, 1] \cup [6, 7]) \cap [3, 4] = [3, 4]$$

Now consider intervals represented as up-sets. The meet in this
lattice is union and join is intersection, so it is trivially distributive.
We represent up-sets by their minimal elements.

$$(\{x \geq 0, x \leq 1\} \cap \{x \geq 6, x \leq 7\}) \cup \{x \geq 3, x \leq 4\}$$

$${} = \{x \geq 0, x \leq 7\} \cup \{x \geq 3, x \leq 4\} = \{x \geq 3, x \leq 4\}$$

A similar calculation yields the expected interval for the other
interval expression. The cube representation expresses the same
concrete elements as the classic pair representation of intervals but
contains more redundancy.

**Binary Implication Graphs** If the set of generalized literals contains
classes of length 2 (such as $\{p, \neg q\}$) ordered by equality, each
generalised literal represents two implications $q \Rightarrow p$ and $\neg p \Rightarrow q$,
which can be viewed as edges in a directed graph. The resulting
Binary Implication Graph abstract domain is used for preprocessing
in SAT solvers [29]. Though a lattice-based analysis has not been
previously applied to Binary Implication Graphs, we need not
explicitly define Galois connections or their approximation properties,
as these follow from the cube representation. As with equality
graphs, transitive closure is a reduction.

**The CNF Domains** The generalized CNF domains show that CNF
formulae have a lattice structure that is usually not recognized.
To see the difference between $g$CNF and CNF, consider $BLit$. The
set $\{x \geq 2, x \leq 5, x \leq 7\}$ represents a clause but not a generalized
clause, while $\{x \geq 2, x \leq 5\}$ is a generalized clause.

Solvers use a variety of subsumption techniques to minimize
formulae without explicitly checking implication. The inclusion
orders in Figure 2 correspond to subsumption notions and also
underapproximate implication. For instance, $\{\{p\} : \{p, q\}\}$ and
$\{\{p\} : \{p, q\}\}$ are logically equivalent but we can only show $\{\{p\}\} \subseteq
\{\{p\} : \{p, q\}\}$ using the lattice order. From the abstract interpretative
perspective, subsumption is not a substitute for implication but
is fundamental to the lattice structure of CNF and DNF domains.

### 6.2 Abstract Transformers in Solvers

**Generalized Unit Rule** The unit rule in SAT solvers asserts that if
each cube represents a region of the search space and the cube falsifies
all but one literal in a clause, the remaining literal can be added
to the clause. The Abstract Conflict Driven Learning algorithm
(ACDCL) of [19] generalises the unit rule to abstract domains based
on the notion of complementable meet irreducibles. The unit rule
can be viewed as a technique for refining generalized cubes using
generalized clauses.

$${\text{gunit}} : g\text{Clause} \rightarrow (g\text{Cube} \rightarrow g\text{Cube})$$

$${\text{gunit}}_{\theta} \equiv \pi \mapsto \pi \cup \{\theta\}, \text{where} \theta \in \theta, \gamma(\pi) \cap \gamma(\phi) \neq \emptyset, \text{and}
\text{for all } b \in \theta \setminus \{\theta\}, \gamma(\pi) \cap \gamma(b) = \emptyset$$

We write $\text{gfp}_{\phi}(f)$ for the function that maps elements $a$ to fixed
points $\text{gfp}_{\phi}(f(a, x))$ of a function $f(y, x)$. The main observation
of [18] was that Boolean Constraint Propagation (BCP) is a fixed
point $\text{bcp}(\varphi, x) \equiv \text{gfp}_{\phi}(\bigcap_{\phi \in \text{gunit}_{\theta}})$ defined poitwise over unit
rules. The generalized cubes in Table 1, when combined with the
generalized clauses, also support the unit rule and BCP.

**Failed Literal Probing** Failed literal probing [21, 22] is a prepro-
cessing technique in SAT solvers. The technique chooses literal $\alpha$,
computes $\text{bcp}(\alpha, \varphi)$ and if the result is $\perp$, adds the singleton clause
$\{b\}$ to $\varphi$, where $b$ satisfies that $\gamma(\alpha) \cup \gamma(b) = \top$. No action is
taken if the result is not bottom. Note that we have not only described
failed literal probing, but also its generalization to CNF domains.

**Clause Dropping** Variable elimination is a fundamental operation
underlying quantifier elimination, deduction and syntactic simpli-
ification of formulæ. The simplest form of sound variable elimination
is to drop clauses from a CNF formulà in which the target variable
occurs. This idea lifts directly to generalized CNF domains if we drop
constraints based on the literals they contain.

$${\text{drop}} : g\text{Lit} \times g\text{CNF} \rightarrow g\text{CNF}$$

$${\text{drop}}_{\phi}(\varphi) \equiv \{C \mid \{p, \neg p\} \cap C = \emptyset, C \in \varphi\}$$

In general, $\text{drop}_{\phi}$ is an overapproximation of the deduction trans-
former. If $p$ is a Boolean variable, $\text{drop}_{\phi}$ also overapproximates
existential quantification with respect to $p$. 

![Figure 3. Equality graphs over three variables. The shaded ele-
ments at the top are literals and the shaded elements at the bottom
all concretise to the empty set.](image)

![Figure 4. The lattice of interval environments over two variables.
The shaded elements at the top represent the poset of literals while
elements at the bottom represent singleton values.](image)
Resolution The resolution principle asserts that if \( C \lor p \) and \( \neg p \lor D \) are both satisfiable, so is \( C \lor D \). The variable \( p \) is the pivot and \( C \lor D \) is the resolvent. Generalised CNF domains support a generalisation of resolution: \( g\text{Lit} \times g\text{CNF} \rightarrow g\text{CNF} \).

\[
\text{res}(p, \varphi) \equiv \{ C \lor D \mid C \lor \{ p \}, D \lor \{ q \} \in \varphi, \gamma(p) \land \gamma(q) = \emptyset \}
\]

Generalized CNF domains that apply this transformer can produce resolution-style proofs, which is a first step towards combining existing abstract domains with proof-theoretic techniques.

Example 7. The generalised CNF element

\[
\varphi \equiv \{ \{ x \geq 10, x \leq 5 \} \mid \{ x \geq 7, x \leq 13 \} \}
\]

represents a conjunction of disjunctions of bounds. Standard interval propagation will lose precision in the clauses, while standard resolution does not apply because no constraint is the negation of another. The result of generalized resolution with respect to the literal \( x \geq 7 \) is \( \text{res}(x \geq 7, \varphi) = \{ \{ x \geq 10, x \leq 13 \} \} \). Though arithmetic techniques methods can deduce the same information, generalised resolution is simple and suffices in this case.

The Pure Literal Rule Clause dropping is a sound but incomplete simplification technique because a formula \( \varphi \) that is unsatisfiable might become satisfiable after clause dropping. The pure literal rule, introduced in the original algorithm of Davis and Putnam [16], can be understood as an application of clause dropping only in situations where it does not change the satisfiability of a formula. We formalize the pure literal rule for generalized CNF domains.

Recall that \( g\text{Lit} \) is the set of generalized literals. We say that a set of literals \( A \) is pure if for each \( a \in A \), no \( b \) satisfying \( \gamma(a) \land \gamma(b) = \emptyset \) is also in \( A \). We assume that \( g\text{Lit} \) is the disjoint union of two pure sets \( P \) and \( N \) such that for each \( a \in P \), there exists a \( b \) in \( N \) satisfying that \( \gamma(a) \land \gamma(b) = \emptyset \) and vice-versa. The elements of \( P \) are positive literals and of \( N \) are negative literals. Define a set \( \{ +, - \} \) of polarities, a lattice of polarities \( P \rightarrow P(\{ +, - \}) \), and a lattice of polarity maps \( P \rightarrow P(\{ +, - \}) \).

Polarity analysis of a generalised CNF formula \( \varphi \) computes a polarity map \( \rho \), such that \( \rho(\ell) \in \{ + \} \lor \{ - \} \) if the generalized literal \( \ell \) only occurs positively or only negatively in \( \varphi \), and is \( \emptyset \) if \( \ell \) does not occur in \( \varphi \), and is \( \{ +, - \} \) if \( \ell \) occurs both positively and negatively in \( \varphi \). The polarity of a literal \( \ell \) is \( \{ + \} \) if \( \ell \) is in \( P \), and is \( \{ - \} \) if \( \ell \) is in \( N \). The polarity map for a literal \( \ell \) sends \( \ell \) to its polarity and sends all literals \( b \neq a \) to the empty set. The polarity map for a generalized clause is the pointwise join of of polarity maps of its literals. The polarity map of a generalized CNF element is the pointwise join of of polarity maps of its clauses. Note that the join is used in both cases.

The pure literal rule for a generalized CNF formula, applies \( \text{drop}_\varphi \) to a formula \( \varphi \) only if the polarity map for \( \varphi \) does not send the literal \( \ell \) to \( \{ +, - \} \). The pure literal rule is a sound overapproximation of the deduction transformer and is a refinement of clause dropping. In propositional logic, the pure literal rule applied to a variable \( p \) is also a sound abstraction of the existential quantification transformer \( \text{wk}_p \circ \text{some}_p \).

7. Discussion and Related Work

This paper contributes to a research programme that seeks to close the gap between abstract interpretation techniques and deduction algorithms, both in theory and practice. One direction of this program is to use deduction algorithms to refine static analyses. Precision loss due to joins was reduced by boosting a static analysis with unification [46], DPLL(T) [27], or CDCL [20]. Best abstract transformers have been synthesized using satisfiability solvers [41], and Stålmarck’s method [45], while [37] applied satisfiability techniques to reduce precision loss in fixed point iteration.

Another direction in this programme is to characterize satisfiability procedures as abstract interpretations. Boolean constraint propagation was shown to be an abstract interpretation in [18], and CDCL was generalized to combine deduction and abduction over lattices in [19]. Stålmarck’s method was characterized as a technique for refining abstract transformers in [44]. The Nelson-Oppen method for theory combination implemented in SMT solvers was shown to be a special case of the reduced product of abstract domains [14]. The DPLL(T) technique for reasoning about a theory by combining a SAT solver with a theory solver was also shown to be a special case of the reduced product in [2].

In addition to static analysis applications, these characterizations provide a new way for lifting solver algorithms to new logics. For example, Stålmarck’s method was lifted to arithmetic in [45], while CDCL was lifted to floating point logic in [26]. Note that the Nelson-Oppen combination was lifted to abstract domains [25] prior to the reduced product characterization.

Abstract satisfaction is lattice-theoretic in an attempt to align with static analysis. If static analysis is ignored, the DPLL(T) framework provides one generic approach to implementing decision procedures [23]. The separation of Boolean and theory reasoning in DPLL(T) can be detrimental to performance and has driven the search for other frameworks. Abstract DPLL [39], natural domain SMT [6], generalized DPLL [35], and the model construction calculus [17] are attempts in this direction.

8. Conclusion

Abstraction is fundamental to practical reasoning about computationally intractable problems. Abstract interpretation has traditionally been applied to reason about undecidable problems such as checking semantic properties of programs. This paper introduced a framework for applying abstract interpretation to problems that are NP-hard but decidable, such as satisfiability.

This framework allows for novel perspectives of SMT algorithms. Solvers can be viewed as abstract interpretation portfolios, which combine several different, weak abstractions to achieve a conclusive result. Moreover, while solvers use incomplete abstractions they produce complete results. This is not due to brute-force enumerations but clever, semantics-based refinement techniques. Our framework makes some of these techniques explicit, but more importantly provides a general vocabulary for studying a wide range of satisfiability procedures as abstract interpretations. Boolean constraint propagation was shown to be an abstract interpretation in [18], and CDCL was generalized to combine deduction and abduction over lattices in [19]. Stålmarck’s method was characterized as a technique for refining abstract transformers in [44]. The Nelson-Oppen method for theory combination implemented in SMT solvers was shown to be a special case of the reduced product of abstract domains [14]. The DPLL(T) technique for reasoning about a theory by combining a SAT solver with a theory solver was also shown to be a special case of the reduced product in [2].

In addition to static analysis applications, these characterizations provide a new way for lifting solver algorithms to new logics. For example, Stålmarck’s method was lifted to arithmetic in [45], while CDCL was lifted to floating point logic in [26]. Note that the Nelson-Oppen combination was lifted to abstract domains [25] prior to the reduced product characterization.

Abstract satisfaction is lattice-theoretic in an attempt to align with static analysis. If static analysis is ignored, the DPLL(T) framework provides one generic approach to implementing decision procedures [23]. The separation of Boolean and theory reasoning in DPLL(T) can be detrimental to performance and has driven the search for other frameworks. Abstract DPLL [39], natural domain SMT [6], generalized DPLL [35], and the model construction calculus [17] are attempts in this direction.
Acknowledgments

This work was supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/H017585/1, and the FP7 STREP PINCETTE. The research reported in this paper was conducted between 2011 and 2013. In 2011, Vijay D’Silva was supported by a Microsoft Research scholarship.

References