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

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.

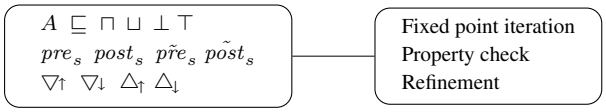
Categories and Subject Descriptors F.4.1 [*Mathematical Logic*]: Mechanical Theorem Proving; I.2.3 [*Deduction and Theorem Proving*]: Deduction

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].

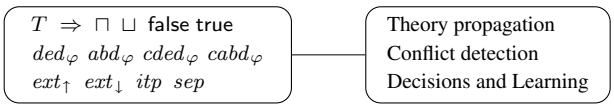
Abstract interpretation is a lattice-theoretic framework for reasoning about fixed points [10, 13]. The idiomatic approach to applying abstract interpretation to a problem is to characterise solutions to the problem by fixed points, identify a space of fixed point approximations, and design an algorithm 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 $(A, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$ 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 pre_s maps a to states the program *may* have come from before executing s, while the successor transformer $post_s$ maps a to states the program *may* reach after executing s. The transformers $\tilde{pre}_s$ and $\tilde{post}_s$ 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 ($\triangledown\uparrow$) or dual widening ($\triangledown\downarrow$) operator to accelerate convergence. If the result is not precise enough, a narrowing ($\triangle\downarrow$) or dual narrowing ($\triangle\uparrow$) 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 $(T, \Rightarrow, \sqcap, \sqcup, \mathsf{false}, \mathsf{true})$. A solver can use *deduction* to compute facts implied by φ, or use *abduction* to compute facts that imply φ. These operations define deduction and abduction transformers (ded_φ and abd_φ), and their counterparts ($cded_\varphi$ and $cabd_\varphi$) for contrapositive reasoning.



POPL '14, January 22–24, 2014, San Diego, CA, USA.

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

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 $ext_\uparrow$ and $ext_\downarrow$), while techniques like subsumption and clause minimization correspond to a relaxation of narrowing, which we call *interpolation* (denoted $int_\uparrow$ and $int_\downarrow$). 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 m-*termed* A-*sequence* is a function $\bar{s} : \{0, \ldots, m-1\} \to A$, whose length m is denoted $len(\bar{s})$. We write $f(\bar{s})$ for the application $f(s_0, \ldots, s_{n-1})$ and leave implicit that $\bar{s}$ has length n. Given a function $g : A \to C$, 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 $g[\bar{a} \Mapsto \bar{c}]$.

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 $f(x) \sqsubseteq x$ 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. 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.

A lattice is *distributive* if every x, y and z satisfy $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$, 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 $x \sqcap y = \bot$ and $x \sqcup y = \top$. 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 $\tilde{f} \mathrel{\hat{=}} \neg \circ f \circ \neg$. The *powerset lifting* of $f : A \to B$ is the function $f : \mathcal{P}(A) \to \mathcal{P}(B)$ 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 $\mathsf{gfp}(f)$ and $\mathsf{lfp}(f)$.

Galois Connections Let $(L, \sqsubseteq)$ and $(M, \preccurlyeq)$ be posets. Two functions $\alpha : L \to M$ and $\gamma : M \to L$ form a *Galois connection* if for all $x \in L$ and $y \in M$, $\alpha(x) \preccurlyeq y$ if and only if $x \sqsubseteq \gamma(y)$. A Galois connection is written as $L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} M$ or (L, α, γ, M). The function α is called the *left adjoint* and γ is called the *right adjoint* of the Galois connection. In a Galois insertion, α 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 (Sig, ar) consists of disjoint sets $Sig = Pred \cup Fun$ of predicate and function symbols whose arity is $ar : Sig \to \mathbb{N}$. 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 $Vars$ be a set of variables and x, y, z range over variables. A *first-order context* Γ is a finite sequence of variables in which each variable occurs exactly once. We write $[]$ for the empty sequence, Γ, Γ' for sequence concatenation, and $var(\Gamma)$ 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 Γ' is a *subcontext* of Γ if $var(\Gamma') \subseteq var(\Gamma)$.

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 (VAR) and function composition (FUN), we use a rule (SEQ) for forming a sequence of terms. The rule for function composition has the side condition that $ar(F)$ is $len(\bar{t})$. We leave such side conditions implicit in the remaining rules.

$$\textsc{var}\ \frac{}{x : \Gamma, x, \Gamma'} \qquad \textsc{fun}\ \frac{\bar{t} : \Gamma}{F(\bar{t}) : \Gamma} \qquad \textsc{seq}\ \frac{t_0 : \Gamma \cdots t_{n-1} : \Gamma}{\bar{t} : \Gamma}$$

From these rules, we can derive the rules for complete substitution (CSUB) and weakening (WEAK) given below.

$$\textsc{csub}\ \frac{t : \bar{x} \qquad r_0 : \Gamma' \cdots r_{n-1} : \Gamma'}{t[\bar{x} \mapsto \bar{r}] : \Gamma'} \qquad \textsc{weak}\ \frac{t : \Gamma}{t : \Gamma, \Gamma'}$$

Terms-in-context are composed with predicate symbols and Boolean operations to obtain *formulae-in-context*. We henceforth abbreviate 'formula-in-context' to 'formula'. In the Boolean operator rule below, op is one of true, false, $\vee$, $\wedge$ or $\neg$ applied to the appropriate number of arguments.

$$\textsc{pred}\ \frac{\bar{t} : \Gamma}{P(\bar{t}) : \Gamma} \qquad \textsc{op}\ \frac{\varphi : \Gamma \qquad \psi : \Gamma}{\varphi\ op\ \psi : \Gamma}$$

The weakening rule for formulae is similar to that for terms. Quantification changes the set of free variables in a formula and causes contraction of a context.

$$\exists\text{-Q}\ \frac{\varphi : \Gamma, x, \Gamma'}{\exists x.\varphi : \Gamma, \Gamma'} \qquad \forall\text{-Q}\ \frac{\varphi : \Gamma, x, \Gamma'}{\forall x.\varphi : \Gamma, \Gamma'}$$

The sets of terms and formulae in a context Γ are denoted $Term_\Gamma$ and $Form_\Gamma$, respectively. An *atomic predicate* is the composition of a predicate symbol with terms and a *literal* is an atomic predicate or its negation. A *clause* is a disjunction of literals and a *cube* is a conjunction of literals. A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses and one in *Disjunctive Normal Form* (DNF) is a disjunction of cubes. The sets containing these formulae are denoted Lit_Γ, $Clause_\Gamma$, $Cube_\Gamma$, CNF_Γ, and DNF_Γ, respectively. If not specified, the formulae we deal with are quantifier-free.

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 Sig-interpretation $\mathcal{M} = (Val, int)$, which consists of a universe Val and an interpretation that maps each function symbol F to a function $int(F) : Val^{ar(F)} \to Val$ and each predicate symbol P to a relation $int(P) \subseteq Val^{ar(P)}$.

An *environment* over Γ maps variables in Γ to values. Let $Env_\Gamma \mathrel{\hat{=}} var(\Gamma) \to Val$ be the set of environments over Γ. 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 $Env_\Gamma \to Val$ is isomorphic to $Val^{len(\Gamma)}$. 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 powersets 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 $\models$ 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 $\mathcal{P}(Val^n)$ represent the domains over which function symbols are interpreted. The lattices of environments $\mathcal{P}(Env_\Gamma)$ represent the domains over which terms-in-context are interpreted. We use the function $v\text{-}to\text{-}e_\Gamma$ that maps values to environments to deal with substitution into a term, and the function $e\text{-}to\text{-}v_\Gamma$ to deal with substitution into a predicate. The existential and universal projection functions (epr and upr) give a concrete transformer semantics to quantifiers.

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

1. *The lattices of tuples $\{(\mathcal{P}(Val^n), \subseteq) \mid n \in \mathbb{N}\}$.*
2. *The lattices of environments $(\mathcal{P}(Env_\Gamma), \subseteq)$ for every context Γ.*
3. *For every context $\Gamma = (x_0, \ldots, x_{n-1})$ of length n, there are two translation functions for mapping between tuples and environments.*

$$v\text{-}to\text{-}e_\Gamma : \mathcal{P}(Val^n) \to \mathcal{P}(Env_\Gamma)$$
$$v\text{-}to\text{-}e_\Gamma \mathrel{\hat{=}} V \mapsto \{x_i \mapsto v_i \mid \bar{v} \in V\}$$
$$e\text{-}to\text{-}v_\Gamma : \mathcal{P}(Env_\Gamma) \to \mathcal{P}(Val^n)$$
$$e\text{-}to\text{-}v_\Gamma \mathrel{\hat{=}} E \mapsto \{\{i \mapsto v_i \mid \varepsilon(x_i) = v_i\} \mid \varepsilon \in E\}$$

4. *There is a weakening function for mapping environments from a subcontext Γ of Γ' to environments over Γ'.*

$$wk_{\Gamma,\Gamma'} : \mathcal{P}(Env_\Gamma) \to \mathcal{P}(Env_{\Gamma'})$$
$$wk_{\Gamma,\Gamma'} \mathrel{\hat{=}} E \mapsto \{\varepsilon' \mid \varepsilon \in E, \text{ for all } x \text{ in } \Gamma, \varepsilon(x) = \varepsilon'(x)\}$$

5. *For every subcontext Γ of Γ' an existential projection $epr_{\Gamma',\Gamma}$ and a universal projection $upr_{\Gamma',\Gamma}$, in $\mathcal{P}(Env_{\Gamma'}) \to \mathcal{P}(Env_\Gamma)$.*

$$epr_{\Gamma',\Gamma} \mathrel{\hat{=}} E \mapsto \{\varepsilon \mid\ wk_{\Gamma',\Gamma}(\varepsilon) \cap E \neq \emptyset\}$$
$$upr_{\Gamma',\Gamma} \mathrel{\hat{=}} E \mapsto \{\varepsilon \mid\ wk_{\Gamma',\Gamma}(\varepsilon) \subseteq E\}$$

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

$$cint(F) : \mathcal{P}(Val^{ar(F)}) \to \mathcal{P}(Val)$$
$$cint(F) \mathrel{\hat{=}} V \mapsto \{int(F)(\bar{v}) \mid \bar{v} \in V\}$$

7. *The relation $int(P)$ 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 epr_x and upr_x for projections that extract a single variable by mapping environments over the context Γ, x to those over x. Conversely, we write $some_x$ and all_x for projections that eliminate

a single variable by mapping environments over Γ, x to environments over Γ. We also write wk_x for the weakening function from environments over Γ to those over Γ, x.

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

$$[\![\cdot]\!]_{\mathcal{M}} : \mathit{Term}^n_\Gamma \to (\mathcal{P}(\mathit{Env}_\Gamma) \to \mathcal{P}(\mathit{Val}^n))$$

The semantics function is defined inductively below. The semantics of a term t with respect to a set of environments E is given by evaluating t in each environment in E. The semantics of a sequence of terms is a set of sequences of values. Substitution constructs a sequence of values and lifts it to an environment.

$$\begin{aligned}
[\![x : \Gamma]\!] &\mathrel{\hat{=}} E \mapsto \{\varepsilon(x) \mid \varepsilon \in E\} \\
[\![\bar{t} : \Gamma]\!] &\mathrel{\hat{=}} E \mapsto \{\bar{v} \in \mathit{Val}^{len(\bar{t})} \mid \text{ for some } \varepsilon \in E, \\
&\qquad v_i \in [\![t_i : \Gamma]\!](\varepsilon) \text{ for all } 0 \le i < len(\bar{t})\} \\
[\![F(\bar{t}) : \Gamma]\!] &\mathrel{\hat{=}} \mathit{cint}(F) \circ [\![\bar{t} : \Gamma]\!] \\
[\![t[\bar{x} \mapsto \bar{r}] : \Gamma]\!] &\mathrel{\hat{=}} [\![t : \Gamma']\!] \circ \mathit{v\text{-}to\text{-}e}_{\Gamma'} \circ [\![\bar{r} : \Gamma]\!]
\end{aligned}$$

Example 1 below demonstrates how to calculate the semantics of a term in the presence of substitution. Observe that the term does not have to be rewritten before being evaluated.

Example 1. Consider the term $t \mathrel{\hat{=}} x + 2y[y \mapsto 2x]$ derived below.

$$\dfrac{\dfrac{\dfrac{\text{VAR}\ \overline{x : x}}{\text{WEAK}\ \overline{x : x, y}}}{\text{FUN}\ \overline{x + 2y : x, y}} \quad \dfrac{\dfrac{\text{VAR}\ \overline{y : y}}{\text{WEAK}\ \overline{y : x, y}}}{\text{FUN}\ \overline{2y : x}}}{\text{CSUB}\ \overline{x + 2y[x \mapsto x, y \mapsto 2x] : x}}$$

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

$$\begin{aligned}
&[\![x + 2y[x \mapsto x, y \mapsto 2x] : x]\!] \\
&= [\![x + 2y : x, y]\!] \circ \mathit{v\text{-}to\text{-}e}_{x,y} \circ [\![(x, 2x) : x]\!] \\
&= [\![x + 2y : x, y]\!] \circ \mathit{v\text{-}to\text{-}e}_{x,y} \circ \lambda E.\, \{(\varepsilon(x), 2\varepsilon(x)) \mid \varepsilon \in E)\} \\
&= [\![x + 2y : x, y]\!] \circ \lambda E.\, \{\{x \mapsto \varepsilon(x), y \mapsto 2\varepsilon(x)\} \mid \varepsilon \in E\} \\
&= [\![x + 2y : x, y]\!] \circ \lambda E.\, \{\varepsilon(x) + 4\varepsilon(x) \mid \varepsilon \in E\} \\
&= \lambda E.\, \{5\varepsilon(x) \mid \varepsilon \in E\}
\end{aligned}$$

In short, a set of environments is mapped to values obtained by multiplying the value of x by 5. ◁

The collecting semantics of terms is the standard evaluation semantics (also called 'forward interpretation') implemented in program analyzers. The semantics of quantifier-free formulae given next is a 'backward interpretation' and is known [9] but is less standard. The inverse $[\![\bar{t} : \Gamma]\!]^{-1} : \mathcal{P}(\mathit{Val}^{len(\bar{t})}) \to \mathcal{P}(\mathit{Env}_\Gamma)$ is defined as follows.

$$[\![\bar{t} : \Gamma]\!]^{-1} \mathrel{\hat{=}} V \mapsto \{\varepsilon \in \mathit{Env}_\Gamma \mid [\![\bar{t}]\!](\varepsilon) \cap V \neq \emptyset\}$$

The semantics of a formula is given by a function $[\![\cdot]\!]_{\mathcal{M}} : \mathit{Form}_\Gamma \to \mathcal{P}(\mathit{Env}_\Gamma)$ defined inductively below. Boolean operators have their standard set-theoretic interpretation.

$$\begin{aligned}
[\![P(\bar{t}) : \Gamma]\!] &\mathrel{\hat{=}} [\![\bar{t} : \Gamma]\!]^{-1}(\mathit{int}(P)) \qquad [\![\mathsf{true} : \Gamma]\!] \mathrel{\hat{=}} \mathit{Env}_\Gamma \\
[\![\varphi \vee \psi : \Gamma]\!] &\mathrel{\hat{=}} [\![\varphi : \Gamma]\!] \cup [\![\psi : \Gamma]\!] \qquad [\![\mathsf{false} : \Gamma]\!] \mathrel{\hat{=}} \emptyset \\
[\![\varphi \wedge \psi : \Gamma]\!] &\mathrel{\hat{=}} [\![\varphi : \Gamma]\!] \cap [\![\psi : \Gamma]\!] \qquad [\![\neg\varphi : \Gamma]\!] \mathrel{\hat{=}} \neg[\![\varphi : \Gamma]\!]
\end{aligned}$$

Quantifiers are interpreted using projection functions.

$$[\![\exists x.\varphi : \Gamma]\!] \mathrel{\hat{=}} \mathit{some}_x([\![\varphi : \Gamma, x]\!]) \quad [\![\forall x.\varphi : \Gamma]\!] \mathrel{\hat{=}} \mathit{all}_x([\![\varphi : \Gamma, x]\!])$$

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

$$\varphi \mathrel{\hat{=}} \exists x.x + 2y[y \mapsto 2x] = z$$

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

$$\begin{aligned}
&[\![\exists x.x + 2y[x \mapsto x, y \mapsto 2x] = z : z]\!] \\
&= \mathit{some}_x \circ [\![(x + 2y[x \mapsto x, y \mapsto 2x], z)]\!]^{-1}(\mathit{cint}(=)) \\
&= \mathit{some}_x \circ (\lambda E.\, \{(5\varepsilon(x), \varepsilon(z)) \mid \varepsilon \in E\})^{-1}(\mathit{cint}(=)) \\
&= \mathit{some}_x(\{\varepsilon \mid \varepsilon(z) = 5(\varepsilon(x))\}) \\
&= \{\{z \mapsto 5n\} \mid n \in \mathbb{N}\}
\end{aligned}$$

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 $\varphi : \Gamma$ in non-empty context Γ, classical, first-order interpretation $\mathcal{M}$ and environment $\varepsilon \in \mathit{Env}_\Gamma$, $\mathcal{M}, \varepsilon \models \varphi$ exactly if $\varepsilon \in [\![\varphi : \Gamma]\!]_{\mathcal{M}}$.*

We also refer to environments as *structures*. Let Γ be a non-empty context. A structure ε is a *model* of φ if $\varepsilon \in [\![\varphi : \Gamma]\!]$. A formula φ is *unsatisfiable in* $\mathcal{M}$ if $[\![\varphi : \Gamma]\!]_{\mathcal{M}}$ is the empty set and is *satisfiable in* $\mathcal{M}$ 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 $[\![\varphi : []]\!]_{\mathcal{M}}$ is $\{\emptyset\}$, we say that φ is *true in* $\mathcal{M}$, and otherwise, φ is *false in* $\mathcal{M}$.

4. Concrete Reasoning

The basic operations in logical reasoning can be viewed as giving a dynamic interpretation to an implication $\varphi \Rightarrow \psi$. Deduction is the process of deriving ψ from φ. Abduction is the process of deriving φ from ψ. In classical logic, these processes have contrapositive formulations: we can start with $\neg\psi$ and attempt to deduce $\neg\varphi$, a process we call *contradeduction*, or start with $\neg\varphi$ and attempt to abduce $\neg\psi$, a process we call *contraabduction*. In this section, we model these processes using transformers and characterize properties of formulae as fixed points of these transformers. As with fixed point characterisations of program correctness, these fixed points are not meant to be computed but will be used to design fixed point approximation algorithms.

The set of formulae that can be derived from a set of formulae Φ using a set of rules R forms the *deductive closure* of Φ 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 Form_Γ is a function $T_\varphi : \mathcal{P}(\mathit{Env}_\Gamma) \to \mathcal{P}(\mathit{Env}_\Gamma)$. Structure transformers encode reasoning about the models and countermodels of a formula. The *deduction transformer* ded_φ which encodes reasoning about models of φ. In

the definition below, we assume that the set-theoretic operations are lifted pointwise to functions.

$$ded_{P(\bar{t})}(X) \mathrel{\hat{=}} X \cap [\![P(\bar{t}) : \Gamma]\!] \qquad ded_{\varphi\wedge\psi} \mathrel{\hat{=}} ded_\varphi \cap ded_\psi$$
$$ded_{\neg\varphi} \mathrel{\hat{=}} \neg[\![\varphi : \Gamma]\!] \qquad ded_{\varphi\vee\psi} \mathrel{\hat{=}} ded_\varphi \cup ded_\psi$$

The use of negation in defining $ded_{\neg\varphi}$ 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 *contradeduction* transformer $cded_\varphi$ encodes contrapositive reasoning and manipulates countermodels of φ. If ded_φ is used for satisfiability checking, $cded_\varphi$ can be used for validity checking.

$$cded_{P(\bar{t})}(X) \mathrel{\hat{=}} X \cap \neg[\![P(\bar{t}) : \Gamma]\!] \qquad cded_{\varphi\wedge\psi} \mathrel{\hat{=}} cded_\varphi \cup cded_\psi$$
$$cded_{\neg\varphi} \mathrel{\hat{=}} \neg cded_\varphi \qquad cded_{\varphi\vee\psi} \mathrel{\hat{=}} cded_\varphi \cap cded_\psi$$

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 φ, we know that φ is valid with respect to a set of structures.

$$abd_{P(\bar{t})}(X) \mathrel{\hat{=}} X \cup \neg[\![P(\bar{t}) : \Gamma]\!] \qquad abd_{\varphi\wedge\psi} \mathrel{\hat{=}} abd_\varphi \cup abd_\psi$$
$$abd_{\neg\varphi} \mathrel{\hat{=}} cabd_\varphi \qquad abd_{\varphi\vee\psi} \mathrel{\hat{=}} abd_\varphi \cap abd_\psi$$

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.

$$cabd_{P(\bar{t})}(X) \mathrel{\hat{=}} X \cup [\![P(\bar{t}) : \Gamma]\!] \qquad cabd_{\varphi\wedge\psi} \mathrel{\hat{=}} cabd_\varphi \cap cabd_\psi$$
$$cabd_{\neg\varphi} \mathrel{\hat{=}} abd_\varphi \qquad cabd_{\varphi\vee\psi} \mathrel{\hat{=}} cabd_\varphi \cup cabd_\psi$$

In addition to deduction and abduction, quantifier elimination is fundamental to logical reasoning. The transformers $some_x$ and all_x 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 ($wk_x \circ some_x$) as an upper closure, and of universal quantification ($wk_x \circ all_x$) 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.*

$$ded_\varphi(X) = (X \cap [\![\varphi : \Gamma]\!]) \qquad cded_\varphi(X) = (X \cap \neg[\![\varphi : \Gamma]\!])$$
$$cabd_\varphi(X) = (X \cup [\![\varphi : \Gamma]\!]) \qquad abd_\varphi(X) = (X \cup \neg[\![\varphi : \Gamma]\!])$$

2. *The transformers ded_φ and $cded_\varphi$ are lower closures.*
3. *The transformers $cabd_\varphi$ and abd_φ are upper closures.*
4. *The composition $wk_x \circ some_x$ is an upper closure and $wk_x \circ all_x$ is a lower closure.*
5. *The pairs of transformers $(ded_\varphi, abd_\varphi)$, $(cded_\varphi, cabd_\varphi)$ and $(all_x, some_x)$ are De Morgan duals.*
6. *The pairs of transformers $(ded_\varphi, abd_\varphi)$ $(cded_\varphi, cabd_\varphi)$ form a Galois connection on $(\mathcal{P}(Env), \subseteq)$.*

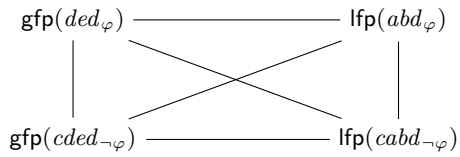


Figure 1. The vertices represent deduction or abduction procedures for checking satisfiability of φ or validity of $\neg\varphi$. The edges represent combinations of deduction and abduction procedures.

7. *The quantification transformers are related to weakening by the following Galois connections.*

$$(\mathcal{P}(Env_\Gamma), \subseteq) \underset{wk_x}{\overset{all_x}{\leftrightarrows}} (\mathcal{P}(Env_{\Gamma,x}), \subseteq), \text{ and}$$
$$(\mathcal{P}(Env_{\Gamma,x}), \subseteq) \underset{some_x}{\overset{wk_x}{\leftrightarrows}} (\mathcal{P}(Env_\Gamma), \subseteq)$$

The Galois connections are useful for deriving equivalent formulations of properties of a formula. For example, the models of φ are $ded_\varphi(Env)$. A formula is unsatisfiable exactly if $ded_\varphi(Env) \subseteq \emptyset$, which by the Galois connection, is equivalent to $Env \subseteq abd_\varphi(\emptyset)$. In words, we can determine if a formula is unsatisfiable by trying to deduce false from φ or trying to abduce φ from false. Similarly, a formula is valid exactly if $cded_\varphi(Env) \subseteq \emptyset$, or equivalently $Env \subseteq cabd_\varphi(\emptyset)$.

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 φ. We initially know nothing about φ, which we can express as $\varphi \Rightarrow \mathsf{true}$. A single step of a reasoning algorithm may indicate that $\varphi \Rightarrow \psi_1$. After k steps, the algorithm may deduce that $\varphi \wedge \cdots \psi_1 \wedge \psi_{k-1} \Rightarrow \psi_k$. What we know about the models of φ can be represented by the sequence $[\![\mathsf{t}]\!]$, $[\![\psi_1]\!]$, $\ldots$, $[\![\psi_1 \wedge \cdots \psi_k]\!]$. 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 φ.

Fixed points of structure transformers represent brute force algorithms. The greatest fixed point $\mathsf{gfp}(ded_\varphi)$ represents the semantics of a solver that initially assumes that every structure is a model of φ and then eliminates countermodels of φ. The fixed point $\mathsf{gfp}(cded_\varphi)$ represents a procedure that progresses by eliminating countermodels of φ. The fixed point $\mathsf{lfp}(abd_\varphi)$ represents a procedure that initially assumes φ has no countermodels and progresses by finding countermodels of φ, while $\mathsf{lfp}(cabd_\varphi)$ does the same for models of φ. 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 φ is unsatisfiable.*
2. *The greatest fixed point $\mathsf{gfp}(ded_\varphi)$ is empty.*
3. *The least fixed point $\mathsf{lfp}(abd_\varphi)$ has all structures.*
4. *The formula $\neg\varphi$ is valid.*
5. *The least fixed point $\mathsf{lfp}(cabd_{\neg\varphi})$ has all structures.*
6. *The greatest fixed point $\mathsf{gfp}(cded_{\neg\varphi})$ 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 $\mathcal{P}(Env) \times \mathcal{P}(Env)$ with the lattice order shown alongside. For intuition about these combinations, consider a greatest fixed point iteration with the transformer da_φ, which combines deduction and abduction. The first element is $(Env, \emptyset)$, representing that every structure is a potential model and no structure is a potential countermodel. A single application of da_φ yields $(ded_\varphi(Env), abd_\varphi(\emptyset))$, which is a fixed point. If φ is unsatisfiable, this fixed point is $(\emptyset, Env)$, 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.

$$
\begin{array}{rll}
dcd_\varphi(X,Y) \mathrel{\hat{=}} (ded_\varphi(X \cap Y), cded_{\neg\varphi}(X \cap Y)) & \subseteq \times \subseteq \\
da_\varphi(X,Y) \mathrel{\hat{=}} (ded_\varphi(X \cap \neg Y), abd_\varphi(\neg X \cup Y)) & \subseteq \times \supseteq \\
dca_\varphi(X,Y) \mathrel{\hat{=}} (ded_\varphi(X \cup Y), cabd_{\neg\varphi}(\neg X \cup Y)) & \subseteq \times \supseteq \\
cda_\varphi(X,Y) \mathrel{\hat{=}} (cded_{\neg\varphi}(X \cap \neg Y), abd_\varphi(\neg X \cup Y)) & \subseteq \times \supseteq \\
cdca_\varphi(X,Y) \mathrel{\hat{=}} (cded_{\neg\varphi}(X \cap \neg Y), cabd_{\neg\varphi}(\neg X \cup Y)) & \subseteq \times \supseteq \\
aca_\varphi(X,Y) \mathrel{\hat{=}} (abd_\varphi(X \cup Y), cabd_{\neg\varphi}(X \cup Y)) & \supseteq \times \supseteq
\end{array}
$$

The dcd_φ 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_φ) and abduction with universal quantification (aa_φ). These are only two of many possible combinations.

$$
\begin{array}{rll}
ds_\varphi(X,Y) \mathrel{\hat{=}} (ded_\varphi(X \cap Y), wk_x \circ some_x(X \cap Y)) & \subseteq \times \subseteq \\
aa_\varphi(X,Y) \mathrel{\hat{=}} (abd_\varphi(X \cup Y), wk_x \circ all_x(X \cup Y)) & \supseteq \times \supseteq
\end{array}
$$

The application of these transformers to satisfiability is below.

Theorem 5. *The following statements are equivalent.*

1. *The formula φ is unsatisfiable.*
2. *The fixed point $\mathsf{gfp}(da_\varphi)$ is $(\emptyset, Env)$.*
3. *The fixed point $\mathsf{gfp}(dcd_\varphi)$ is $(\emptyset, \emptyset)$.*
4. *The fixed point $\mathsf{gfp}(dca_\varphi)$ is $(\emptyset, Env)$.*
5. *The fixed point $\mathsf{gfp}(cda_\varphi)$ is $(\emptyset, Env)$.*
6. *The fixed point $\mathsf{gfp}(cdca_\varphi)$ is $(\emptyset, Env)$.*
7. *The fixed point $\mathsf{gfp}(aca_\varphi)$ is (Env, Env).*
8. *The fixed point $\mathsf{gfp}(ds_\varphi)$ is $(\emptyset, \emptyset)$.*
9. *The fixed point $\mathsf{gfp}(aa_\varphi)$ is (Env, Env).*

4.3 Connection to Programs

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

$$rel([\mathsf{b}]) \mathrel{\hat{=}} \{(\varepsilon, \varepsilon) \mid \varepsilon \in [\![b]\!]\}$$

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.

$$
\begin{array}{rlrl}
post_{[\mathsf{b}]} &\mathrel{\hat{=}} X \mapsto X \cap [\![\mathsf{b}]\!] & \tilde{post}_{[\mathsf{b}]} &\mathrel{\hat{=}} \neg \circ post_{[\mathsf{b}]} \circ \neg \\
pre_{[\mathsf{b}]} &\mathrel{\hat{=}} X \mapsto X \cap [\![\mathsf{b}]\!] & \tilde{pre}_{[\mathsf{b}]} &\mathrel{\hat{=}} \neg \circ pre_{[\mathsf{b}]} \circ \neg
\end{array}
$$

Theorem 6. *For a quantifier-free test $[\varphi]$ we have that $ded_\varphi = post_{[\varphi]}$ and $abd_\varphi = \tilde{pre}_{[\varphi]}$.*

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 lattices 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 $\alpha : C \to A$ and a concretisation function $\gamma : A \to C$. 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, α, γ, A) be a Galois connection between two complete lattices and $F : C \to C$ and $aF : A \to A$ be transformers satisfying $\alpha \circ F \sqsubseteq aF \circ \alpha$. Then, $\alpha(\mathsf{lfp}(F)) \sqsubseteq \mathsf{lfp}(aF)$ and $\alpha(\mathsf{gfp}(F)) \sqsubseteq \mathsf{gfp}(aF)$.*

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 $aF : A \to A$ is a *sound overapproximation* of a concrete transformer $F : \mathcal{P}(S) \to \mathcal{P}(S)$ 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 *α-complete for F at c* if it satisfies $\alpha(F(c)) = aF(\alpha(c))$ and is *α-complete* if $\alpha \circ F = aF \circ \alpha$. We say that f is *α-fixed point complete* if $\alpha(\mathsf{lfp}(F)) = \mathsf{lfp}(aF)$. An abstract transformer aF is *γ-complete for F at a* if $\gamma(aF(a)) = F(\gamma(a))$ and is *γ-complete* if $\gamma \circ aF = F \circ \gamma$.

5.1 Abstract Interpretation of Formulae

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

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

1. *Abstract value lattices $(aVal_n, \sqsubseteq, \sqcap, \sqcup)$ for each $n \in \mathbb{N}$.*
2. *Abstract environment lattices $(aEnv_\Gamma, \sqsubseteq, \sqcap, \sqcup)$ for each Γ.*

3. *Abstract translation functions* $av\text{-}to\text{-}e_\Gamma : aVal_n \to aEnv_\Gamma$ *and* $ae\text{-}to\text{-}v_\Gamma : aEnv_\Gamma \to aVal_n$ *whenever* $len(\Gamma) = n$.
4. *An abstract weakening function* $awk_{\Gamma,\Gamma'} : aEnv_\Gamma \to aEnv_{\Gamma'}$ *for each context* Γ' *and subcontext* Γ.
5. *Projection functions* $aepr_{\Gamma',\Gamma}, aupr_{\Gamma',\Gamma} : aEnv_{\Gamma'} \to aEnv_\Gamma$, *for every subcontext* Γ *of* Γ'.
6. *An abstract interpretation* $aint(F) : aVal_{ar(F)} \to aVal_1$ *and its inverse* $ainv(F) : aVal_1 \to aVal_{ar(F)}$ *for every function symbol* $F \in Sig$.
7. *An abstract interpretation* $aint(P) \in aVal_{ar(P)}$ *of every predicate symbol* $P \in Sig$.

Every abstract lattice above is related to the corresponding concrete lattice by a Galois connection (C, α, γ, A). *We extend the convention for collecting structures and write* $asome_x$ *and* $aall_x$ *for elimination of a single variable.*

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

$$[\![\cdot]\!]_\mathcal{A} : Term^n_\Gamma \to aEnv_\Gamma \to aVal_n \quad [\![\cdot]\!]_\mathcal{A} : Form_\Gamma \to aEnv_\Gamma$$

We obtain these by replacing concrete transformers and semantics by their abstract counterparts in the definition $[\![]\!]_\mathcal{C}$ 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 φ *overapproximates the concrete semantics* if $[\![\varphi : \Gamma]\!]_\mathcal{C} \subseteq \gamma([\![\varphi : \Gamma]\!]_\mathcal{A})$. 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 Sig-structure* $\mathcal{A}$ *soundly overapproximates a collecting Sig-structure* $\mathcal{C}$ *if every lattice in* $\mathcal{A}$ *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 is more detail shortly.

Theorem 10. *If* $\mathcal{A}$ *is a sound overapproximation of* $\mathcal{C}$ *and* φ *is negation-free* $[\![\varphi : \Gamma]\!]_\mathcal{C} \subseteq \gamma([\![\varphi : \Gamma]\!]_\mathcal{A})$.

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 $(Intv, \sqsubseteq, \sqcap, \sqcup)$ is defined over the set $Intv \;\hat{=}\; \{[a, b] \mid a \leq b, a, b \in \mathbb{Z} \cup \{-\infty, \infty\}\}$, with $\bot$ representing the empty interval and $\top$ being $[-\infty, \infty]$. The interval structure $\mathcal{I}$ contains the items below.

1. The value lattices $aVal_n$ are product lattices $Intv^n$.
2. The interval environments $aEnv_\Gamma$ are $var(\Gamma) \to Intv$.
3. The weakening function extends an interval with the new variable going to $\top$.

$$awk_{\Gamma,\Gamma'} \;\hat{=}\; \varepsilon \mapsto \varepsilon \cup \left\{y \mapsto \top \mid y \in var(\Gamma) \setminus var(\Gamma')\right\}$$

4. The projection functions are given below.

$$aepr_{\Gamma,\Gamma'} \;\hat{=}\; \varepsilon \mapsto \left\{x \mapsto \varepsilon(x) \mid x \in var(\Gamma')\right\}$$

$$aupr_{\Gamma,\Gamma'} \;\hat{=}\; \varepsilon \mapsto \begin{cases} \{x \mapsto \varepsilon(x) \mid x \in var(\Gamma')\} \text{ if} \\ \quad \varepsilon(y) = \top \text{ for all } y \in var(\Gamma) \setminus var(\Gamma') \\ \bot \text{ otherwise} \end{cases}$$

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 $[\![\varphi]\!]_{Intv}$ of the formula $\varphi \;\hat{=}\; x \geq 5 \wedge x \leq 10 \wedge y = 2$ using the interval domain.

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

The domain also supports abstract quantification.

$$[\![\exists y.\varphi]\!] = asome_y([\![\varphi]\!]) = \{x \mapsto [5, 10]\}$$
$$[\![\forall y.\varphi]\!] = aall_y([\![\varphi]\!]) = \bot$$

Now consider $\psi \;\hat{=}\; \varphi \wedge y = 3$ whose semantics is $[\![\psi]\!]_{Intv} = \{x \mapsto [5, 10], y \mapsto \bot\}$. We have that

$$asome_y([\![\psi]\!]_{Intv}) = \{x \mapsto [5, 10]\}, \text{ but}$$
$$\alpha(some_y(\gamma([\![\psi]\!]_{Intv}))) = \alpha(some_y(\emptyset)) = \bot$$

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 $Par \;\hat{=}\; (\{\bot, \mathsf{E}, \mathsf{O}, \top\}, \sqsubseteq)$, which represents even and odd numbers. Let $[\![x = k]\!]_{Par}$ be E if k is even and be O otherwise. The complement $\sim$ in this lattice *cannot be used* to approximate negation. To see why, consider evaluating $\neg(x = 2)$ as $\sim[\![x = 2]\!]_{Par}$, which is $\sim\mathsf{E} = \mathsf{O}$ but $\gamma(\mathsf{O})$ 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 $aint(\neg P)$ 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 philosophical 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 $A \sqsubseteq B$ to an element I satisfying $A \sqsubseteq I$ and $I \sqsubseteq B$. 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 $A \sqcap B \sqsubseteq \bot$ to an I satisfying $A \sqsubseteq I$ and $I \sqcap B \sqsubseteq \bot$. 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 an abstract Sig structure with the following operations.*

1. *Abstract transformers $aded_\varphi$, $acded_\varphi$, $aabd_\varphi$ and $acabd_\varphi$ for deduction and abduction.*
2. *A unary, extensive function $ext_\uparrow : L \to L$ called upwards extrapolation and its dual $ext_\downarrow : L \to L$ called downwards extrapolation.*
3. *A partial, interpolation function $itp : L \times L \to L$, satisfying $x \sqsubseteq itp(x,y) \sqsubseteq y$ whenever $x \sqsubseteq y$. The element $itp(x,y)$ is called an interpolant.*
4. *A partial, separation function $sep : L \times L \to L$, satisfying $x \sqsubseteq sep(x,y)$ and $sep(x,y) \sqcap y \sqsubseteq \bot$ whenever $x \sqcap y \sqsubseteq \bot$. The element $sep(x,y)$ 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 $\mathsf{gfp}(aded_\varphi)$ with $\varphi \mathrel{\hat{=}} x \geq 5 \wedge x \leq 10 \wedge y = 2x$ and $E_0 = \top$.

$$\begin{aligned} E_1 &= aded_\varphi(E_0) \\ &= [\![x \geq 5]\!](\top) \sqcap [\![x \leq 10]\!](\top) \sqcap [\![y = 2x]\!](\top) \\ &= \{x \in [5,10], y \mapsto [-\infty, \infty]\} \\ E_2 &= aded_\varphi(E_1) \\ &= [\![x \geq 5]\!](E_1) \sqcap [\![x \leq 10]\!](E_1) \sqcap [\![y = 2x]\!](E_1) \\ &= \{x \in [5,10], y \mapsto [10,20]\} = \mathsf{gfp}(aded_\varphi) \end{aligned}$$

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 $\gamma(\mathsf{gfp}(aded_\varphi)) = \emptyset$ in $\mathcal{O}$ then φ is unsatisfiable.*
2. *If $\gamma(\mathsf{lfp}(aabd_\varphi)) = Env_\Gamma$ in $\mathcal{U}$ then φ 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 settings 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 φ such that $S = [\![\varphi]\!]$. These notions lead to a Galois connection.

$$th : \mathcal{P}(Env_\Gamma) \to \mathcal{P}(Form_\Gamma) \qquad st : \mathcal{P}(Form_\Gamma) \to \mathcal{P}(Env_\Gamma)$$
$$th \mathrel{\hat{=}} S \mapsto \{\varphi \mid S \subseteq [\![\varphi]\!]\} \qquad st \mathrel{\hat{=}} \Phi \mapsto \bigcap_{\varphi \in \Phi} [\![\varphi]\!]$$

Theorem 13. *There is a Galois connection $(\mathcal{P}(Env), \subseteq) \underset{th}{\overset{st}{\leftrightarrows}} (\mathcal{P}(Form), \supseteq)$ 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 primacy to formulae and view structures as abstractions. By the Galois connection, a *complete theory* Φ is one satisfying $th(st(\Phi))$ and a set of structures S is *definable* if $st(th(S)) = S$.

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 φ_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 ψ is *derived from* a set of formulae Φ using a proof system, written $\Phi \vdash \psi$, if ψ occurs in Φ or if ψ 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 Φ. The *deductive closure* of a set of axioms Φ with respect to a proof system $\vdash$ is the set of all formulae that can be derived from Φ with $\vdash$. A proof system is *deductively sound* if every ψ derived from Φ is in the theory of Φ. A proof system is *deductively complete* if every ψ in the theory of Φ can be derived from Φ. A *refutation* is a derivation of false.

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

$$aded_\varphi \mathrel{\hat{=}} \Phi \mapsto \{\psi \mid \text{ there exist } \varphi_i \in \Phi \cup \{\varphi\} \\ (\varphi_0, \ldots, \varphi_{n-1}, \psi) \text{ is in a proof rule } \}$$

The fixed point $\mathsf{gfp}(aded_\varphi)$ (with respect to the superset order) contains all formulae derivable from φ. Since multiple applications of proof rules are required to derive a conclusion, $aded_\varphi$ 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 $ded_\varphi \circ st \supseteq aded_\varphi \circ th$ and is deductively complete exactly if $th \circ ded_\varphi \circ st = \mathsf{gfp}(aded_\varphi)$.*

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 lattices support logical notions such as resolution, the pure literal rule, and subsumption. We demonstrate that pre- and in-processing techniques used in SAT solvers can be understood as abstract quantification procedures. Thus, these procedures have a semantic justification, 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 formulae to logics defined over posets.

A subset Q of a poset 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. The *downset lattice* over A, written $(\mathcal{D}(A), \subseteq, \cap, \cup)$, 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 $\mathcal{P}(S)$. The dual notion to downset is *up-sets*. The smallest up-set containing Q is $Q\!\uparrow$, and $(\mathcal{U}(A), \supseteq)$ is the *up-set* lattice with intersection as join and union as meet. Let $min(Q)$ and $max(Q)$ denote the minimal and maximal elements 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 $\gamma(a \sqcup b) = \gamma(a) \cup \gamma(b)$. Downset completion is an operation that enriches 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 underapproximating abstractions.

$$\gamma_{\mathcal{D}(A)} : \mathcal{D}(A) \to \mathcal{P}(S) \quad \gamma_{\mathcal{D}(A)}(Q) \mathrel{\hat{=}} \bigcup \{\gamma(x) \mid x \in Q\}$$
$$\alpha_{\mathcal{D}(A)} : \mathcal{P}(S) \to \mathcal{D}(A) \quad \alpha_{\mathcal{D}(A)}(P) \mathrel{\hat{=}} \{x \mid \gamma(x) \subseteq P\}$$
$$\gamma_{\mathcal{U}(A)} : \mathcal{U}(A) \to \mathcal{P}(S) \quad \gamma_{\mathcal{U}(A)}(Q) \mathrel{\hat{=}} \bigcap \{\gamma(x) \mid x \in Q\}$$
$$\alpha_{\mathcal{U}(A)} : \mathcal{P}(S) \to \mathcal{U}(A) \quad \alpha_{\mathcal{U}(A)}(P) \mathrel{\hat{=}} \{x \mid \gamma(x) \supseteq P\}$$

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

$$gClause \mathrel{\hat{=}} (\mathcal{D}(gLit), \subseteq) \qquad gCube \mathrel{\hat{=}} (\mathcal{U}(gLit), \supseteq)$$
$$gCNF \mathrel{\hat{=}} (\mathcal{U}(gClause), \supseteq) \qquad gDNF \mathrel{\hat{=}} (\mathcal{D}(gCube), \subseteq)$$

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 formulae, the concretisation function for the poset A is the function $st : gLit \to \mathcal{P}(Env_\Gamma)$ from formulae to models. If a is a generalized literal, we write $\sim a$ for a literal that satisfies $\gamma(a) \cup \gamma(\sim a) = \top$.

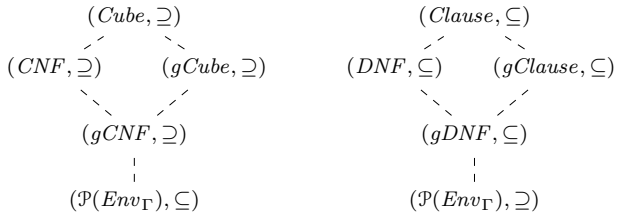


Figure 2. Refinement order between the generalised CNF domains.

$gLit$	$gCube$
$(\{p, \neg p \mid p \in Prop\}, =)$	Partial assignments
$(\{x = y, x \neq y \mid x \in Vars\}, =)$	Equality graphs
$(\{x <= y, x < y \mid x \in Vars\}, =)$	Difference graphs
$(\{x \leq k, x \geq k \mid k \in \mathbb{N}\}, \Rightarrow)$	Interval cubes
$(\{m \vee n \mid m, n \in Lit\}, =)$	Binary Implication Graphs

Table 1. Domains viewed as generalised clauses, cubes and CNF

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 overapproximations of generalised cubes and CNF formulae, which are in turn overapproximations of generalised CNF, which are overapproximations of environments. On the other hand, the dual constructions give us underapproximating domains of clauses, generalised clauses, DNF and generalised DNF respectively.

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

Partial Assignments Consider a set of Boolean variables $Prop$, a set of literals Lit and the identity relation. The lattice $gCube$ is isomorphic 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 $ELit = \{x = y, x \neq y\}$ 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 $trans : \mathcal{U}(ELit) \to \mathcal{U}(ELit)$.

$$trans \mathrel{\hat{=}} G \mapsto G \cup \{x = y \mid x = z, z = y \in G\}$$

Recall that a reduction operator is an abstract transformer ρ satisfying 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

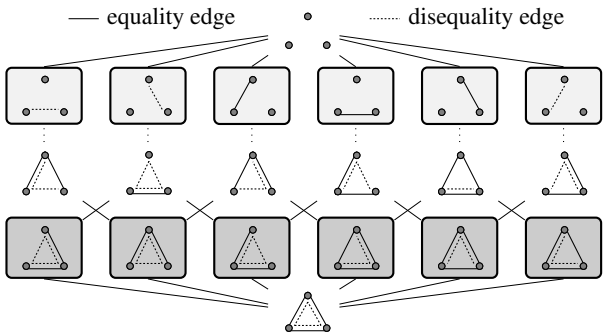


Figure 3. Equality graphs over three variables. The shaded elements at the top are literals and the shaded elements at the bottom all concretise to the empty set.

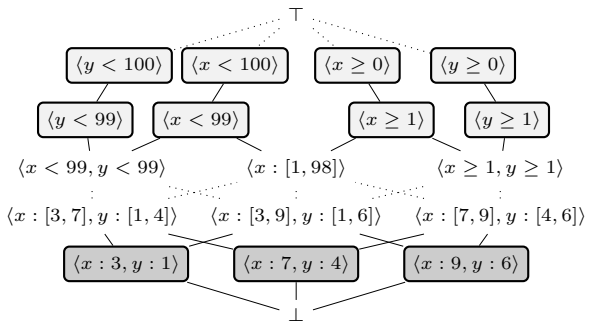


Figure 4. The lattice of interval environments over two variables. The shaded elements at the top represent the poset of literals while elements at the bottom represent singleton values.

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 $(BLit = \{x \leq k, x \geq k \mid k \in \mathbb{Z}\}, \Rightarrow)$. 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.

$$([0,1] \sqcup [6,7]) \sqcap [3,4] = [3,4]$$
$$\neq ([0,1] \sqcap [3,4]) \sqcup ([6,7] \sqcap [3,4]) = \bot$$

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.

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

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 generalised literals contains clauses 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 $gCNF$ and CNF, consider $BLit$. The set $\{x \geq 2, x \leq 5, x \leq 7\}$ represents a clause but not a generalized clause, while $\{x \geq 2, x \leq 5\}\downarrow$ is a generalised 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 $\{\{p\}\}$ are logically equivalent but we can only show $\{\{p\}\} \subseteq \{\{p\}, \{p, q\}\}$ using the lattice order. From the abstract interpretation 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 a 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.

$$gunit : gClause \to (gCube \to gCube)$$
$$gunit_\theta \mathrel{\hat{=}} \pi \mapsto \pi \cup \{a\}, \text{ where } a \in \theta, \gamma(\pi) \cap \gamma(a) \neq \emptyset, \text{ and}$$
$$\text{for all } b \in \theta \setminus \{a\}, \gamma(\pi) \cap \gamma(b) = \emptyset$$

We write $\mathsf{gfp}_x(f)$ for the function that maps elements a to fixed points $\mathsf{gfp}(f(a, x))$ of a function $f(y, x)$. The main observation of [18] was that Boolean Constraint Propagation (BCP) is a fixed point $bcp(\varphi, x) \mathrel{\hat{=}} \mathsf{gfp}_x(\bigsqcap_{\theta \in \varphi} gunit_\theta)$ defined pointwise 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 preprocessing technique in SAT solvers. The technique chooses literal a, computes $bcp(a, \varphi)$ and if the result is $\bot$, adds the singleton clause $\{b\}$ to φ, where b satisfies that $\gamma(a) \cup \gamma(b) = \top$. 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 simplification of formulae. The simplest form of sound variable elimination is to drop clauses from a CNF formula in which the target variable occurs. This idea lifts directly to generalized CNF domains if we drop constraints based on the literals they contain.

$$drop : gLit \times gCNF \to gCNF$$
$$drop_p(\varphi) \mathrel{\hat{=}} \{C \mid \{p, \sim p\} \cap C = \emptyset, C \in \varphi\}$$

In general, $drop_p$ is an overapproximation of the deduction transformer. If p is a Boolean variable, $drop_p$ also overapproximates existential quantification with respect to p.

Resolution The resolution principle asserts that if $C \vee p$ and $\neg p \vee D$ are both satisfiable, so is $C \vee D$. The variable p is the *pivot* and $C \vee D$ is the *resolvent*. Generalised CNF domains support a generalization of resolution $res : gLit \times gCNF \to gCNF$.

$$res(p, \varphi) \hat{=} \{C \cup D \mid C \cup \{p\}, D \cup \{q\} \in \varphi, \gamma(p) \cap \gamma(q) = \emptyset\}$$

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

$$\varphi \hat{=} \{\{x \geq 10, x \leq 5\} \rfloor, \{x \geq 7, x \leq 13\} \rfloor\}$$

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 $res(x \geq 7, \varphi) = \{\{x \geq 10, x \leq 13\}\}$. Though arithmetic techniques methods can deduce the same information, generalised resolution is simple and suffices in this case. $\lhd$

The Pure Literal Rule Clause dropping is a sound but incomplete simplification technique because a formula φ 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 $gLit$ 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 $\gamma(a) \cap \gamma(b) = \emptyset$ is also in A. We assume that $gLit$ 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 $\gamma(a) \cap \gamma(b) = \emptyset$ 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 $\mathcal{P}(\{+, -\})$, and a lattice of polarity maps $P \to \mathcal{P}(\{+, -\})$.

Polarity analysis of a generalized CNF formula φ computes a polarity map ρ. such that $\rho(\ell)$ is $\{+\}$ or $\{-\}$ if the generalized literal ℓ only occurs positively or only negatively in φ, and is $\emptyset$ if ℓ does not occur in φ, and is $\{+, -\}$ if ℓ occurs both positively and negatively in φ. The polarity of a literal ℓ is $\{+\}$ if a is in P, and is $\{-\}$ if ℓ is in N. The polarity map for a literal ℓ sends ℓ 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 $drop_\ell$ to a formula φ only if the polarity map for φ does not send the literal ℓ 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 $(wk_p \circ some_p)$.

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.

While the focus of this paper has been theoretical, our goal is to contribute to the practical state of the art. The original abstract interpretation framework provided a simple recipe for constructing static analyzers. Abstract satisfaction plays a similar role and provides a foundation for the development of programmable, lattice-based SMT solvers.

There are three different axes for future work. One is to apply abstract interpretation to the implementation of SMT solvers by constructing sound but incomplete solvers and abstract quantification procedures from existing abstract domains. The second axis is to lift techniques in SMT solvers to improve the precision and efficiency of program analysis. The classic DPLL, DPLL(T), CDCL and Stålmarck's method have each been lifted to a single static analysis problem, but more applications and evaluation are required to understand their strengths in a static analysis context. Preprocessing, subsumption, and sparsity techniques have all been integral to improving the performance of solvers, and the connections in this paper indicate that such techniques should lift to program analysis as well. The final axis is to investigate new implementations of abstract domains with interfaces rich enough to support SMT solving, static analysis, implication graph construction, and domain and theory combinations. We look forward to these developments.

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

[1] N. Bjørner, B. Duterte, and L. de Moura. Accelerating lemma learning using joins – DPLL(⊔). In *LPAR*, 2008.

[2] M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening. An abstract interpretation of DPLL(T). In *VMCAI*, 2012.

[3] M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening. Interpolation-based verification of floating-point programs with abstract CDCL. In *SAS*, 2013.

[4] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *TACAS*, pages 358–372. Springer, 2007.

[5] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, Sept. 1994.

[6] S. Cotton. Natural domain SMT: A preliminary assessment. In *FORMATS*, pages 77–91, 2010.

[7] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., 1981.

[8] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[9] P. Cousot. Abstract interpretation. MIT course 16.399, 2005.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM Press, 1977.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM Press, 1979.

[12] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

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

[14] P. Cousot, R. Cousot, and L. Mauborgne. Theories, solvers and static analysis by abstract interpretation. *JACM*, 59(6):31:1–31:56, Jan. 2013.

[15] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, Cambridge, UK, 1990.

[16] M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7:201–215, July 1960.

[17] L. M. de Moura and D. Jovanovic. A model-constructing satisfiability calculus. In *VMCAI*, pages 1–12, 2013.

[18] V. D'Silva, L. Haller, and D. Kroening. Satisfiability solvers are static analysers. In *SAS*, pages 317–333. Springer, 2012.

[19] V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. In *POPL*, pages 143–154, New York, NY, USA, 2013. ACM Press.

[20] V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric bounds analysis with conflict-driven learning. In *TACAS*, pages 48–63. Springer, 2012.

[21] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, Munich, Germany, 2005. Springer.

[22] J. W. Freeman. Failed literals in the Davis-Putnam procedure for SAT. Technical report, Rutgers University, 1993.

[23] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 175–188, 2004.

[24] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *JACM*, 47(2):361–416, 2000.

[25] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386. ACM Press, 2006.

[26] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, pages 131–140, 2012.

[27] W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL*, pages 71–82, 2010.

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

[29] M. J. H. Heule, M. Järvisalo, and A. Biere. Efficient CNF simplification based on binary implication graphs. In *SAT*, pages 201–215, 2011.

[30] D. Kroening, J. Ouaknine, S. A. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *CAV*, pages 308–320, July 2004.

[31] D. Kroening and G. Weissenbacher. An interpolating decision procedure for transitive relations with uninterpreted functions. In *HVC*, pages 150–168, 2011.

[32] W. Lawvere. Adjointness in foundations. *Dialectica*, 23:281–296, 1969.

[33] K. R. M. Leino and F. Logozzo. Using widenings to infer loop invariants inside an SMT solver, or: A theorem prover as abstract domain. In *Workshop on Invariant Generation*, pages 70–84. RISC Report 07-07, 2007.

[34] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, pages 1–13, 2003.

[35] K. L. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to richer logics. In *CAV*, pages 462–476, 2009.

[36] O. Meir and O. Strichman. Yet another decision procedure for equality logic. In *CAV*, pages 307–320, 2005.

[37] D. Monniaux and L. Gonnord. Using bounded model checking to focus fixpoint iterations. In *SAS*, pages 369–385, 2011.

[38] I. Németi. Algebraization of quantifier logics, an introductory overview. *Studia Logica: An International Journal for Symbolic Logic*, 50(3/4):485–569, 1991.

[39] 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.

[40] A. M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press, 2000.

[41] T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.

[42] P. Smith. The Galois connection of syntax and semantics. Technical report, Cambridge University, 2010.

[43] P. J. Stuckey. There are no CNF problems. SAT, pages 19–21, 2013.

[44] A. Thakur and T. Reps. A generalization of Stålmarck's method. In *SAS*. Springer, 2012.

[45] A. V. Thakur and T. W. Reps. A method for symbolic computation of abstract operations. In *CAV*, 2012.

[46] A. Tiwari and S. Gulwani. Logical interpretation: Static program analysis using theorem proving. In *CADE*, pages 147–166, 2007.

[47] O. Tveretina. DPLL-based procedure for equality logic with uninterpreted functions. In *IJCAR Doctoral Programme*, volume 106 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.