An Abstract Interpretation of DPLL(T)[⋆]

Martin Brain[1], Vijay D'Silva[1], Leopold Haller[1],
Alberto Griggio[2⋆⋆], and Daniel Kroening[1]

[1] Computer Science Department, University of Oxford, Oxford, UK
`first.last@cs.ox.ac.uk`
[2] Fondazione Bruno Kessler, Trento, Italy
`griggio@fbk.eu`

Abstract. DPLL(T) is a central algorithm for Satisfiability Modulo Theories (SMT) solvers. The algorithm combines results of reasoning about the Boolean structure of a formula with reasoning about conjunctions of theory facts to decide satisfiability. This architecture enables modern solvers to combine the performance benefits of propositional satisfiability solvers and conjunctive theory solvers. We characterise DPLL(T) as an abstract interpretation algorithm that computes a product of two abstractions. Our characterisation allows a new understanding of DPLL(T) as an instance of an abstract procedure to combine reasoning engines beyond propositional solvers and conjunctive theory solvers. In addition, we show theoretically that the split into Boolean and theory reasoning is sometimes unnecessary and demonstrate empirically that it can be detrimental to performance.

1 Introduction

The previous decade has witnessed the development of efficient solvers for deciding satisfiability of formulae in a wide range of logical theories. The development of these *Satisfiability Modulo Theory* (SMT) solvers can be understood as a consequence of three advances. Two advances are improvements in the performance of solvers for Boolean satisfiability, and for the conjunctive fragments of first-order theories such as equality with uninterpreted functions [12], difference logic [20], or linear rational arithmetic [10]. The third advance is DPLL(T), an algorithm that efficiently combines the strengths of propositional SAT solvers and conjunctive theory solvers to decide satisfiability of a theory formula [12].

We explain the principles of DPLL(T) with an example. A satisfiability checker for the formula φ below has to reason about Boolean combinations of equality constraints.

$$\varphi \hat{=} (x = y \lor y \neq z) \land x = z \land y = z \qquad \mathsf{BoolSkel}(\varphi) \hat{=} (p \lor \neg q) \land r \land q$$

⋆ Supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/J012564/1, and the FP7 STREP PINCETTE.

⋆⋆ Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

A DPLL(T) solver first constructs a *Boolean skeleton* of φ, given as $\mathsf{BoolSkel}(\varphi)$ above. The Boolean skeleton has the same structure as φ, but does not include information about the theory. If $\mathsf{BoolSkel}(\varphi)$ is unsatisfiable, so is φ. If $\mathsf{BoolSkel}(\varphi)$ is satisfiable, each satisfying assignment defines a conjunction of equality constraints. A solver for the conjunctive fragment of the theory can be then used to determine if the conjunction is satisfiable. If the conjunction defined by a specific satisfying assignment π to $\mathsf{BoolSkel}(\varphi)$ is not satisfiable, the solver can *learn* $\neg\pi$ and iterate the process above with $\mathsf{BoolSkel}(\varphi) \wedge \neg\pi$. Propositional and theory reasoning alternate in this manner until a first-order structure satisfying the theory formula is found, or until the formula is shown to be unsatisfiable.

The primary aim of this paper is to explain and analyse DPLL(T) in the abstract interpretation framework. We show that reasoning about the Boolean structure and about conjunctions of theory facts is, in a strict, mathematical sense, an abstract interpretation of the semantics of a formula. Extensions of DPLL(T) such as theory propagation, early pruning, theory explanations, conflict set generation and generation of multiple reasons for a single conflict have natural characterisations in the language of abstract interpretation.

We emphasise that the purpose of this work is not to trivialise DPLL(T) by claiming it is "just abstract interpretation". Instead we aim to illuminate the link between SMT solvers and abstract interpretation to allow the transfer of results and intuition. Though some of our results are intuitively clear and known to the satisfiability community, our formalisation is not obvious. Our work shows that DPLL(T) is an instance of a generic, greatest fixed point computation that overapproximates the reduced product of two abstract domains. This result allows the static analysis community to better place DPLL(T) in the rich landscape of results concerning fixed point computations and domain combinations.

The secondary aim of this paper is to show that the product construction involved in DPLL(T) is sometimes unnecessary. We empirically compare splitting-on-demand [2], an extension of classic DPLL(T), with ACDCL [14, 8], an algebraic generalisation of CDCL that does not operate over a product.[3]

Contributions This paper makes the following contributions.

1. A new understanding of DPLL(T) within the abstract interpretation framework. We show that DPLL(T) is an instance of a product construction over a Boolean abstraction and a conjunctive theory abstraction.
2. A view of DPLL(T) as an instance of a more abstract procedure which permits combination of reasoning engines beyond the classic Boolean-theory split.
3. A empirical demonstration that, under some circumstances, the construction of products in DPLL(T) is unnecessary and detrimental to performance.

Related work A number of recent publications have given abstract interpretation accounts of decision procedures: [7] gives an account of propositional SAT procedures such as DPLL and CDCL using the same framework as this paper which is the basis for the generalisation of CDCL in [8]. Independently of the above,

[3] Our benchmarks and an extended version of this paper with proofs can be found at http://www.cprover.org/papers/vmcai2013/

[23] gives an abstract-interpretation account and generalisation of Stålmarck's method. In [6], Nelson-Oppen theory combination is characterised as a product construction over abstract domains.

A number of practical approaches have been derived directly from this point of view. These include extensions of the CDCL algorithm to the interval abstraction to decide floating-point logic [14] and reachability queries [9], and the synthesis of abstract transformers using the generalisation of Stålmarck's method mentioned above [22]. Before these, [15] proposed combining propositional SAT solvers and abstract interpreters in a DPLL(T)-style architecture.

A popular operational formalisation of DPLL(T) is given in [21]. Our work is closely related to research efforts to develop alternatives to DPLL(T). These approaches, called *natural-domain* SMT [4], lift the CDCL algorithm to operate directly on theory formulae. Notable examples have been presented for equality logic with uninterpreted functions [1], linear real arithmetic and difference logic [19, 4], linear integer arithmetic [17], non-linear arithmetic [11, 18], and floating-point arithmetic [14].

2 Abstract Satisfaction

This section provides a concise review of SMT [3], abstract interpretation [5], and the application of abstract interpretation to logic [7].

2.1 Satisfiability Modulo Theories

A *signature* Σ is a set of *function symbols* and *predicate symbols*, each associated with a non-negative *arity*. Predicate and function symbols with arity zero are called, respectively, *propositions* and *constants*. Ground terms are constants or function applications $f(t_1, \ldots, t_n)$ where f is an n-ary function and the t_i are ground terms. All formulae we consider are quantifier-free and have no first-order variables. For convenience, we omit these qualifiers in the rest of the paper. As is common in the SMT literature, we refer to uninterpreted constants as variables.

An *atomic formula* is a proposition, an n-ary predicate $p(t_1, \ldots, t_n)$ applied to terms $t_1, \ldots, t_n$, or a truth value in $\mathbb{B} = \{\mathsf{t}, \mathsf{f}\}$. A *literal* is an atomic formula or its negation. A literal is in *positive phase* if it is an atomic formula and in *negative phase* otherwise. For a literal l, we denote by $\mathsf{neg}(l)$ its opposite-phase counterpart. For a set of formulae Ψ we denote by $\neg\Psi$ the set $\{\neg\psi \mid \psi \in \Psi\}$. A *clause* is a disjunction of literals, and a *formula* is in Conjunctive Normal Form (CNF) if it is a conjunction of clauses. We follow standard convention and denote clauses and CNF formulae as sets of literals, resp., sets of clauses where convenient. Unless otherwise specified, we assume all formulae to be in CNF. We denote by $\mathcal{A}(\varphi)$ the set of atomic subformulae of φ, by $\mathcal{L}(\varphi)$ the set of literals $\mathcal{A}(\varphi) \cup \neg\mathcal{A}(\varphi)$ and by $\mathcal{H}(\varphi)$ the set of terms occurring in φ. We denote by $\mathcal{V}(\varphi)$ the set of variables (uninterpreted constants) in φ.

Semantics Formulae are interpreted over first-order structures. A *structure* for a signature Σ is a pair (U, ϵ) consisting of a non-empty set U called the *universe* and an *interpretation function* ϵ which maps every element of the signature to an appropriate object over U, e.g. constants are mapped to elements of U, n-ary functions to n-ary functions over U, etc. We denote (U, ϵ) simply by ϵ when U is clear from context or irrelevant. The *semantic entailment relation* $\models$ is defined as usual. Given a structure σ and formula φ, if $\sigma \models \varphi$ holds, then σ satisfies φ, and it is a *model* of φ. Otherwise, it is a *countermodel.*

Theories We define a (Σ-)*theory* T_Σ as a set of first-order structures over a signature Σ (as is common in the SMT literature, e.g. [3]). We call a model $\sigma \in T_\Sigma$ of φ a T_Σ-model and a formula φ T_Σ-satisfiable if it has a T_Σ-model. The satisfiability problem modulo a theory T_Σ, for a quantifier-free ground formula φ, is to decide whether φ has a T_Σ-model.

Let P be a fixed set of propositions. A *propositional formula* is a P-formula, and a *propositional structure* or *propositional assignment* is an element of the set $\mathsf{PA}_P \mathrel{\hat{=}} P \to \mathbb{B}$. When discussing theories T_Σ in the context of propositional logic, we assume that P is disjoint from the signature Σ.

2.2 Abstract Interpretation

We briefly review some concepts in abstract interpretation. For convenience, we work in the Galois connection framework. We write $(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ for a Galois connection between the complete lattices C and A. An underapproximation is defined by a Galois connection $(C, \succeq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsupseteq)$. In this paper, we assume all Galois connections we consider to satisfy $\gamma(\bot) = \bot$ and $\gamma(\top) = \top$. A *transformer* is a monotone function on a lattice. A transformer f on a complete lattice has a greatest fixed point, denoted $\mathsf{gfp}\ f$ or $\mathsf{gfp}\ X.\ f(X)$ and a least fixed point $\mathsf{lfp}\ f$ or $\mathsf{lfp}\ X.\ f(X)$. The gfp *closure* f^* of a transformer f is the transformer $a \mapsto \mathsf{gfp}\ X.\ f(X) \sqcap a$. The *best approximation* of $f : C \to C$ is $g \mathrel{\hat{=}} \alpha \circ f \circ \gamma$.

A *reduction operator* is a transformer ρ in an abstract domain A that is (i) *reductive*, i.e., for all $a \in A$ it holds that $\rho(a) \sqsubseteq a$ and (ii) *sound*, i.e., $\gamma \circ \rho = \gamma$. Reductions refine the representation of an abstract object without changing its meaning. A *dual reduction operator* generalises the representation without changing the meaning of an object.

Let $(A, \sqsubseteq)$ be an overapproximation of a powerset domain $(\wp(S), \subseteq)$ with $\wp(S) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$. The *downset completion* of A is the lattice $\mathcal{D}(A) \mathrel{\hat{=}} (\mathsf{ds}(A), \subseteq)$ where $\mathsf{ds}(A)$ is the set of all $Q \in \wp(A)$ s.t. Q is *downwards closed*, i.e. $\forall a \in Q, a' \in A.\ a' \sqsubseteq a \implies a' \in Q$. When possible, we represent a set in $\mathsf{ds}(A)$ as the set of its maximal elements. It underapproximates the concrete domain $\wp(S)$ with $\alpha_\mathcal{D} : \wp(S) \to \mathcal{D}(A)$, $\alpha_\mathcal{D}(Q) \mathrel{\hat{=}} \{a \in A \mid \gamma(a) \subseteq Q\}$ and $\gamma_\mathcal{D} : \mathcal{D}(A) \to \wp(S)$, $\gamma_\mathcal{D}(D) \mathrel{\hat{=}} \bigcup_{d \in D} \gamma(d)$.

Let $(A, \sqsubseteq_A), (B, \sqsubseteq_B)$ be abstract domains over the concrete domain $(C, \subseteq)$, with Galois connections (α_A, γ_A) and (α_B, γ_B), respectively. The *Cartesian product* $A \times B$ is defined the abstract domain over the lattice $(A \times B, \sqsubseteq)$ with

$(a, b) \sqsubseteq (a', b')$ exactly if $a \sqsubseteq a'$ and $b \sqsubseteq b'$. There is a Galois connection to the concrete given by $\alpha_{A\times B}(c) = (\alpha_A(c), \alpha_B(c))$ and $\gamma_{A\times B}(a, b) = \gamma_A(a) \cap \gamma_B(b)$.

2.3 Interpreting Logics over Theories

We can use the formal machinery of abstract interpretation to approximate the meaning of logical formulae. Let T_Σ be a Σ-theory. The *concrete theory domain* of T_Σ is the powerset lattice $(\wp(T_\Sigma), \subseteq)$ together with the *model transformer* and *universal countermodel transformer* for each Σ-formula φ, given below.

$$\begin{aligned} mods_\varphi^{T_\Sigma}(S) &\hat{=} \{\sigma \in T_\Sigma \mid \sigma \in S \wedge \sigma \models \varphi\} \\ ucmods_\varphi^{T_\Sigma}(S) &\hat{=} \{\sigma \in T_\Sigma \mid \sigma \in S \vee \sigma \not\models \varphi\} \end{aligned}$$

Abstractions of these operators are implemented in existing abstract domains for program analysis for the following reason. The function $mods_\varphi^{T_\Sigma}$ is equivalent to the strongest post-condition of an $\mathsf{assume}(\varphi)$ statement, while $ucmods_\varphi^{T_\Sigma}$ is equivalent to the weakest liberal pre-condition. In logical inference terms, $mods_\varphi^{T_\Sigma}$ implements *deduction*, since it maps a set of structures S to the strongest consequence of S w.r.t. φ, expressed as a set. Similarly, $ucmods_\varphi^{T_\Sigma}$ implements *abduction*, because it maps an element R to the weakest explanation for R.

Abstract domains and transformers can be used to perform sound but incomplete satisfiability checks. We refer to an abstraction of the concrete theory domain as an *abstract theory domain*.

Theorem 1 (Abstract Satisfaction). *Let amods be an overapproximation of $mods_\varphi^{T_\Sigma}$ and aucmods be an underapproximation of $ucmods_\varphi^{T_\Sigma}$. The formula φ is not T_Σ-satisfiable (i)* gfp *$amods = \bot$ or (ii)* lfp *$aucmods = \top$.*

Refutational Completeness in Abstract Interpretation Let f be a concrete transformer and g be a sound approximation of f in a lattice A, and a be an element of A. Then g is *γ-complete at a* if $\gamma \circ g(a) = f \circ \gamma(a)$ holds.

We now introduce new notions of completeness to express adequate precision. The transformer g is *$\gamma_\bot$-complete* at $a \in A$ if $\gamma \circ g(a) = \bot$ exactly if $f \circ \gamma(a) = \bot$, and it is *$\bot$-complete* at $a \in A$ if $g(a) = \bot$ whenever $f \circ \gamma(a) = \bot$. If a transformer is $\bot$-complete at every element we simply say it is $\bot$-complete. The same holds for γ- and $\gamma_\bot$-completeness. A reduction operator is $\bot$-complete (respectively γ- or $\gamma_\bot$-complete) if it is complete w.r.t. the concrete identity function.

3 Boolean Reasoning as Abstract Interpretation

This section shows that the Boolean reasoning employed by the DPLL(T) algorithm is an instance of abstract interpretation. More precisely, we show that computing propositional solutions over the Boolean skeleton of a formula is an abstract interpretation of the formula's theory semantics.

Fix φ to be a Σ-formula and $P \subseteq Props$ to be a fresh set of propositions disjoint from Σ. We assume a bijective function $\mathsf{pmap} : \mathcal{A}(\varphi) \to P$ that relates the atoms in φ to the propositions in P.

Definition 1. *The* Boolean skeleton $\mathsf{BoolSkel}(\varphi)$ *is the propositional formula obtained by replacing each atomic formula* $\psi_{\mathcal{A}}$ *occurring in* φ *with* $\mathsf{pmap}(\psi_{\mathcal{A}})$.

Reasoning about Boolean structure can be understood as an abstraction of the semantics of a formula. From this perspective, the introduction of propositions for subformulae, and consequently the construction of an independent, propositional formula can be considered an implementation detail.

Definition 2. *For a set of* Σ*-formulae* F *we define the* Boolean abstraction Bool_F *as the abstract lattice* $(\wp(F \to \mathbb{B}), \subseteq)$ *with the Galois connection below.*

$$(\wp(T_\Sigma), \subseteq) \underset{\alpha_{\mathsf{B}}}{\overset{\gamma_{\mathsf{B}}}{\leftrightarrows}} (\mathsf{Bool}_F, \subseteq)$$

$$\alpha_{\mathsf{B}}(S) \hat{=} \{\beta \in F \to \mathbb{B} \mid \exists \sigma \in S\ \forall \psi \in F.\ \sigma \models \psi \iff \beta(\psi) = \mathsf{t}\}$$

$$\gamma_{\mathsf{B}}(B) \hat{=} \{\sigma \in T_\Sigma \mid \exists \beta \in B\ \forall \psi \in F.\ \sigma \models \psi \iff \beta(\psi) = \mathsf{t}\}$$

DPLL(T) applied to a formula φ employs the Boolean abstraction $\mathsf{Bool}_{\mathcal{A}(\varphi)}$. A set of propositional assignments from PA_P represents an element of $\mathsf{Bool}_{\mathcal{A}(\varphi)}$. We can move between these views by lifting pmap to map a set $S \subseteq \mathcal{A}(\varphi) \to \mathbb{B}$ bijectively to a subset of PA_P by mapping each assignment from subformulae to truth values to its corresponding assignment from propositions to truth values. Formally, we define $\mathsf{pmap}(S) \hat{=} \{\lambda a.\beta(\mathsf{pmap}(a)) \mid \beta \in S\}$.

Relating Boolean Abstractions and the Skeleton The set of propositional models can be computed by implementing an abstract transformer on $\mathsf{Bool}_{\mathcal{A}(\varphi)}$.

Proposition 1. *Let* $\psi = \mathsf{BoolSkel}(\varphi)$*, then the* skeleton transformer

$$\mathsf{BSkelModels} \hat{=} \mathsf{pmap}^{-1} \circ \mathit{mods}_{\psi}^{\mathsf{PA}_P} \circ \mathsf{pmap}$$

is a sound overapproximation of the model transformer $\mathit{mods}_{\varphi}^{T_\Sigma}$.

The object amods_φ defined above is not the best overapproximation of the model transformer, since it only captures Boolean, but not theory reasoning. It is still precise when considered in the concrete.

Proposition 2. $\mathsf{BSkelModels}$ *is* $\gamma_\perp$*-complete w.r.t.* $\mathit{mods}_{\varphi}^{T_\Sigma}$.

In other words, even though the resulting element may not be the best abstract representation of the set of models of φ, its concretisation is precise. The remaining question is how one can determine whether the set of models it represents is empty. In DPLL(T), this is performed using a satisfiability check.

Definition 3. *The function* $\mathsf{BoolCheck} : \mathsf{Bool}_F \to \mathsf{Bool}_F$*, defined below, eliminates assignments not consistent in the theory.*

$$\mathsf{BoolCheck}(B) \hat{=} \Big\{\beta \in B \mid \bigwedge\{\varphi \mid \beta(\varphi) = \mathsf{t}\} \cup \{\neg\varphi \mid \beta(\varphi) = \mathsf{f}\} \textit{ is } T_\Sigma\textsf{-SAT}\Big\}$$

Proposition 3. $\mathsf{BoolCheck}$ *is a* $\perp$*-complete reduction operator over* Bool_F.

Example 1. Consider the first-order formula below.

$$\varphi \mathrel{\hat{=}} (x = y) \ \wedge \ (\neg(y = z) \ \vee \ \neg(x = z))$$

We fix the theory T to give equality its natural interpretation. We denote by $v_1 v_2 v_3$ the assignment $\{(x = y) \mapsto v_1, (y = z) \mapsto v_2, (x = z) \mapsto v_3\}$ in $\mathcal{A}(\varphi) \to \mathbb{B}$. For the mapping $\mathsf{pmap} \mathrel{\hat{=}} \{(x = y) \mapsto p, (y = z) \mapsto q, (x = z) \mapsto r\}$ we obtain the Boolean skeleton below, which yields a skeleton transformer.

$$\mathsf{BoolSkel}(\varphi) \mathrel{\hat{=}} p \wedge (\neg q \vee \neg r) \qquad \mathsf{BSkelModels}(\top) = \{\mathsf{ttf}, \mathsf{tft}, \mathsf{tff}\}$$

$\mathsf{BSkelModels}(\top)$ contains the assignment ttf which represents the empty set, since no structure in the theory satisfies $x = y$, $y = z$ but not $x = z$. The same holds for tft. Since both represent the empty set, this does not affect the precision of the transformer in the concrete, i.e., the transformer is γ-complete at $\top$ since $\gamma_{\mathsf{B}}(\mathsf{BSkelModels}(\top))$ is equal to $mods_{\varphi}^{T}(\top)$. Calling $\mathsf{BoolCheck}(\{\mathsf{ttf}, \mathsf{tft}, \mathsf{tff}\})$ refines the representation to $\{\mathsf{tff}\}$.

Satisfiability via Deduction and Reduction We reformulate the initial step of DPLL(T) using abstract interpretation. Let $amods_{\varphi}$ be a $\gamma_{\perp}$-complete approximation of $mods_{\varphi}^{T_\Sigma}$ and let ρ be a $\perp$-complete reduction operator.
Step 1 Compute $a = amods_{\varphi}$ (e.g. with $amods_{\varphi} = \mathsf{BSkelModels}$)
Step 2 Return SAT if $\rho(a) \neq \perp$ (e.g. with $\rho = \mathsf{BoolCheck}$)

We can sketch DPLL(T) as depth-first variant of the above framework. Propositional models are enumerated on-the-fly by a SAT solver rather than computed in a single step; the reduction to $\perp$ is computed and checked by a theory solver. The following summarises the soundness and completeness argument.

Proposition 4. *If $amods_{\varphi}$ is $\gamma_{\perp}$-complete w.r.t. $mods_{\varphi}^{T_\Sigma}$ and ρ is a $\perp$-complete reduction, then $\rho(amods_{\varphi}(\top)) \neq \perp$ exactly if φ is T_Σ-satisfiable.*

3.1 Efficient Disjunction via the Cartesian Abstraction

The transformer $\mathsf{BSkelModels}$ generates the set of models of a propositional formula and is hence expensive to compute. Therefore, DPLL(T) instead uses a guided search process to enumerate models.

Partial Assignments and the Cartesian Abstraction The main data structure for the guided search in a DPLL(T) solver is a partial assignment, a map from propositions to t, f, an unknown value $\top$ or a value $\perp$ representing a conflict. Partial assignments are refined using deduction and search. A partial assignment $f : P \to \{\mathsf{t}, \mathsf{f}, \top, \perp\}$ represents a set of propositional literals Q such that $f(p) = \mathsf{t}$, $f(p) = \mathsf{f}$, $f(p) = \top$ and $f(p) = \perp$ represent, respectively, that $p \in Q$, $\neg p \in Q$, $p, \neg p \notin Q$ and $p, \neg p \in Q$. Since we view the Boolean skeleton as an implementation detail, the description below directly uses atomic formulae.

Definition 4. *For a set of Σ-formulae F we define the* Cartesian abstraction Cart_F *as the abstract lattice $(\wp(F \cup \neg F), \sqsubseteq)$ with $\sqsubseteq = \supseteq$, $\bigsqcap = \bigcup$ and $\bigsqcup = \bigcap$.*

Cart_F *abstracts* Bool_F *(and, as a consequence, the concrete theory domain). The Galois connections are as below.*

$$\begin{tikzcd}
(\wp(T_\Sigma), \subseteq) \arrow[r, shift right, "\alpha_{\mathsf{B}}"'] \arrow[rr, bend right=20, "\alpha_{\mathsf{C}} \mathrel{\hat{=}} \alpha_{\mathsf{BC}} \circ \alpha_{\mathsf{B}}"'] & (\mathsf{Bool}_F, \subseteq) \arrow[l, shift right, "\gamma_{\mathsf{B}}"'] \arrow[r, shift right, "\alpha_{\mathsf{BC}}"'] & (\mathsf{Cart}_F, \sqsubseteq) \arrow[l, shift right, "\gamma_{\mathsf{BC}}"'] \arrow[ll, bend right=10, "\gamma_{\mathsf{C}} \mathrel{\hat{=}} \gamma_{\mathsf{BC}} \circ \gamma_{\mathsf{B}}"']
\end{tikzcd}$$

$$\alpha_{\mathsf{BC}}(B) \mathrel{\hat{=}} \{\psi \mid \forall \beta \in B.\ \beta(\psi) = \mathsf{t}\} \sqcap \{\neg\psi \mid \forall \beta \in B.\ \beta(\psi) = \mathsf{f}\}$$
$$\gamma_{\mathsf{BC}}(\Theta) \mathrel{\hat{=}} \{\beta \mid \forall \psi \in F.\ (\beta(\psi) = \mathsf{t} \Rightarrow \neg\psi \notin \theta) \wedge (\beta(\psi) = \mathsf{f} \Rightarrow \psi \notin \theta)\}$$

The use of propositional partial assignments in existing DPLL(T) solvers can be viewed as a way of representing $\mathsf{Cart}_{\mathcal{A}(\varphi)}$.

Unit Rule and BCP DPLL(T) solvers perform Boolean reasoning over partial assignments using the *unit rule*, which states that if all but one literal in a propositional clause are contradicted by the current partial assignment, the remaining literal must be true. Below, we give the corresponding transformer over Cart_F.

Definition 5. *For a Σ-clause C and set of formulae F with $\mathcal{A}(C) \subseteq F$, the* unit rule *over Cart_F is the function $\mathsf{unit}_C^F : \mathsf{Cart}_F \to \mathsf{Cart}_F$ defined as:*

$$\mathsf{unit}_C^F(\Theta) \mathrel{\hat{=}} \begin{cases} \bot & \text{if } \psi, \neg\psi \in \Theta \text{ or for all } l \in C,\ \mathsf{neg}(l) \in \Theta \\ \Theta \sqcap \{l\} & \text{else if } C = C' \cup \{l\} \text{ s.t. for all } l' \in C',\ \mathsf{neg}(l') \in \Theta \\ \Theta & \text{otherwise} \end{cases}$$

For a set of propositions P and propositional clause C, the propositional unit rule *is the rule $\mathsf{unit}_C^P : \mathsf{Cart}_P \to \mathsf{Cart}_P$.*

Example 2. Consider the formula from before, $\varphi \mathrel{\hat{=}} (x = y) \wedge C$ where $C = (\neg(y = z) \vee \neg(x = z))$. We can apply $\mathsf{unit}_{x=y}^{\mathcal{A}(\varphi)}(\top)$ to obtain the element $\Theta = \{x = y\}$. Applying $\mathsf{unit}_C^{\mathcal{A}(\varphi)}(\Theta)$ gives no new information but simply returns Θ. We can refine the element with an unsound assumption by computing $\Theta' = \Theta \sqcap \{y = z\} = \{x = y, y = z\}$. Now, applying $\mathsf{unit}_C^{\mathcal{A}(\varphi)}(\Theta')$ yields $\Theta' \sqcap \{\neg(x = z)\}$.

Unit rule applications soundly approximate the model transformer, regardless of the underlying theory.

Proposition 5. *Let C be a clause such that $\mathcal{A}(C) \subseteq F$. For any theory T_Σ, the transformer unit_C^F is a sound approximation of $\mathsf{mods}_C^{T_\Sigma}$.*

DPLL(T) solvers use a process called Boolean Constraint Propagation (BCP) in which the unit rule is applied exhaustively to deduce new theory facts. This process computes a greatest fixed point with the function defined earlier.

Definition 6. *For a Σ-formula φ and a set of Σ-formulae $F \supseteq \mathcal{A}(\varphi)$, the* BCP *transformer $\mathsf{bcp}_\varphi : \mathsf{Cart}_F \to \mathsf{Cart}_F$ is the following function.*

$$\mathsf{bcp}_\varphi(\Theta) \mathrel{\hat{=}} \mathsf{gfp}\ X.\ \bigsqcap_{C \in \varphi} \mathsf{unit}_C^F(X \sqcap \Theta)$$

During the run of DPLL(T), the propositional formula changes in a process called learning. Here, we take the point of view that the use of a propositional formula is an implementation detail. Changing the propositional formula then amounts to refining the model transformer over the Cartesian abstraction.

3.2 Satisfiability via Abstract Splitting

In lazy DPLL(T), theory consistency is checked once a partial assignment that satisfies every clause is found. The following operator is used for the check.

Definition 7. *We define* $\mathsf{CartCheck} : \mathsf{Cart}_F \to \mathsf{Cart}_F$ *as*

$$\mathsf{CartCheck}(\Theta) \mathrel{\hat{=}} \begin{cases} \bot & \textit{if } \bigwedge \theta \textit{ is not } T_\Sigma\textit{-satisfiable} \\ \Theta & \textit{otherwise} \end{cases}$$

Proposition 6. $\mathsf{CartCheck}$ *is a* $\bot$*-complete reduction operator.*

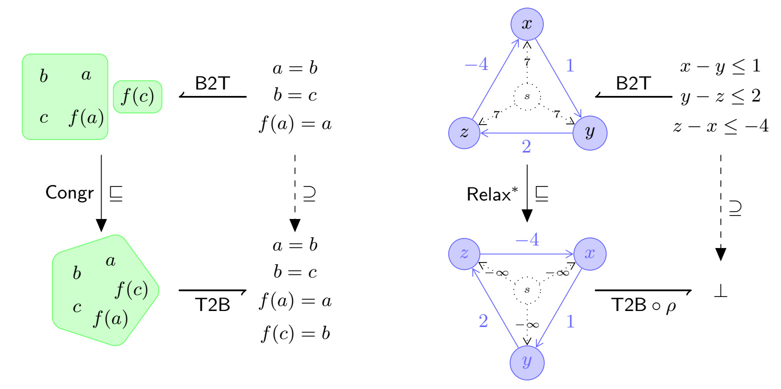
The previous section showed that bcp_φ soundly approximates the model transformer and $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ is a $\bot$-complete reduction. Proposition 4 cannot be applied though, since bcp_φ lacks the necessary completeness requirement and solely performing deduction and reduction does not give a complete procedure. In the absence of this global completeness, DPLL(T) searches for points at which the model transformer is locally complete. The proposition below shows that a common stopping criterion in DPLL(T) is a local completeness check.

Proposition 7. *Let* φ *be a* Σ*-formula in* CNF*, and let* $\Theta \in \mathsf{Cart}_{\mathcal{A}(\varphi)}$ *such that for every clause* $C \in \varphi$ *there is a literal* $l \in C$ *such that* $l \in \Theta$*. Then* bcp_φ *is* γ*-complete at* Θ*.*

The search proceeds as follows. After the BCP step, classic DPLL chooses a variable in a partial assignment that is assigned to $\top$ and explores separately the cases where it is t and f. In terms of abstract interpretation this amounts to decomposing a partial assignment a into two more precise assignments a_1, a_2 that, taken together, have the same meaning as the original assignment, i.e., $\gamma(a_1) \cup \gamma(a_2) = a$. Let $amods_\varphi : A \to A$ be a sound approximation of $mods_\varphi^{T_\Sigma}$ and let $\rho : A \to A$ be a $\bot$-complete reduction, then we can state the abstract algorithm as follows.

(Init) Let $a_{\mathsf{init}} = \top$.
Step 1 Compute the greatest fixed point $a = \mathsf{gfp}\ X.amods_\varphi(X \sqcap a_{\mathsf{init}})$.
Step 2 If $a = \bot$ then return.
Step 3 If $amods_\varphi$ is $\gamma_\bot$-complete at a and $\rho(a) \neq \bot$ then return SAT.
Step 4 Split a into two smaller elements a_1, a_2 s.t. $a_1 \sqsubset a$, $a_2 \sqsubset a$ and $\gamma(a_1) \cup \gamma(a_2) = \gamma(a)$, and call the algorithm recursively.

(a) If a call with $a_{\mathsf{init}} = a_1$ returns SAT then return SAT
(b) If a call with $a_{\mathsf{init}} = a_2$ returns SAT then return SAT



(a) EUF with congruence closure (b) DL with Bellman-Ford

Fig. 1. Examples of theory solvers as abstract domains

Example 3. Consider again the formula $\varphi \hat{=} x = y \wedge C$ where C is the clause $(\neg(y = z) \vee \neg(x = z))$ We fix the theory T to give equality its natural interpretation. Computing $\mathsf{bcp}_\varphi(\top)$ yields the result $a = \{x = y\}$. This is not γ-complete reasoning, since it abstracts structures where $x = y = z$, which are not models. We split Θ into the smaller elements $a_1 = a \sqcap \{y = z\}$ and $a_2 = a \sqcap \{\neg(y = z)\}$. In the first recursive call, we obtain $a' = \{x = y, y = z, \neg(x = z)\}$ from $\mathsf{bcp}_\varphi(a_1)$. The transformer bcp_φ is γ-complete at a', therefore we know that a' is a set of models. It remains to check whether a' is an empty set of models, by calling $\mathsf{CartCheck}(a')$, which returns $\bot$. In the second recursive call, BCP yields no further refinement. But bcp_φ is already γ-complete at a_2, therefore we check the conjunction $(x = y) \wedge \neg(y = z)$ with $\mathsf{CartCheck}(a_2)$. The check returns a_2, indicating that a_2 represents a non-empty set and we return SAT.

Depending on details of the logic and abstract domain used the above algorithm may not be complete, i.e., it may not return SAT exactly if φ is satisfiable. We will discuss conditions for completeness in a bit more detail later. Whenever the algorithm returns SAT, then the formula is satisfiable.

Proposition 8. *Let $amods_\varphi : A \to A$ be an overapproximation of $mods_\varphi^{T_\Sigma}$ and ρ be a $\bot$-complete reduction operator. If for some element $a \in A$, $amods_\varphi$ is $\gamma_\bot$-complete at a and $\rho(amods_\varphi(a)) \neq \bot$, then φ is satisfiable.*

4 Theory Solvers as Abstract Domains

In this section, we show that theory solvers for equality with uninterpreted functions, and for difference logic can be viewed as reduction operators. These serve as examples of the general approach as it is not feasible to cover all theory solvers in one paper.

4.1 Equality with Uninterpreted Functions

An *equality formula* contains the predicate $=$ and function symbols. We use $t \neq t'$ as a shorthand to denote $\neg(t = t')$. We define the theory of Equality with Uninterpreted Functions (EUF) as the set T_{EUF} containing all structures $(\mathbb{Z}, \epsilon)$ where ϵ interprets $=$ as the standard equality relation over $\mathbb{Z}$. The congruence closure algorithm decides satisfiability of conjunctions of equality literals. The algorithm constructs congruence classes containing terms from $\mathcal{H}(\varphi)$ (often implemented using union-find data structures) and a set of pairs in $\mathcal{H}(\varphi)$ that are known to be unequal. The data structure used by congruence closure forms a lattice. A *partition* of a set X is a collection of disjoint, non-empty subsets of X whose union is X. $\mathsf{Part}(X)$ denotes the partitions of a set X.

Definition 8. *For an* EUF φ*, the* EUF abstraction*,* EUF_φ *is* $(\mathsf{TS}, \sqsubseteq)$ *where:*

$$\mathsf{TS} \mathrel{\hat{=}} \mathsf{Part}(\mathcal{H}(\varphi)) \times \wp(\mathcal{H}(\varphi) \times \mathcal{H}(\varphi))$$

and $(P, D) \sqsubseteq (P', D')$ *exactly if* $\forall p' \in P'. \exists p \in P$ *s.t.* $p \supseteq p'$ *and* $D \supseteq D'$*. Note that* EUF_φ *abstracts the concrete and refines* $\mathsf{Cart}_{\mathcal{A}(\varphi)}$*. As both domains are lattices,* α_{TS} *and* $\mathsf{B2T}$ *are uniquely defined from* γ_{TS} *and* $\mathsf{T2B}$*.*

$$\begin{tikzcd} (\wp(T_\Sigma), \subseteq) \arrow[r, shift right, "\alpha_{\mathsf{TS}}"'] & (\mathsf{TS}, \sqsubseteq) \arrow[l, shift right, "\gamma_{\mathsf{TS}}"'] \arrow[r, shift right, "\mathsf{T2B}"'] & (\mathsf{Cart}_{\mathcal{A}(\varphi)}, \supseteq) \arrow[l, shift right, "\mathsf{B2T}"'] \end{tikzcd}$$

$$\gamma_{\mathsf{TS}}(P, D) \mathrel{\hat{=}} \{\sigma \mid \forall (t_1, t_2) \in D.\ \sigma \models t_1 \neq t_2 \land \forall p \in P\ \forall t_1, t_2 \in p.\ \sigma \models t_1 = t_2\}$$
$$\mathsf{T2B}(P, D) \mathrel{\hat{=}} \mathcal{L}(\varphi) \cap (\{t_1 = t_2 \mid \exists p \in P.\ t_1, t_2 \in p\} \cup \{t_1 \neq t_2 \mid (t_1, t_2) \in D\})$$

We define the steps of the algorithm as transformers over the abstraction. A *congruence operator* $\mathsf{Congr} : \mathsf{EUF}_\varphi \to \mathsf{EUF}_\varphi$ merges the congruence classes of two terms if all their subterms s, t are pairwise *congruent* in the current element P, i.e., if they are in the same congruence class. If in $(P, D) \in \mathsf{EUF}_\varphi$ terms are found to both equal and unequal, i.e., for some $p \in P$ and $(t_1, t_2) \in D$ it holds that $t_1, t_2 \in p$, then $\bot$ is returned. Otherwise, we define for a partition $P = \{p_1, \ldots, p_k\}$:

$$\mathsf{Congr}(P, D) \mathrel{\hat{=}} \begin{cases} (P \setminus \{p, p'\} \cup \{p \cup p'\}, D) \text{ for some disjoint } p, p' \in P \text{ s.t.} \\ \quad f(s_1, \ldots, s_k) \in p, f(t_1, \ldots, t_k) \in p' \text{ s.t. all } s_i, t_i \text{ are congr. in } P \\ (P, D) \text{ if no such } p, p' \text{ exist} \end{cases}$$

The congruence operator is reductive (it gains in precision in each step), and refines the representation of a set of structures without changing the set itself. Figure 1(a) illustrates Congr along with the Galois connection between the EUF and the Cartesian abstractions. The set of formulae in the top right can be concretised to the pair of congruence classes in the top left. These are then merged by Congr as $a = c$ implies $f(a) = f(c)$ and finally can be abstracted to give the set of formulae in the bottom right; simulating inference in the Cartesian domain.

Proposition 9. Congr *is a reduction operator.*

The congruence closure algorithm then computes the greatest fixed point $\mathsf{gfp}\ \mathsf{Congr}$ over EUF_φ by iterating Congr until no new information can be deduced. It is a refutationally complete procedure, i.e., if a conjunction of equality literals is empty, then the fixed point will be $\perp$.

Proposition 10. *The* gfp *closure* Congr^* *is* $\perp$*-complete.*

4.2 Difference Logic

Formulae in *difference logic* (DL) contain the binary function symbol $-$ and the binary predicate $\leq$, and have atoms of the form $x - y \leq c$. The theory of *integer difference logic* (T_{IDL}) is the set of structures of the form $(\mathbb{Z}, \epsilon)$ where ϵ maps the symbols $\leq$ and $-$ to their natural interpretations over the integers.

A conjunct of difference logic atoms can be modelled by a weighted directed graph in which the set of nodes N corresponds to the set of variables in the conjunct. An atom $x - y \leq c$ is denoted as an edge (x, y) with weight c. The conjunct is satisfiable if and only if the graph contains no negative cycles.

Negative cycles can be detected using the Bellman-Ford algorithm (BF). The main data structure of BF associates a *weight* in $\mathbb{Z}_\infty \mathrel{\hat{=}} \mathbb{Z} \cup \{-\infty, \infty\}$ with each node n. The weight is an upper bound on the shortest path from the source to n. The weight $-\infty$ indicates a negative cycle. For handling DL, we choose the source to be s, a fresh node, and assume that s is connected to all variables with weight M_φ, which is an integer constant larger than the longest possible path.[4] The initial node weights are also M_φ. Node weights are reduced in each round if there is a neighbouring node that gives a shorter, negative cost path. After $|N| - 1$ iterations, the path lengths will have converged if and only if there are no negative cycles. If a final iteration changes the scores, the graph contains a negative cycle.

We make two observations which allow us to simplify presentation: (i) since edge weights represent upper bounds on the minimal distance between two variables, node weights can simply be viewed as special edges (s, n), (ii) BF can then be viewed to operate solely over edge weights (missing edges are given weight ∞). For a formula φ, we define the edge set E_φ as the set $(\{s\} \cup \mathcal{V}(\varphi)) \times \mathcal{V}(\varphi)$, where s is the fresh source node.

Definition 9. *For a* DL*-formula* φ*, the* BF abstraction BF_φ *is* $(\mathsf{TS}, \sqsubseteq)$ *where:*

$$\mathsf{TS} \mathrel{\hat{=}} \{f : E_\varphi \to \mathbb{Z}_\infty \mid \forall x \in \mathcal{V}(\varphi).\ f(s,x) \leq M_\varphi\}$$
$$f \sqsubseteq g \text{ iff } \forall e \in E_\varphi.\ f(e) \leq g(e)$$

BF_φ *abstracts the concrete and refines* $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ *and again, only half of each Galois connection is explicitly defined.*

$$(\wp(T_\Sigma), \subseteq) \underset{\alpha_{\mathsf{TS}}}{\overset{\gamma_{\mathsf{TS}}}{\leftrightarrows}} (\mathsf{TS}, \sqsubseteq) \underset{\mathsf{T2B}}{\overset{\mathsf{B2T}}{\leftrightarrows}} (\mathsf{Cart}_{\mathcal{A}(\varphi)}, \supseteq)$$
$$\gamma_{\mathsf{TS}}(f) \mathrel{\hat{=}} \{\sigma \mid \forall (x,y) \in \mathcal{V}(\varphi) \times \mathcal{V}(\varphi).\ \sigma \models\ x - y \leq f(x,y)\}$$
$$\mathsf{B2T}(\Theta) \mathrel{\hat{=}} \lambda(x,y).\ \min(\{k \mid x - y \leq k \in \Theta\} \cup \{\top_{\mathsf{BF}}(x,y)\})$$

[4] E.g., M_φ can be the sum of the absolute values of all the integer constants in φ.

As in the case of EUF, the steps of the algorithm are reduction operators. In the case of BF, there are two reductions; the relax step and the cycle check.

Proposition 11. $\mathsf{Relax} : \mathsf{BF}_\varphi \to \mathsf{BF}_\varphi$ *and* $\mathsf{NegC} : \mathsf{BF}_\varphi \to \mathsf{BF}_\varphi$ *are reductions:*

$$\mathsf{Relax}(f)(x,y) \mathrel{\hat{=}} \begin{cases} f(x,y) & x \neq s \\ \min(\{f(x,y)\} \cup \{f(x,z)+f(z,y) \mid z \in \mathcal{V}(\varphi)\}) & x = s \end{cases}$$

$$\mathsf{NegC}(f) \mathrel{\hat{=}} \begin{cases} \bot & \textit{if } \mathsf{Relax}^{|\mathcal{V}(\varphi)|} \neq \mathsf{Relax}^{|\mathcal{V}(\varphi)|-1} \\ \mathsf{Relax}^{|\mathcal{V}(\varphi)|} & \textit{otherwise} \end{cases}$$

In addition to the above function, consider a simple canonicity reduction ρ s.t. $\rho(f) = \bot$ if f maps some edge to $-\infty$ and $\rho(f) = f$ otherwise. Relax, ρ and the Galois connections to the Cartesian domain are shown in Figure 1(b). Similarly to Figure 1(a), the Cartesian domain is on the right and by mapping to the concrete (BF on the left) and performing reduction, it is possible to find the inconsistency. The function NegC can then be viewed as a fixed point computation (not based on Kleene iteration) over the relaxation function.

Proposition 12. NegC *computes the fixed point* $(\rho \circ \mathsf{Relax})^*$ *and is* $\bot$*-complete.*

5 DPLL(T) as a Product Construction

We have given separate accounts of the Boolean and theory reasoning components of DPLL(T) as abstract interpretation. We now show that DPLL(T) can be viewed to compute a fixed points over a product between the Cartesian abstraction over the formula atoms $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ and an abstract theory domain TS.

Definition 10. *We define a* DPLL(T) theory domain *to be an abstract lattice* $(\mathsf{TS}, \sqsubseteq)$ *such that the following conditions hold.*
(i) TS *abstracts the concrete with Galois connection* $(\alpha_{\mathsf{TS}}, \gamma_{\mathsf{TS}})$,
(ii) $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ *abstracts* TS *with Galois connection* $(\mathsf{T2B}, \mathsf{B2T})$,
(iii) $\gamma_{\mathsf{C}} = \gamma_{\mathsf{TS}} \circ \mathsf{B2T}$ *and* $\alpha_{\mathsf{C}} = \alpha_{\mathsf{TS}} \circ \mathsf{T2B}$.

$$\begin{tikzcd}
(\wp(T_\Sigma), \subseteq) \arrow[r, shift right, "\alpha_{\mathsf{TS}}"'] \arrow[rr, bend right, "\alpha_{\mathsf{C}} \mathrel{\hat{=}} \mathsf{T2B} \circ \alpha_{\mathsf{TS}}"'] & (\mathsf{TS}, \sqsubseteq) \arrow[l, shift right, "\gamma_{\mathsf{TS}}"'] \arrow[r, shift right, "\mathsf{T2B}"'] & (\mathsf{Cart}_{\mathcal{A}(\varphi)}, \sqsubseteq) \arrow[l, shift right, "\mathsf{B2T}"'] \arrow[ll, bend right, "\gamma_{\mathsf{C}} \mathrel{\hat{=}} \gamma_{\mathsf{TS}} \circ \mathsf{B2T}"']
\end{tikzcd}$$

The first condition ensures that datastructure of the theory solver represent sets of T_Σ structures. The other conditions require some motivation: The second condition ensures that conjunctions of literals in $\mathcal{A}(\varphi)$ can be expressed in TS without loss of precision. This corresponds to the requirement that the logic fragment handled by the theory solver includes conjunctions over $\mathcal{A}(\varphi) \cup \neg\mathcal{A}(\varphi)$, i.e., that satisfiability queries generated by $\mathsf{CartCheck}$ can be expressed. For convenience, we use a Galois connection to model this relation, even though in practice a weaker relation between the two might suffice. We assume that $\mathsf{T2B}$ and $\mathsf{B2T}$ can be computed. The third condition ensures that the Galois connections are compatible. We can now formally define DPLL(T) abstractions.

Definition 11. *For a T_Σ-formula φ and a* DPLL(T) *theory domain* TS*, the* DPLL(T) abstract domain $\mathsf{DPLL(TS)}$ *is the product domain* $\mathsf{Cart}_{\mathcal{A}(\varphi)} \times \mathsf{TS}$.

Example 4. We consider equality formulae φ. EUF_φ is a DPLL(T) theory domain, since it abstracts the concrete, and it refines the Cartesian abstraction.

We illustrate operations described in this section over $\mathsf{DPLL}(\mathsf{EUF}_\varphi)$. For convenience, we denote for three terms x, $f(x)$, z the partition $\{\{x\}, \{f(x), z\}\}$ either by $[x][f(x), z]$ or simply by $[f(x), z]$, omitting singleton partitions.

BCP with Theory Propagation The classic DPLL(T) architecture only uses theory reasoning to check satisfiability of candidates. *Theory propagation* is a common refinement of this basic architecture. There, an element $\Theta \in \mathsf{Cart}_{\mathcal{A}(\varphi)}$ is refined with information deduced in the theory solver. One propagation step in a DPLL(T) solver with theory propagation can be broken down into these substeps:

(i)	*Boolean deduction:*	Perform Boolean reasoning.
(ii)	*Theory instantiation:*	Communicate Boolean facts to theory.
(iii)	*Theory deduction:*	Perform theory reasoning.
(iv)	*Theory propagation:*	Find implied Boolean consequences.

Definition 12. *We define the* theory instantiation *and* theory propagation *transformers over* DPLL(T) *below.*

$$\mathsf{tinst}(\Theta, \mathsf{te}) \mathrel{\hat{=}} (\Theta, \mathsf{te} \sqcap \mathsf{B2T}(\Theta)) \qquad \mathsf{tprop}(\Theta, \mathsf{te}) \mathrel{\hat{=}} (\Theta \sqcap \mathsf{T2B}(\mathsf{te}), \mathsf{te})$$

Example 5. We assume that $\mathcal{A}(\varphi) = \{x = y,\ y = z\}$. Consider the element $(\Theta, \mathsf{te}) \mathrel{\hat{=}} (\{x = y\}, ([x][y][z], \{y, z\})$ of $\mathsf{DPLL}(\mathsf{EUF}_\varphi)$. Applying $\mathsf{tinst}(\Theta, \mathsf{te})$ yields $(\Theta, ([x, y][z], \{y, z\}))$. Applying $\mathsf{tprop}(\Theta, \mathsf{te})$ yields $(\{x = y, \neg(y = z)\}, \mathsf{te})$. Neither operator changes the semantics of the tuple.

Proposition 13. *The transformers* tinst *and* tprop *are reductions over* $\mathsf{DPLL(TS)}$.

We note that *early pruning* [3] is just a special case of theory propagation in the lattice theoretic setting, i.e., it is the case where theory propagation finds $\bot$.

Deduction over $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ is performed using the unit rule, while deduction inside the theory solver is handled by some reduction operator.

Definition 13. *The* Boolean deduction transformer bded_φ *is a sound overapproximation of* $mods_\varphi^{T_\Sigma}$ *over* $\mathsf{Cart}_{\mathcal{A}(\varphi)}$.

In practice, $\mathsf{bded}_\varphi = \mathsf{bcp}_\varphi$, but in principle other sound abstract transformers could be used.

Definition 14. *A* theory deduction transformer tded *is a reduction over* TS.

We extend the functions bded_φ and tded to $\mathsf{DPLL(TS)}$ as follows.

$$\mathsf{bded}_\varphi^\times(\Theta, \mathsf{te}) \mathrel{\hat{=}} (\mathsf{bded}_\varphi(\Theta), \mathsf{te}) \qquad \mathsf{tded}^\times(\Theta, \mathsf{te}) \mathrel{\hat{=}} (\Theta, \mathsf{tded}(\mathsf{te}))$$

We can now describe BCP with theory deduction as the following function, which executes the steps listed in the beginning of this section.

Definition 15. *We define* $\mathsf{deduce}_\varphi : \mathsf{DPLL(TS)} \rightarrow \mathsf{DPLL(TS)}$ *as follows.*

$$\mathsf{deduce}_\varphi \mathrel{\hat{=}} \mathsf{tprop} \circ \mathsf{tded}^\times \circ \mathsf{tinst} \circ \mathsf{bded}^\times_\varphi$$

Proposition 14. deduce_φ *is a sound overapproximation of* $mods^{T_\Sigma}_\varphi$*.*

Example 6. Consider the formula φ given as $f(x) = y \wedge x = z \wedge (f(z) \neq y \vee y = z)$. We compute deduce_φ, starting from $(\top, \top)$. Applying $\mathsf{bded}^\times_\varphi(\top, \top)$ refines the left-hand side to $\{f(x) = y, x = z\}$. Applying tinst communicates the deduction to the theory and obtains $([f(x), y][x, z], \emptyset)$ on the right. Theory deduction tded refines this to $([f(x), y, f(z)][x, z], \emptyset)$ using congruence. Finally, theory propagation tprop obtains $\{f(x) = y, x = z, f(z) = y\}$ on the left.

The deduction step in DPLL(T) computes a greatest fixed point over deduce_φ. A decision over an element Θ constructs an assignment $\Theta \cup l$, where l is a literal that occurs in neither positive nor negative phase in Θ. In abstract-interpretation terminology, this corresponds to a jump down the lattice which underapproximates the greatest fixed point and can be viewed as a dual widening operator [7].

Conflict Analysis with Theory Explanations DPLL(T) solvers are based on propositional clause learning algorithms. The power of these algorithms rests significantly in the conflict analysis step, which extracts general, sufficient conditions for unsatisfiability from specific contradictory cases. We describe conflict analysis abstractly (see [14, 8] for a lifting of conflict analysis algorithms to abstract domains). Conflict analysis computes a least fixed point over sets of elements over the underlying domain [7]: In general, there may be incomparable reasons a and b for a given deduction c, the most general conflict analysis will therefore return the set $\{a, b\}$. Indeed, conflict analyses that collect more than one conflict do exist [16].

In order to integrate theory solvers meaningfully into the analysis, they need to be able to supply explanations for deduced facts whenever theory propagation was applied. A step during conflict analysis with theory explanations can be broken down into the following substeps.

(i) *Boolean abduction:* Find Boolean conflict explanations.
(ii) *Theory justification:* Delegate explanations to the theory solver.
(iii) *Theory abduction:* Find theory explanations.
(iv) *Theory explanation:* Translate theory explanation into Boolean facts.

Recall that deduction corresponds to overapproximation of $mods^{T_\Sigma}_\varphi$. Conversely, finding explanations for deductions corresponds to underapproximation of the $ucmods^{T_\Sigma}_\varphi$ transformer.

Definition 16. *A* Boolean abduction transformer babd_φ *is an underapproximation of* $ucmods^{T_\Sigma}_\varphi$ *over the downset completion* $\mathcal{D}(\mathsf{Cart}_{\mathcal{A}(\varphi)})$*.*

Example 7. Consider $\varphi \mathrel{\hat{=}} \varphi' \wedge (x \neq y \vee r = z) \wedge (x = y \vee r \neq z)$. Assume that the element $\Theta \mathrel{\hat{=}} \{x = y, r = z\}$ leads to a contradiction. A sound abduction may obtain $\mathsf{babd}_\varphi(\{\Theta\}) = \{\{x = y\}, \{r = z\}\}$, indicating that $x = y$ and $r = z$ are both explanations for Θ, since one element in Θ suffices to deduce the other.

Theory solvers have no access to the original formula φ, but only to their internal state. Essentially, they correspond to abduction with respect to the truth-constant t.

Definition 17. *A* theory abduction transformer tabd *is a dual reduction over the downset completion* $\mathcal{D}(\mathsf{TS})$.

Example 8. Consider $\mathsf{te} = ([x, y, z], \{(x, y), (y, z)\})$, which represents a conflict. A sound abduction may return $\mathsf{tabd}(\{\mathsf{te}\}) = \{([x, y], \{(x, y)\})\ ,\ ([y, z], \{(y, z)\})\}$, highlighting two separate reasons for the conflict.

We extend the functions babd_φ and tabd to sets in $\mathcal{D}(\mathsf{DPLL}(\mathsf{TS}))$ as follows.

$$\begin{aligned}\mathsf{babd}^\times_\varphi(\Gamma) &\mathrel{\hat{=}} \{(\Theta, \mathsf{te}) \mid \exists(\Theta', \mathsf{te}) \in \Gamma.\ \Theta \in \mathsf{babd}_\varphi(\{\Theta'\})\}\\ \mathsf{tabd}^\times(\Gamma) &\mathrel{\hat{=}} \{(\Theta, \mathsf{te}) \mid \exists(\Theta, \mathsf{te}') \in \Gamma.\ \mathsf{te} \in \mathsf{tabd}(\{\mathsf{te}'\})\}\end{aligned}$$

The above transformers find reasons in their respective domains. The transformers we define next explain facts by crossing domain boundaries. When crossing from the theory abstraction to the less precise Cartesian abstraction the issue of expressibility arises, since some abstract theory facts may not have precise counterparts in the Cartesian domain. For an element $\mathsf{te} \in \mathsf{TS}$, we write *expressible*$(\mathsf{te})$ to denote the condition that te is precisely expressible in $\mathsf{Cart}_{\mathcal{A}(\varphi)}$, i.e. $\gamma_{\mathsf{TS}}(\mathsf{te}) = \gamma_{\mathsf{C}} \circ \mathsf{T2B}(\mathsf{te})$.

Definition 18. *We define the* theory justification *and* theory explanation *transformer over* $\mathcal{D}(\mathsf{DPLL}(\mathsf{TS}))$ *below.*

$$\begin{aligned}\mathsf{tjustify}(\Gamma) &\mathrel{\hat{=}} \{(\Theta, \mathsf{B2T}(\Theta') \sqcap \mathsf{te}) \mid (\Theta \sqcap \Theta', \mathsf{te}) \in \Gamma\}\\ \mathsf{texpl}(\Gamma) &\mathrel{\hat{=}} \{(\Theta \sqcap \mathsf{T2B}(\mathsf{te}), \mathsf{te}') \mid (\Theta, \mathsf{te} \sqcap \mathsf{te}') \in \Gamma \mathit{s.t.}\ \mathit{expressible}(\mathsf{te})\}\end{aligned}$$

Example 9. Consider a set of atoms $\mathcal{A}(\varphi) = \{x = y,\ y = z\}$, and an element (θ, te) with $\theta = \{x = y\}$ and $\mathsf{te} = ([x][y][z], \{(y, z)\})$. Then $\mathsf{tjustify}(\{(\theta, \mathsf{te})\})$ contains the justification $(\top, ([x, y][z], \{(y, z)\})$, and $\mathsf{texpl}(\{(\theta, \mathsf{te})\})$ contains the explanation $(\{x = y,\ \neg(y = z)\}, \top)$.

The transformer $\mathsf{tjustify}$ explains information from the Cartesian domain in terms of the theory domain. The transformer texpl does the opposite, but can only do so if a given theory domain fact can be precisely expressed in $\mathsf{Cart}_{\mathcal{A}(\varphi)}$. In both cases, the formula φ is not taken into consideration.

Proposition 15. $\mathsf{tjustify}$ *and* texpl *are dual reductions.*

We note that *conflict set generation* [3] is a combination of theory abduction tabd of the $\bot$ element, followed by theory explanation.

A step of conflict analysis with theory justification can then be modelled as a function that executes the steps outlined in the beginning of this section.

Definition 19. *We define the transformer* abduce_φ *over* $\mathcal{D}(\mathsf{DPLL}(\mathsf{TS}))$ *as:*

$$\mathsf{abduce}_\varphi \mathrel{\hat{=}} \mathsf{texpl} \circ \mathsf{tabd}^\times \circ \mathsf{tjustify} \circ \mathsf{babd}^\times_\varphi$$

$$\begin{tikzcd}
\gamma_\bot\text{-complete deduction} \arrow[d, dashed, no head] & & \bot\text{-complete reduction} \arrow[d, dashed, no head] \\
\boxed{\mathsf{BS}} \arrow[loop left, "\mathsf{bded}"] \arrow[loop right, "\mathsf{babd}"] \arrow[rr, bend left, "\mathsf{B2T}"] & \times & \boxed{\mathsf{TS}} \arrow[loop left, "\mathsf{tabd}"'] \arrow[loop right, "\mathsf{tded}"] \arrow[ll, bend left, "\mathsf{T2B}"] \\
\text{Base domain} & & \text{Theory domain}
\end{tikzcd}$$

	Model Search
Domain	$\mathsf{BS} \times \mathsf{TS}$ s.t. $\mathsf{TS} \underset{\mathsf{T2B}}{\overset{\mathsf{B2T}}{\leftrightarrows}} \mathsf{BS}$
Req. Transfomers	$\mathsf{bded} : \mathsf{BS} \to \mathsf{BS}$, $\mathsf{tded} : \mathsf{TS} \to \mathsf{TS}$ overapprox. of $mods_\varphi^{T_\Sigma}$
Theory Instantiation	$\mathsf{tinst}(\mathsf{be}, \mathsf{te}) \mathrel{\hat{=}} (\mathsf{be}, \mathsf{te} \sqcap \gamma_{\mathsf{TS}}(\mathsf{be}))$
Theory Propagation	$\mathsf{tprop}(\mathsf{be}, \mathsf{te}) \mathrel{\hat{=}} (\mathsf{be} \sqcap \alpha_{\mathsf{TS}}(\mathsf{te}), \mathsf{te})$
Deduction	$\mathsf{deduce}_\varphi \mathrel{\hat{=}} \mathsf{tprop} \circ \mathsf{tded}^\times \circ \mathsf{tinst} \circ \mathsf{bded}^\times$
Model Search	gfp over deduce_φ with dual widening over BS

	Conflict Analysis
Domain	Downset completion $\mathcal{D}(\mathsf{BS} \times \mathsf{TS})$
Req. Transfomers	babd over $\mathcal{D}(\mathsf{BS})$, tabd over $\mathcal{D}(\mathsf{TS})$ u.-approx. of $ucmods_\varphi^{T_\Sigma}$
Theory Justification	$\mathsf{tinst}(\Gamma) \mathrel{\hat{=}} \{(\Theta, \gamma_{\mathsf{TS}}(\Theta') \sqcap \mathsf{te}) \mid (\Theta \sqcap \Theta', \mathsf{te}) \in \Gamma\}$
Theory Explanation	$\mathsf{texpl}(\Gamma) \mathrel{\hat{=}} \{(\Theta \sqcap \mathsf{T2B}(\mathsf{te}), \mathsf{te}') \mid (\Theta, \mathsf{te} \sqcap \mathsf{te}') \in \Gamma$ s.t. $\mathit{expressible}(\mathsf{te})\}$
Abduction	$\mathsf{abduce}_\varphi(\Gamma) \mathrel{\hat{=}} \mathsf{texpl} \circ \mathsf{tabd}^\times \circ \mathsf{tjustify} \circ \mathsf{babd}^\times$
Conflict Analysis	lfp over abduce_φ with dual narrowing

Fig. 2. DPLL(T) as Abstraction

Proposition 16. abduce_φ *is a sound underapproximation of* $ucmods_\varphi^{T_\Sigma}$.

Conflict analysis can then be viewed to compute a least fixed point over abduce_φ, starting from a propositional conflict $\{(\bot, \mathsf{te})\}$ or theory conflict $\{(\Theta, \bot)\}$. In practice, solvers do not keep track of sets of explanations for a conflict, but will instead consider only one. Choosing specific explanations can be viewed as a dual narrowing, since it underapproximates a least fixed point [7].

6 Algebraic Extensions of DPLL(T)

In this section, we first generalise the product construction of DPLL(T) and then show empirically that the communication restrictions induced by products are sometimes unnecessary and disadvantageous.

An Abstract View of DPLL(T) The overall architecture, domains and required transformers for DPLL(T) are depicted in Figure 2. We view the product construction $\mathsf{DPLL}(\mathsf{TS})$, as a special instance of a more general construction in which the Cartesian abstraction is a parameter. Due to space constraints, we only cover splitting-based DPLL(T) formally.

Definition 20. *An* abstract DPLL(T) domain *for a* base domain BS *and* theory domain TS *is the domain* $\mathsf{ADPLL}(\mathsf{BS}, \mathsf{TS}) \mathrel{\hat{=}} \mathsf{BS} \times \mathsf{TS}$ *with Galois connections, and transformers specified as in Figure 2.*

In order to extract the algebraic essence of DPLL(T), one can view the algorithm in terms of two synergistic strategies: (i) DPLL(T) uses γ-complete deduction to obtain a precise representation of models, and then uses $\bot$-complete reduction to check emptiness; (ii) DPLL(T) uses case splits (and learning) to resolve imprecision. It is important to see that these two strategies are independent. To illustrate, consider computing the γ-complete transformer $\mathsf{BSkelModels}$ explicitly, e.g., using BDDs instead of a case split procedure.

Theorem 2. *For an abstract* DPLL(T) *domain* $\mathsf{ADPLL}(\mathsf{BS},\mathsf{TS})$ *where* bded^* *is* $\gamma_\bot$*-complete and* tded^* *is a* $\bot$*-complete reduction, it holds that* φ *is satisfiable exactly if* $\mathsf{gfp}\ \mathsf{deduce}_\varphi \neq \bot$.

This property may be hard to achieve in practice unless an expensive abstraction is chosen for BS. In this case, case analysis with splitting (or other techniques such as clause learning) can be employed. We model these algorithms abstractly as procedures that provide decompositions of elements into precise cases. For a more detailed account, consider [7, 14, 8].

Definition 21. *A* $\gamma_\bot$-precise decomposition *is a function* $\mathsf{dc} : \mathsf{BS} \to \wp(\mathsf{BS})$ *s.t. for all elements* $\mathsf{be} \in \mathsf{BS}$ *it holds that (i)* $\mathsf{dc}(\mathsf{be})$ *is finite, (ii)* $\gamma_{\mathsf{BS}}(\mathsf{be}) \subseteq \bigcup\{\gamma(\mathsf{be}') \mid \mathsf{be}' \in \mathsf{dc}(\mathsf{be})\}$ *and (iii) for any* $\mathsf{bded}' \in \mathsf{dc}(\mathsf{bded})$ *the transformer* bded^* *is* $\gamma_\bot$*-complete at* bded'.

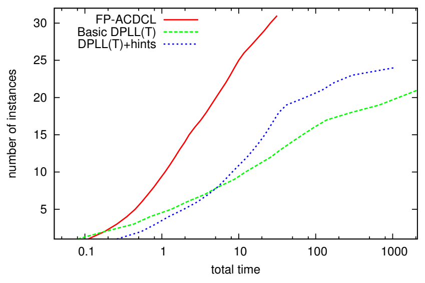
Splitting or learning-based algorithms can be viewed to generate this decomposition on demand. For an element $\mathsf{be}' \in \mathsf{BS}$, we denote by $\mathsf{deduce}_{\varphi,\mathsf{be}'}$ the function $\lambda(\mathsf{be},\mathsf{te}).\ \mathsf{deduce}_\varphi(\mathsf{be}' \sqcap \mathsf{be},\mathsf{te})$.

Theorem 3. *For an abstract* DPLL(T) *domain* $\mathsf{ADPLL}(\mathsf{BS},\mathsf{TS})$ *with* $\gamma_\bot$*-precise decomposition function* dc *and* $\bot$*-complete reduction* tded*, it holds that* φ *is satisfiable exactly if there exists a* $\mathsf{be} \in \mathsf{dc}(\top)$ *such that* $\mathsf{gfp}\ \mathsf{deduce}_{\varphi,\mathsf{be}} \neq \bot$.

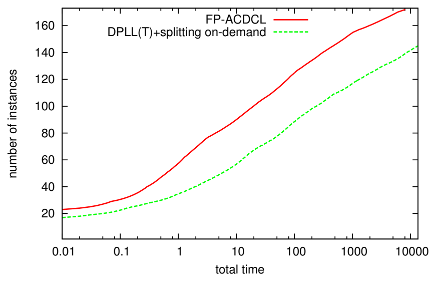
Unifying Base and Theory Reasoning An interesting consequence of the algebraic view of DPLL(T) is that we can consider architectures of the form $\mathsf{ADPLL}(\mathsf{TS},\mathsf{TS})$, which perform all steps of the algorithm directly over TS. We refer to this strategy as Abstract Conflict Driven Clause Learning (ACDCL), it is developed in detail in [8]. We present experiments in this section, based on the FP-ACDCL solver [14], an SMT solver for floating-point logic.

In DPLL(T), the vocabulary of the primary solver is limited by the structure of the formula. This can cause suboptimal performance, which is the reason why refinements of DPLL(T) introduce fresh propositions at certain points when needed. We will consider *splitting on demand* [2], which allows the introduction of new propositions during case splits, to model the effect of decision making directly in the theory.

Comparing ACDCL and DPLL(T) We present two experiments: (i) A comparison of classic DPLL(T) and ACDCL on set of hand-crafted formulae, in which the vocabulary restrictions of DPLL(T) cause enumeration behaviour. (ii) A comparison of DPLL(T) with splitting on demand and ACDCL on a set conjunctive formulae that require splitting within the theory for completeness. It is important to note that the benchmarks are specifically chosen to illustrate some limitations of DPLL(T), which can be overcome in the algebraic framework advocated in this



(a)



(b)

Fig. 3. Experimental results.

paper. To compare against classic DPLL(T), we have integrated FP-ACDCL as a black-box theory solver in the MATHSAT5 SMT solver [13].

An example of a formula (parametrised by N) used in experiment (i) is below.

$$((x = 1) \vee \ldots \vee (x = N)) \wedge ((y = 1) \vee \ldots \vee (y = N)) \wedge ((x + y < 0) \vee (x + y > 2N))$$

Classic DPLL(T) generates lemmas only in terms of the propositions in the Boolean skeleton. In FP-ACDCL, lemmas are directly inferred over disjunctions of interval constraints, independent of whether they occur in the formula or not.

The results of the comparison are given in Figure 3 (a), which plots the number of solved instances against total execution time for FP-ACDCL and DPLL(T). To boost the power of classic DPLL(T) we experimented with a variant in which FP-ACDCL provides hints to the SAT solver: At every theory conflict, we introduce a set of propositions corresponding to the theory deductions leading up to the conflict. Although this variant is a significant improvement over default DPLL(T), it still performs much worse than FP-ACDCL.

For the second set of experiments, we have used the benchmark problems from [14]. The formulae in this set are simple conjunctions of atoms, but they require a significant amount of case splits in the interval domain. The plot in Figure 3 (b) compares FP-ACDCL and splitting-on-demand. The results show that performing case splits directly in the interval domain is more effective than splitting-on-demand. When generating lemmas during conflict analysis, FP-ACDCL can use conflict generalisation [14] to improve the strength of learnt lemmas. We attribute the faster runtime of FP-ACDCL to the better quality of the resulting learnt lemmas.

References

1. B. Badban, J. van de Pol, O. Tveretina, and H. Zantema. Generalizing DPLL and satisfiability for equalities. *Inf. Comput.*, 205(8), 2007.
2. C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In *LPAR*, volume 4246 of *LNCS*. Springer, 2006.

3. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*. IOS Press, 2009.
4. S. Cotton. Natural domain SMT: a preliminary assessment. In *FORMATS*, volume 6246 of *LNCS*. Springer, 2010.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
6. P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *FoSSaCS*, volume 6604 of *LNCS*. Springer, 2011.
7. V. D'Silva, L. Haller, and D. Kroening. Satisfiability solvers are static analyzers. In *SAS*, volume 7460 of *LNCS*, pages 317–333. Springer, 2012.
8. V. D'Silva, L. Haller, and D. Kroening. Abstract Conflict Driven Clause Learning. In *POPL*, 2013. (to appear).
9. V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric bounds analysis with conflict-driven learning. In *TACAS*. Springer, 2012.
10. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*. Springer, 2006.
11. M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *JSAT*, 1(3-4), 2007.
12. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV*. Springer, 2004.
13. A. Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8, 2012.
14. L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, 2012.
15. W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL*, 2010.
16. H. Jin and F. Somenzi. Strong conflict analysis for propositional satisfiability. In *DATE*, 2006.
17. D. Jovanovic and L. de Moura. Cutting to the chase: Solving linear integer arithmetic. In *CADE*. Springer, 2011.
18. D. Jovanovic and L. de Moura. Solving non-linear arithmetic. In *IJCAR*. Springer, 2012.
19. K. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to richer logics. In *CAV*. Springer, 2009.
20. R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *CAV*, volume 3576 of *LNCS*. Springer, 2005.
21. 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, 2006.
22. A. Thakur and T. Reps. A Method for Symbolic Computation of Abstract Operations. In *CAV*, volume 7358 of *LNCS*. Springer, 2012.
23. A. Thakur and T. Reps. A generalization of Stålmarck's method. In *SAS*, volume 7460 of *LNCS*. Springer, 2012.