Deciding Floating-Point Logic with Systematic Abstraction

Leopold Haller*, Alberto Griggio†, Martin Brain*, Daniel Kroening*
*Computer Science Department, University of Oxford, Oxford, UK
first.last@cs.ox.ac.uk
†Fondazione Bruno Kessler, Trento, Italy
griggio@fbk.eu

***Abstract*—We present a bit-precise decision procedure for the theory of binary floating-point arithmetic. The core of our approach is a non-trivial generalisation of the conflict analysis algorithm used in modern SAT solvers to lattice-based abstractions. Existing complete solvers for floating-point arithmetic employ bit-vector encodings. Propositional solvers based on the Conflict Driven Clause Learning (CDCL) algorithm are then used as a back-end. We present a natural-domain SMT approach that lifts the CDCL framework to operate directly over abstractions of floating-point values. We have instantiated our method inside MATHSAT5 with the floating-point interval abstraction. The result is a sound and complete procedure for floating-point arithmetic that outperforms the state-of-the-art significantly on problems that check ranges on numerical variables. Our technique is independent of the specific abstraction and can be applied to problems beyond floating-point satisfiability checking.**

***Index Terms*—floating point, decision procedures, abstract interpretation.**

I. Introduction

Floating-point computations are used pervasively in low-level control software and embedded applications. Such programs are frequently used in areas where safety is of critical importance, such as the automotive and aerospace industry.

Floating-point numbers have a dual nature that complicates complete logical reasoning. On the one hand, they are approximate representations of real numbers, suggesting a numeric approach to their analysis. On the other hand, their discrete nature leads to "odd behaviours", which purely numeric techniques are ill-equipped to handle.

Current complete satisfiability decision procedures for constraints over floating-point numbers are based on bit-vector encodings [1]. The resulting instances are often hard for current Satisfiability Modulo Theory (SMT) solvers. On the other hand, inexpensive techniques such as floating-point interval propagation [2] can be employed to solve *some* instances very efficiently.

To illustrate this point, consider the formula $x \in [0.0, 10.0] \wedge y = x^5 \wedge y > 10^5$, over double-precision floating-point variables x and y. Interval propagation can deduce in a fraction of a second that $y \in [0.0, 100000.0]$ holds, which contradicts the final conjunct $y > 10^5$. In stark contrast, the SMT solver Z3 requires 16 minutes on a modern processor to prove unsatisfiability of a corresponding bit-vector encoding. Likewise, it is possible to construct very simple formulas that interval propagation cannot solve: Consider the floating-point formula $z = y \wedge x = y \cdot z \wedge x < 0$. Standard interval propagation cannot determine that $y \cdot z$ must be positive and fails to prove unsatisfiability. Z3 solves the problem above in less than a second.

The power of an incomplete proof technique such as interval propagation can be boosted by decomposing the proof attempt into cases. In classic DPLL(T) [3], for example, a SAT solver based on the Conflict Driven Clause Learning (CDCL) algorithm enumerates cases by assigning predicates occurring in the formula to candidate truth values. A separate theory decision procedure is then used to check whether the resulting cases are consistent.

In the above examples, classic DPLL(T) would not be able to provide a further refinement since all predicates must be true for the formulas to be satisfiable. However, further decomposition into cases is still possible if we directly enter the domain of the theory. If we assume that $y < 0$ it follows that $z < 0$, which is sufficient to show that $x > 0$. The complementary case $y \geq 0$ can be shown with similar ease. A complete procedure can be obtained in this way since i) interval propagation is complete for sufficiently small cases, e.g., the case where every variable is assigned to a singleton range and ii) there is a finite number of such cases that need to be checked.

In essence, it is possible to use the DPLL(T) framework to perform case splitting directly in the theory [4]. This requires introduction of a potentially large number of new propositions to represent theory facts and makes implementation of good learning heuristics difficult, since the propositional learning algorithm is unaware of the theory semantics associated with propositions. To handle problems such as the above, the emerging area of *natural domain* SMT procedures [5]–[9] aims at increasing the power of SMT techniques by lifting them directly to richer logics. For example, where a CDCL solver makes *decisions* that force Boolean variable to true or false, a natural domain SMT solver for linear integer arithmetic may set a variable to some specific integer value [6]. Such procedures

Supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/H017585/1, 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.

typically require custom, domain-specific decision heuristics and learning procedures.

The work presented in this paper can be seen as a systematic derivation of a learning algorithm for floating-point logic from an abstract domain. We exploit a simple insight, advocated in an earlier paper [10]: Propositional SAT solvers internally operate over a lattice-theoretic abstraction that overapproximates the space of possible solutions. Natural liftings of CDCL-style learning to richer logics can be obtained by considering a wider scope of abstractions.

In this paper, we show how the FIRST-UIP learning algorithm [11] used in CDCL solvers can be lifted to a wider range of domains. This lifting is non-trivial since it has to address the additional complexity of abstractions for domains that go beyond propositional logic. We present a new implementation of our approach for floating-point logic as part of the MATHSAT5 framework. The implementation outperforms approaches based on bit-blasting significantly on our set of benchmarks.

Contribution: The contributions of this paper are threefold: *(i)* we present a novel natural domain solver for the theory of floating-point arithmetic that significantly outperforms the state of the art; *(ii)* we introduce a lifting of the FIRST-UIP conflict analysis algorithm used in modern SAT solvers to abstractions, *(iii)* we evaluate our work on a set of benchmarks.

Outline: Section II provides a brief introduction to floating-point numbers, the theory of floating-point arithmetic and some formal background on abstract interpretation. Section III gives a high-level account of model search and conflict analysis over abstract domains. The main algorithmic contribution is presented in Section IV: A lifting of the FIRST-UIP algorithm to abstract domains. The implementation of our floating-point solver, the specific heuristics we used and experiments are discussed in Section V. An extensive survey of related work from the areas of theorem proving, abstract interpretation, and decision procedures is given in Section VI.

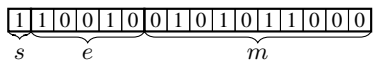
II. Floating-Point Arithmetic and Abstraction

A. Floating-Point Arithmetic

This section gives an informal introduction to the theory of floating-point arithmetic. For an exhaustive treatment, see [12] which formalises the IEEE-754 floating-point standard as an SMT theory.

Floating-point numbers are approximate representations of the reals that allow for fixed size bit-vector encoding. A floating-point number represents a real number as a triple of positive integers (s, m, e), consisting of a *sign bit* s taken from the set of Booleans $\mathbb{B} \triangleq \{0, 1\}$, a *significand* m and an *exponent* e. Its real interpretation is given by $(-1)^s \cdot m \cdot 2^e$. Note that all numbers have a sign, therefore, the real number 0 is represented both by an *unsigned zero* $+0$ and a *signed zero* -0.

A floating-point format determines the number of bits used for encoding significand and exponent. The IEEE-754 standard defines several floating-point formats and their bit-encodings. An example of an IEEE-754 `binary16` floating-point number is given below.



$= -1 \cdot 2^{18-15} \cdot 1.3359375$
$= -10.6875$

Some bit-patterns are used to encode the *special values* positive infinity $+\infty$, negative infinity $-\infty$, and NaN, which represents an invalid arithmetic result. We do not go into details regarding this encoding and simply define $\mathbb{F}$ to be the set of all floating-point numbers including the special values.

Terms in FPA are constructed from floating-point variables, constants, standard arithmetic operators and special operators such as square roots and combined multiply-accumulate operations used in signal processing. Most operations are parameterized by one of five rounding modes. The result of floating-point operations is defined to be the real result (computed with 'infinite precision') rounded to a floating-point number using the chosen rounding mode.

Formulas in FPA are Boolean combinations of predicates over floating-point terms. In addition to the standard equality predicate $=$, FPA offers a number of floating-point specific predicates including a special floating-point equality $=_{\mathbb{F}}$, and floating-point specific arithmetic inequalities $<$ and $\leq$. Since these operators approximate real comparisons they have unusual properties. For example, any comparison with the value NaN returns false, therefore $=_{\mathbb{F}}$ is not reflexive since $NaN =_{\mathbb{F}} NaN$ does not hold. On the other hand, $+0$ and -0 compare as equal since they represent the same real number.

B. Lattices and Abstractions

Following the theory of abstract interpretation [13] we define abstraction in terms of lattices and closure operators. A *complete lattice* is a partially ordered set $(P, \sqsubseteq)$ in which any subset $S \subseteq P$ has a unique least upper bound $\bigsqcup S$ and unique greatest lower bound $\bigsqcap S$. A complete lattice has a least element $\bot$ and a greatest element $\top$. A *powerset lattice* of a set Q is a complete lattice $(\wp(Q), \subseteq)$ with least upper bound $\bigcup$, and greatest lower bound $\bigcap$. A *transformer* on P is a monotone function $f : P \to P$. Transformers over P form a complete lattice under the pointwise order, $f \sqsubseteq g$ if $\forall p \in P. f(p) \sqsubseteq g(p)$. Least upper bounds and greatest lower bounds extend pointwise to the transformer lattice, e.g., $f \sqcap g = \lambda p.\ f(p) \sqcap g(p)$. We denote the least fixed point of a transformer g as $\mathsf{lfp}\, X.\ g(X)$ or $\mathsf{lfp}\, g$, and the greatest fixed point $\mathsf{gfp}\, X.\ g(X)$ or $\mathsf{gfp}\, g$. The *image* of f is the set $Img(f) \triangleq \{f(p) \mid p \in P\}$.

A *closure operator* on P is a transformer $\zeta : P \to P$ such that for all $p, q \in P$, *(i)* ζ is *extensive*, i.e., $p \sqsubseteq \zeta(p)$ and *(ii)* ζ is *idempotent*, i.e., $\zeta(p) = \zeta(\zeta(p))$. An *abstraction* of a lattice $(P, \sqsubseteq)$ is a complete sublattice $(Q, \sqsubseteq)$ with $Q \subseteq P$, such that $Q = Img(\zeta)$ for some closure operator ζ. We call P the concrete and Q the abstract domain. The closure operator ζ maps a set to its most precise abstract representation. We assume throughout this paper that $\zeta(\bot) = \bot$. An *abstract transformer* $g : Q \to Q$ is an *overapproximation* of a transformer $f : P \to P$ if $\forall q \in Q.\ f(q) \sqsubseteq g(q)$ and an *underapproximation* if $\forall q \in Q.\ g(q) \sqsubseteq f(q)$. The unique *best overapproximation* of $f : P \to P$ w.r.t. to a closure operator ζ is the function $g = \zeta \circ f \circ \zeta$. A best underapproximation does,

in general, not exist.

Example II.1 (Intervals for $\wp(\mathbb{F})$)**.** Intervals approximate sets of numbers by their closest enclosing range. In addition to the arithmetic ordering $\leq$, the IEEE-754 standard dictates a total order $\preceq$ over all floating-point values, including special values such as NaN. The interval abstraction is defined by a closure operator $\zeta : \wp(\mathbb{F}) \to \wp(\mathbb{F})$ where $\zeta(S) \mathrel{\hat{=}} \{v \in \mathbb{F} \mid \min_{\preceq}(S) \preceq v \preceq \max_{\preceq}(S)\}$.

C. Logic and Abstraction

In this section, we summarise our basic framework for model-theoretic approximations of logical formulas using abstraction (see [10] for more details) and show how it applies to FPA. Let Forms be the set of *formulas*, $\mathit{Structs}$ be a set of semantic *structures*. The semantics of a logic are given as an interpretation function $[\![\cdot]\!]. : (\mathit{Forms} \times \mathit{Structs}) \to \mathbb{B}$. An element $\sigma \in \mathit{Structs}$ is a *model* of a formula $\varphi \in \mathit{Forms}$ if $[\![\varphi]\!]_\sigma = 1$ and a *countermodel* otherwise. A formula is *satisfiable* if it has a model and *unsatisfiable* otherwise.

Semantic structures in FPA are given by *floating-point assignments*, defined as $\mathit{FloatAsg} \mathrel{\hat{=}} \mathit{Vars} \to \mathbb{F}$, where Vars is a finite set of first-order variables.

For a formula φ, we define two transformers on the powerset lattice $\wp(\mathit{Structs})$.

Definition II.1. The *model transformer* mods_φ and the *conflict transformer* confs_φ are defined as follows.

$$\begin{aligned} \mathit{mods}_\varphi(S) &\mathrel{\hat{=}} \{\sigma \in \mathit{Structs} \mid \sigma \in S \wedge [\![\varphi]\!]_\sigma = 1\} \\ \mathit{confs}_\varphi(S) &\mathrel{\hat{=}} \{\sigma \in \mathit{Structs} \mid \sigma \in S \vee [\![\varphi]\!]_\sigma = 0\} \end{aligned}$$

The model transformer maps a set of structures to its smallest subset that contains the same models. The conflict transformer (also referred to as the *universal countermodel transformer* in [10]) maps a set of structures to its largest superset that contains the same models. The model transformer can be used to refine an overapproximation of a set of models, and the conflict transformer to generalise an underapproximate set of countermodels.

Satisfiability can be expressed in terms of these operators. Note that mods_φ and confs_φ are idempotent, therefore $\mathit{mods}_\varphi(\mathit{Structs}) = \mathsf{gfp}\ \mathit{mods}_\varphi$ and $\mathit{confs}_\varphi(\emptyset) = \mathsf{lfp}\ \mathit{confs}_\varphi$.

Theorem 1. *The following statements hold.*

1) $\mathsf{gfp}\ \mathit{mods}_\varphi = \emptyset$ *exactly if* φ *is unsatisfiable.*
2) $\mathsf{lfp}\ \mathit{confs}_\varphi = \mathit{Structs}$ *exactly if* φ *is unsatisfiable.*

We can compute these fixed points abstractly to perform incomplete satisfiability checks. Propositional solvers use the partial assignment abstraction [10]. For example a partial assignment $\langle p{:}\mathsf{true}, q{:}\mathsf{false}\rangle$ abstractly represents the set of assignments σ from propositions to truth values, where $\sigma(p) = \mathsf{true}$ and $\sigma(q) = \mathsf{false}$ and all other propositions may be mapped to either truth value.

In this paper, we use the interval abstraction. Recall that IEEE-754 requires a total ordering $\preceq$. We use it to define an interval abstraction for the powerset lattice $\wp(\mathit{FloatAsg})$. An *interval assignment*, written $\langle x_1 : [l_1, u_1], \ldots, x_k : [l_k, u_k]\rangle$, is a set of floating-point assignments $\{\sigma \mid \forall i.\ l_i \preceq \sigma(x_i) \preceq u_i\}$. We denote the set of all interval assignments by $\mathbb{I}_\mathbb{F}$, which forms a complete lattice under the set order $\subseteq$. The closure operator defining the interval abstraction is given as $\zeta(S) \mathrel{\hat{=}} \langle x_1 : [l_1, u_1], \ldots, x_k : [l_k, u_k]\rangle$ where $l_i = \min_{\preceq}\{\sigma(x_i) | \sigma \in S\}$ and $u_i = \max_{\preceq}\{\sigma(x_i) | \sigma \in S\}$.

For example, let $f = \{x \mapsto 4.2, y \mapsto 2.3\}$ and $g = \{x \mapsto 1.8, y \mapsto 10.5\}$, then applying ζ yields $\zeta(\{f, g\}) = \langle x : [1.8, 4.2], y : [2.3, 10.5]\rangle$.

We can use the interval abstraction to approximate the fixed points of Theorem 1.

Theorem 2. *Let* amods_φ *be an overapproximation of* mods_φ *and let* aconfs_φ *be an underapproximation of* confs_φ*.*

1) *If* $\mathsf{gfp}\ \mathit{amods}_\varphi = \emptyset$ *then* φ *is unsatisfiable.*
2) *If* $\mathsf{lfp}\ \mathit{aconfs}_\varphi = \mathit{Structs}$ *then* φ *is unsatisfiable.*

One view is that overapproximations of mods_φ perform deduction by establishing necessary properties of models, while underapproximations of confs_φ perform abduction by finding sufficient conditions (or explanations) for conflicts.

III. Lifting CDCL to Abstractions

CDCL consists of two interacting phases, model search and conflict analysis. Model search aims to find satisfying assignments for the formula. This process may fail and encounter a *conflicting* partial assignment, that is, a partial assignment that contains only countermodels. Conflict analysis extracts a general reason which is used to derive a new lemma over the search space in the form of a clause. In this section we show how model search and conflict analysis can be lifted to abstractions to yield an Abstract CDCL (ACDCL) algorithm. We assume familiarity with CDCL [14].

A. Abstract Model Search

Model search alternates two steps, *deductions* and *decisions*, which refine a given partial assignment. We model these steps as transformers on abstract lattices.

1) Deduction: Deduction rules are overapproximations of the model transformer. Modern CDCL solvers use efficient but imprecise overapproximations, such as the unit rule.

Definition III.1. A *deduction rule* for an abstraction Q of $\wp(\mathit{Structs})$ and formula $\varphi \in \mathit{Forms}$ is an overapproximation $\mathit{ded} : Q \to Q$ of mods_φ.

Example III.1. Consider the formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ with $\varphi_1 \mathrel{\hat{=}} (5 \leq x \leq 10)$, $\varphi_2 \mathrel{\hat{=}} (x = y)$ and $\varphi_3 \mathrel{\hat{=}} (y = z)$. We define a deduction rule $\mathit{ded}(S) \mathrel{\hat{=}} \bigsqcap_{i\in\{1,2,3\}} \zeta(\mathit{mods}_{\varphi_i}(\zeta(S)))$ by computing the best overapproximations of $\mathit{mods}_{\varphi_i}$ for $i \in \{1, 2, 3\}$ and intersecting the result. We now compute the greatest fixed point $\mathsf{gfp}\ \mathit{ded}$ which is the analogue of performing Boolean constraint propagation in propositional solvers, where $F_0 = \top$ and $F_i = \mathit{ded}(F_{i-1})$.

$$\begin{aligned} F_0 &= \mathit{FloatAsg} & F_1 &= \langle x : [5.0, 10.0]\rangle \\ F_2 &= F_1 \sqcap \langle y : [5.0, 10.0]\rangle & F_3 &= F_2 \sqcap \langle z : [5.0, 10.0]\rangle \end{aligned}$$

The resulting element $F_3 = \langle x : [5.0, 10.0], y : [5.0, 10.0], z : [5.0, 10.0]\rangle$ imprecisely overapproximates the set of models.

2) Decisions: Once no new information can be deduced, a CDCL solver makes a decision by restricting the value of a proposition p to a truth value v. In lattice theoretic terms, this can be viewed as computation of the greatest lower bound $\pi \sqcap \langle p : v \rangle$, where π is the original partial assignment.

In terms of the abstraction, an important property of singleton partial assignments is that their complement is precisely expressible as a partial assignment. We generalise:

Definition III.2. Let Q be an abstraction of $\wp(\mathit{Structs})$. The set of *complementable elements* $\mathit{Comp}(Q) \subseteq Q$ is the set of all $q \in Q$ such that $\overline{q} = \mathit{Structs} \setminus q$ is also in Q. A transformer $f : Q \to Q$ is *complementable* if $\mathit{Img}(f) \subseteq \mathit{Comp}(Q)$.

Example III.2 (Complementable elements)**.** Complementable interval assignments map a variable to a half-open interval. The element $\sigma = \langle x : [+0, \max_{\preceq}(\mathbb{F})] \rangle$ is complementable, the elements $\langle x{:}[1, 2] \rangle$ and $\langle x{:}[1, \max_{\preceq}(\mathbb{F})], y{:}[4.2, \max_{\preceq}(\mathbb{F})] \rangle$ are not.

For convenience, we write complementable elements $\langle x{:}[c, \max_{\preceq}(\mathbb{F})] \rangle$ and $\langle x{:}[\min_{\preceq}(\mathbb{F}), c] \rangle$ as $\langle x \succeq c \rangle$ and $\langle x \preceq c \rangle$, respectively.

Modern CDCL solvers implement decision heuristics that use statistical information generated from the execution history of the procedure. Since we do not intend to give a fully stateful account of CDCL here, we abstractly formalise this idea by defining $\mathbb{H}$ to be a set of *execution histories*.

Definition III.3. A *decision heuristic* for an abstraction Q of $\wp(\mathit{Structs})$ and $\varphi \in \mathit{Forms}$ is a function $\mathit{decide} : \mathbb{H} \to Q \to Q$ s.t. for all $h \in \mathbb{H}, q \in Q$, $\mathit{decide}(h)(q)$ is a complementable element and $q \sqcap d = q$ implies that q is a set of models of φ.

B. Abstract Conflict Analysis

Model search iterates deduction and decisions until a conflicting element a is encountered, that is, an element that does not represent any models. The aim of conflict analysis is to obtain a more general element $a' \supseteq a$ that is still conflicting. Conflict analysis can be viewed as an instance of abductive reasoning, since the goal is to find a general reason or explanation for a given deduction.

1) Abduction: A conflict analysis procedure computes a *propositional abduction rule*, which generalises explanations for a deduction π over a formula φ. We model this as a transformer $\mathit{abd}_{\varphi,\pi} : \mathit{PartAsg} \to \mathit{PartAsg}$ such that for any model $\sigma \in \mathit{abd}_{\varphi,\pi}(\pi')$ of φ, σ is in π' or in π. In other words, the transformer may only introduce models that are in π. We generalise:

Definition III.4. An *abduction rule* for an abstraction Q of $\wp(\mathit{Structs})$, an element $q \in Q$ and a formula $\varphi \in \mathit{Forms}$ is an extensive underapproximation $\mathit{abd}_{\varphi,q} : Q \to Q$ of the transformer $\lambda x.\ \mathit{confs}_{\varphi}(x) \cup q$.

Essentially, one could simply work with underapproximations of confs_{φ}. The slight variation presented above allows to find explanations not only for conflicts, but also for specific deductions q.

2) Choice: Note that in contrast to deduction rules, which are overapproximations, there is no single best underapproximate abduction rule. In general, multiple maximally general abduction rules of incomparable generality may exist.

Example III.3. Let $\varphi = \ldots \wedge (p \vee q) \wedge (p \vee \neg q)$ be a propositional CNF formula, and assume that the partial assignment $\pi = \langle p : 0, q : 1 \rangle$ is conflicting with φ. The presence of either assignment to p or q is sufficient to deduce the other. We can build two incomparable abduction transformers abd^1 and abd^2 with $\mathit{abd}^1_{\varphi,\bot}(\pi) = \langle p{:}0 \rangle$ and $\mathit{abd}^2_{\varphi,\bot}(\pi) = \langle q{:}1 \rangle$.

Example III.4. Let $\varphi = x + y \leq 10.0$ be an FPA formula. The interval assignment $\sigma = \langle x < 10.0, y \preceq 10.0 \rangle$ is conflicting w.r.t. φ, since $x + y$ is at least 20.0. We can build two incomparable abduction transformers, abd^1 and abd^2 with $\mathit{abd}^1_{\varphi,\bot}(\sigma) = \langle x \preceq 10.0, y \preceq 0.0 \rangle$ and $\mathit{abd}^2_{\varphi,\bot}(\sigma) = \langle x \preceq 1.0, y \preceq 9.0 \rangle$.

The abduction rule used in propositional CDCL solvers is computed using a graph-based algorithm which will be discussed in more detail in the next section. The absence of a best abduction operator is reflected by the possibility of extracting various incomparable partial assignments from a single graph. Among these, one is heuristically chosen. We formalise this heuristic choice as a function that takes as argument an execution history and returns an abduction rule.

Definition III.5. A *choice heuristic* for an abstraction Q of $\wp(\mathit{Structs})$, $q \in Q$ and $\varphi \in \mathit{Forms}$ is a function $\mathit{choose}_{q,\varphi} : \mathbb{H} \to Q \to Q$ s.t. for all $h \in \mathbb{H}$, $\mathit{choose}_{\varphi,q}(h)$ is an abduction rule for q and φ.

IV. Learning in Abstract Implication Graphs

Effective learning is essential for the performance of CDCL. Learning algorithms in CDCL solvers operate over an implication graph, a data structure that records decisions and the result of deductions. We present a generalisation to *abstract* implication graphs. There are various aspects of the CDCL framework that we do not discuss here, such as restarts, backjumps and learning of asserting clauses. These can also be lifted from the propositional case in a relatively straightforward way.

A. Abstract Trails from Complementable Decompositions

Partial assignments and intervals share an important property regarding the decomposition of lattice elements.

Example IV.1. Let $\pi = \langle x{:}1, y{:}0, z{:}1 \rangle$ be a partial assignment. The element π is not complementable, but it can be decomposed into $\pi = \langle x{:}1 \rangle \sqcap \langle y{:}0 \rangle \sqcap \langle z : l \rangle$. Each element of the above decomposition is complementable.

Now let $\sigma = \langle x : [0.5, 2.2], y : [0.5, \max_{\preceq}(\mathbb{F})] \rangle$ be an interval assignment. Analogous to the previous case, the complement of σ is not an interval assignment, but σ can be decomposed into complementable elements as $\langle x \succeq 0.5 \rangle \sqcap \langle x \preceq 2.2 \rangle \sqcap \langle y \succeq 0.5 \rangle$.

Definition IV.1. An abstraction Q of $\wp(\mathit{Structs})$ has *complementable decompositions* if for every element q of Q there is a

finite set $S \subseteq Comp(Q)$ of complementable elements such that $q = \bigsqcap S$.

As illustrated above, both partial assignments and interval assignments admit complementable decompositions. We assume the existence of a decomposition function $\mathsf{decomp} : Q \to \wp(Comp(Q))$. Implication graph construction necessitates a decompositon of deduction rules into complementable transformers.

Example IV.2. Let ded be the best deduction rule over interval assignments for the predicate $-x = y$, and let $\sigma = \langle x : [5.0, 10.0]\rangle$. It holds that $ded(\sigma) = \sigma \sqcap \langle y : [-10.0, -5.0]\rangle$. We can decompose ded into a set of complementable rules $Ded = \{ded_x^l, ded_x^u, ded_y^l, ded_y^u\}$ s.t. $\bigsqcap Ded = ded$, and each of the elements of Ded infers a lower or an upper bound on x or y: $ded_x^l(\sigma) = \langle x \succeq 5.0\rangle$, $ded_x^u(\sigma) = \langle x \preceq 10.0\rangle$, $ded_y^l(\sigma) = \langle y \succeq -10.0\rangle$ and $ded_y^u(\sigma) = \langle y \preceq -5.0\rangle$.

Abstract Trail: CDCL solvers record decisions and deductions in a stack-based data structure called *trail*, which records variable assignments due to decisions and deductions. Deductions are associated with the clause used to derive them.

An *abstract trail* is a finite sequence of complementable elements in $Comp(Q)$. We denote the i-th element of a trail tr by tr_i, the concatenation of two sequences tr, tr' by $tr \cdot tr'$ and the subsequence $tr_i \dots tr_j$ by $tr_{i:j}$. In Algorithm 1, we give a generic model search procedure that extends an abstract trail tr and maps trail indices to reasons in a map $reasons$. The procedure can be instantiated over any abstraction Q with complementable decompositions and a decomposition Ded of the deduction rule into complementable rules.

```
modelSearch(tr, reasons, Ded)
    loop
        repeat
            forall the ded ∈ Ded do
                q ← ded(⨆ tr);
                if (⨆ tr) ⊓ q ⊏ ⨆ tr then
                    tr ← tr · q;
                    reasons[ |tr| ] ← ded;
                end
                if q = ⊥ then return (tr, reasons)
            end
        until tr unchanged;
        q ← decide(getHistory())(⨆ tr);
        if q ⋤ ⨆ tr then return SAT ;
        tr ← tr · q;
```

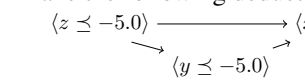
Algorithm 1: Model search with Abstract Trail

The current abstract element is represented by the greatest lower bound $\bigsqcap tr$. Deduction iterates over all deduction rules $ded \in D$, and appends a new element to the abstract trail if applying ded refines $\bigsqcap tr$. If a conflict is deduced, the procedure returns the trail and the reason map. This process is iterated until no new deductions can be made, at which point a decision is attempted. If the current element cannot be refined further, SAT is returned, otherwise the procedure appends the decision to the trail and reenters the deduction phase.

B. FIRST-UIP *in Abstract Conflict Graphs*

In propositional CDCL, a trail implicitly encodes a graph structure that records dependencies between deductions on the trail. The edges are represented implicitly by the clauses associated with each element. The FIRST-UIP algorithm [15] is a popular strategy for learning: it is a strategy to choose a set of nodes in this graph called a *cut* that suffices to produce a conflict. We now give a generalisation of FIRST-UIP to abstractions. Naively lifting the algorithm is insufficient to learn good reasons in the interval abstraction as the following example will illustrate.

Example IV.3. Consider the FPA formula $z = y \wedge x = y \cdot z \wedge x < 0$ and the interval assignment $\sigma = \langle z \preceq -5.0\rangle$. Starting from σ, we can make the following deductions.



Arrows indicate sufficient conditions for deduction, e.g., $\langle x \succeq 25.0\rangle$ can be deduced from the conjunction of $\langle z \preceq -5.0\rangle$ and $\langle y \preceq -5.0\rangle$. The last deduction $\langle x \succeq 25.0\rangle$ conflicts with the constraint $x < 0$. A classic conflict cutting algorithm may analyse the above graph to conclude that $\pi = \langle z \preceq -5.0\rangle$ is the reason for the conflict. It is easy to see though that there is a much more general reason: The conflict can be deduced in this way whenever z is negative.

```
analyse(tr, reasons)
    i ← |tr|; m ← {1 ↦ ⊤, ..., (i − 1) ↦ ⊤, i ↦ ⊥};
    loop
        q ← generalise(⨆ tr_{1:i}, reasons[i], m[i]);
        updateMarking(q, tr, m);
        m[i] ← ⊤; i ← i − 1;
        if open(tr, m) = 1 then
            return ⨆_{1≤i≤|tr|} m[i];
        end

generalise(q, d, r)
    repeat
        abd ← choose_{d,r}(getHistory());
        q ← abd(q);
    until q unchanged;
    return q;
updateMarking(q, tr, m)
    Π ← decomp(q);
    forall the c in Π do
        r ← smallest index r' s.t. tr_{r'} ⊑ c;
        m[r] ← m[r] ⊓ c;
    end
```

Algorithm 2: Abstract FIRST-UIP

Abstract FIRST-UIP breaks down the global abduction task of conflict analysis by finding generalised explanations for single deduction results. We associate with each $ded \in Ded$ a separate choice function $choose_{ded,q}$, which maps an execution history h to an abductive transformer for inferring q.

The procedure is presented in Algorithm 2. It takes as input a conflicting trail tr with final element $\bot$ and a mapping from indices i to the deduction rule used to derive an element tr_i. The main data structure is a *marking* m which maps trail indices to elements of $Comp(Q)$. Essentially, m maps each element of

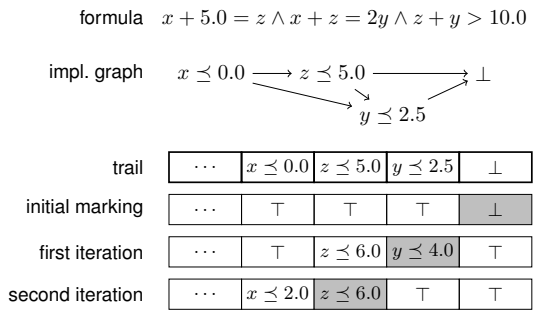


Fig. 1. Markings in Abstract FIRST-UIP

the trail tr to a generalisation that is still sufficient to produce a conflict.

Initially, m maps only the final, conflicting element to $\bot$ and everything else to $\top$. The procedure steps backwards through the trail. A call to a function `generalise(q,d,r)` finds a generalisation of $q \in Q$ such that the current trail marking r can still be deduced. This is done by computing a fixed point using heuristic choice over abductive transformers. The generalised deduction reason is decomposed into its complementables, and for each element c of the decomposition, the earliest occurrence of a stronger element on the trail is marked with c. Finally, the current marking is removed and the algorithm proceeds.

An example execution of the algorithm is illustrated in Figure 1. There, an implication graph and corresponding trail is shown which records consequences of a decision $x \preceq 0.0$. Similar to propositional CDCL, no explicit graph is constructed. Instead, the algorithm implicitly explores the graph via markings, which overapproximate the trail pointwise and encode sufficient conditions for unsatisfiability. The first iteration of the algorithm determines via abduction that $\bot$ can be ensured whenever $z \preceq 6.0$ and $y \preceq 4.0$ are the case. The second iteration finds that $y \preceq 4.0$ can be dropped from the reason if $x \preceq 2.0$ holds in addition to $z \preceq 6.0$.

It is an invariant during the run of the procedure that the greatest lower bound over all markings is sufficient to ensure a conflict. Hence the procedure could essentially terminate during any iteration and yield a sound global abduction result. We use the usual FIRST-UIP termination criterion and return once the number of open paths $\mathtt{open}(tr, m)$ reaches 1. This number is defined as the number of indices j greater or equal to the index of the most recent decision, such that $m[j] \neq \top$.

C. Abstract Clause Learning

Propositional solvers learn new clauses that express the negation of the conflict analysis result. The new clauses open up further possibilities for deduction using the unit rule. The unit rule states that for a clause $l_1 \vee \ldots \vee l_k$, if l_1 to l_{k-1} are contradicted by the current partial assignment, then the partial assignment can be refined to make l_k evaluate to true.

We model learning directly as learning of a new deduction rule, rather than learning a formula in the logic. A lattice-theoretic generalisation of the unit rule is given below. Note that we define the rule directly in terms of the conflicting element, rather than its negation.

Definition IV.2. For an abstraction P of $\wp(Structs)$ with complementable decompositions, let $c \in P$ be an element that contains no models of φ. The *abstract unit rule* $Unit_c : P \to P$ is defined as follows.

$$Unit_c(p) \mathrel{\hat{=}} \begin{cases} \bot & \text{if } p \sqsubseteq c \\ \overline{r} & \text{otherwise, if } r \in \mathsf{decomp}(c) \text{ and} \\ & \forall r' \in \mathsf{decomp}(c) \setminus \{r\}.\ p \sqsubseteq r' \\ \top & \text{otherwise} \end{cases}$$

Example IV.4. Let $c = \langle x{:}[0.0, 10.0], y \preceq 3.2\rangle$ be a conflicting element of φ. Let $p = \langle x{:}[3.0, 4.0], y{:}[1.0, 1.0]\rangle$, then $Unit_c(p) = \bot$, since $p \sqsubseteq c$. Let $p' = \langle x{:}[3.0, 4.0]\rangle$, then $Unit_c(p') = \langle y \succ 3.2\rangle$, since $p' \sqsubseteq \langle x \succeq 0.0\rangle$ and $p' \sqsubseteq \langle x \preceq 10.0\rangle$.

The unit rule $Unit_c$ for a conflicting element c soundly overapproximates the model transformer. Furthermore, it is complementable; we can perform learning by adding $Unit_c$ to Ded.

V. Implementation and Experiments

We have implemented our approach over floating-point intervals inside the MATHSAT5 SMT solver [16]. We call our prototype tool FP-ACDCL. The implementation uses the MATHSAT5 infrastructure, but is currently independent of its DPLL(T) framework. The implementation provides a generic, abstract CDCL framework with FIRST-UIP learning. The overall architecture is shown in Figure 2. An instantiation requires abstraction-specific implementations of the components described earlier, including deduction, decision making, abduction and heuristic choice. We first elaborate on those aspects of the implementation and then report experimental results.

A. Abstract CDCL for Floating-Point Intervals

1) Deductions: We implement the deduction rule ded using standard Interval Constraint Propagation (ICP) techniques for floating-point numbers, defined e.g., in [2], [17]. The implementation operates on CNF formulae over floating-point predicates.

Propagation is performed using an occurrence-list approach, which associates with each variable a list of the FPA clauses in which the variable occurs. Learnt clauses (corresponding

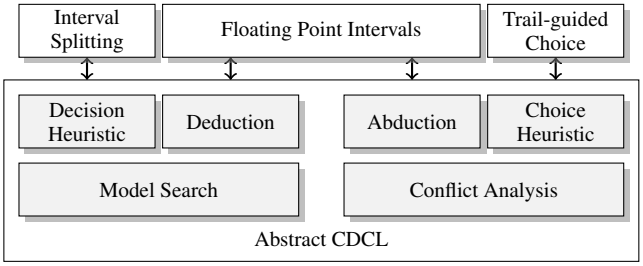


Fig. 2. FP-ACDCL Solver Architecture

to new unit rules) are stored as vectors of complementable elements and are propagated in a similar way. When a deduction is made, we scan the list of affected clauses to check for new deductions to be added to the trail. This is done by applying ICP projection functions to the floating-point predicates in a way that combines purely propositional with theory-specific reasoning. A predicate is conflicting if some variable is assigned the empty interval during ICP. If all predicates of a clause are contradicting, then we have found a conflict with the current interval assignment and ded returns $\bot$. If all but one predicate in a clause are conflicting, then the result of applying ICP to the remaining predicate is the deduction result. In this case, ded returns a list containing one complementable element $\langle x \succeq b\rangle$ (or $\langle x \preceq b\rangle$) for each new bound inferred.

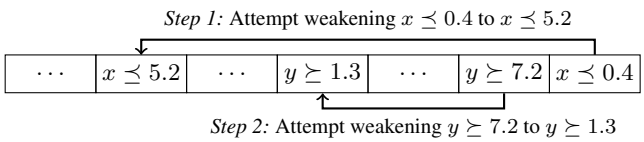
2) Decisions: FP-ACDCL performs decisions by adding to the trail one complementable element $\langle x \succeq b\rangle$ or $\langle x \preceq b\rangle$ that does not contradict the previous value of x. Clearly, there are many possible choices for (i) how to select the variable x, (ii) how to select the bound b, and (iii) how to choose between $\langle x \succeq b\rangle$ and $\langle x \preceq b\rangle$.

In propositional CDCL, each variable can be assigned at most once. In our lifting, a variable can be assigned multiple times with increasingly precise bounds. We have found some level of *fairness* to be critical for performance. Decisions should be balanced across different variables and upper and lower bounds. A strategy that proceeds in a "depth-first" manner, in which the same variable is refined using decisions until it has a singleton value, shows inferior performance compared to a "breadth-first" exploration, in which intervals of all the variables are restricted uniformly. We interpret this finding as indication that the value of abstraction lies in the fact that the search can be guided effectively using general, high-level reasoning, before considering very specific cases.

FP-ACDCL currently performs decisions as follows: (i) variables are statically ordered, and the selection on which variable x to branch is cyclic across this order; (ii) the bound b is chosen to be an approximation of the arithmetic average between the current bounds l and u on x; note that the arithmetic average is different from the median, since floating-point values are unevenly distributed; (iii) the choice between $\langle x \succeq b\rangle$ and $\langle x \preceq b\rangle$ is random. Considering the advances in heuristics for propositional SAT, there is likely a lot of room for enhancing this. In particular, the integration of fairness considerations with activity-based heuristics typically used in modern CDCL solvers could lead to similar performance improvements. This is part of ongoing and future work.

3) Generalised Explanations for Conflict Analysis: In abduction, a trade-off must be made between finding reasons quickly and finding very general reasons. We perform abduction that relaxes bounds iteratively. As mentioned earlier, there may be many incomparable relaxations. Our experiments suggest that the precise way in which bounds are relaxed is extremely important for performance. Fairness considerations similar to those mentioned for the decision heuristic need to be taken into account. However, there is an additional, important criterion. Learnt lemmas are used to drive backjumping. It is therefore preferable to learn deduction rules that allow for backjumping higher in the trail. This will lead to propagations that are affected by a smaller number of decisions, and thus will hold for a larger portion of the search space.

Our choice heuristic, called *trail-guided choice*, is abstraction-independent, and is both fair and aims to increase backjump potential. In the first step, we remove all bounds over variables from the initial reason q which are irrelevant to the deduction. Then we step backwards through the trail and attempt to weaken the current element q using trail elements. The process is illustrated below.



When an element tr_j is encountered such that tr_j is used in q (that is, $q \sqsubseteq tr_j$), we attempt to weaken q by replacing the bound tr_j with the most recent trail element more general than tr_j. If no such element exists, we attempt removing the relevant bound altogether. We check whether the weakened q is still sufficiently strong to deduce d. If not, we undo the weakening, and do not consider any further weakenings with elements more general than tr_j. After this, we repeat the process for element tr_{j-1}. The algorithm terminates once no further generalisations are possible.

Since we step backwards in order of deduction, we heuristically increase the potential for backjumps: The procedure never weakens a bound that was introduced early during model search at the expense of having to uphold a bound that is ensured only at a deep level of the search.

We have experimented with stronger but computationally more expensive generalisation techniques such as finding maximal bounds for deductions by search over floating-point values. Our experiments indicate that the cheaper technique described above is more effective overall. We see two main avenues for improvement: First, for many deductions it is possible to implement good or optimal abduction transformers effectively without search. Second, we expect that dynamic heuristics that take into account statistical information may guide conflict analysis towards useful clauses.

B. Experimental Evaluation

We have evaluated our prototype FP-ACDCL tool over a set of more than 200 benchmark formulas, both satisfiable and unsatisfiable. The formulas have been generated from problems that check (i) ranges on numerical variables and expressions, (ii) error bounds on some numerical computations using different orders of evaluation of subexpressions, and (iii) feasibility of systems of inequalities over bounded floating-point variables. The first two sets originate from verification problems on some C programs performing numerical computations, whereas the instances in the third set are randomly generated. We make our benchmarks and the FP-ACDCL tool available for experimentation by other researchers at http://www.cprover.

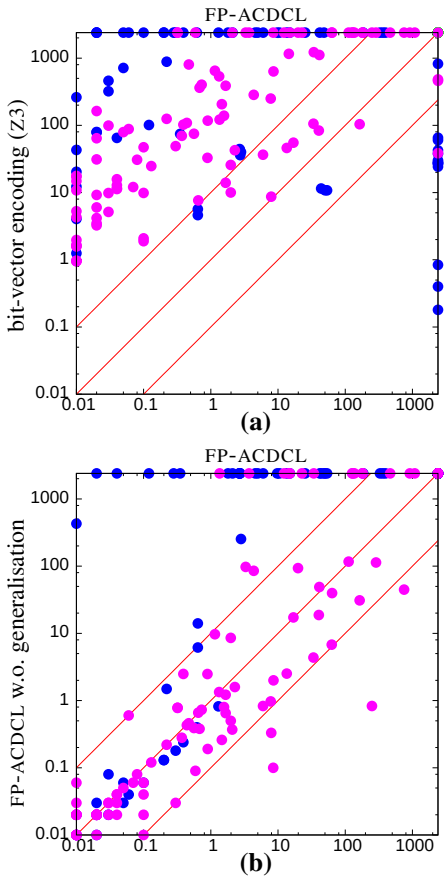


Fig. 3. Comparison of FP-ACDCL against Z3 with bit-vector encoding (a); effects of generalisations in conflict analysis (b). Darker colour indicates unsatisfiability. Points on the borders indicate timeouts (1200 s).

org/fmcad2012/. All results have been obtained on an Intel Xeon machine with 2.6 GHz and 16 GB of memory running Linux, with a time limit of 1200 seconds.

We have performed two different sets of experiments. In the first, we have compared FP-ACDCL with the current state-of-the-art procedures for floating-point arithmetic, based on encoding into bit-vectors. We have generated bit-vector encodings of all the benchmark instances in our set using MATHSAT5 and solved them with the Z3 SMT solver [18], which was the winner of the main bit-vector division in the SMT-COMP 2011 competition. The results of this comparison are reported in Figure 3(a). FP-ACDCL over FPA significantly outperforms Z3 over corresponding bit-vector encodings on most of the instances, often by several orders of magnitude. More specifically, FP-ACDCL could solve 35 benchmarks more than Z3, with an overall total speedup of more than 25x (for the subset of benchmarks that both tools could solve).[1] There are some instances that turn out to be relatively easy for Z3, but cannot be solved by our tool. This is not surprising, since there are simple instances that are not amenable to analysis with ICP, even with the addition of decision-making and learning.[2] To handle such cases, our framework can be instantiated with abstract domains or combinations of domains [13] that are better suited to the problems under analysis.

The second set of experiments is aimed at evaluating the impact of our novel generalisation technique. In order to do this, we have run FP-ACDCL with generalisation of deductions turned off, and compared it with the default FP-ACDCL. Essentially, FP-ACDCL without generalisation corresponds to a naive lifting of the conflict analysis algorithm. The results are summarised in Figure 3(b). From the plot, we can clearly see that generalisation is crucial for the performance of FP-ACDCL: without it, the tool times out in 44 more cases, whereas there is no instance that can be solved only without generalisation. However, there are a number of instances for which performance degrades when using generalisations, sometimes significantly. This can be explained by observing that (i) generalisations come at a runtime cost, which can sometimes induce a non-negligible overhead; (ii) the performance degradation occurs on satisfiable instances (shown in a lighter colour in the plots), for which it is known that the behaviour of CDCL-based approaches is typically unstable (even in the propositional case).

VI. A Survey of Related Work

We separately survey work in three related branches of research: 1) the analysis of floating-point computations, 2) lifting existing decision procedure architectures to richer problem domains and 3) automatic and intelligent precision refinement of abstract analyses.

A. Reasoning about Floating-Point Numbers

This section briefly surveys work in interactive theorem proving, abstract interpretation and decision procedures that target floating-point problems. For a discussion of the special difficulties that arise in this area, see [19].

Theorem Proving: Various floating-point axiomatisations and libraries for interactive theorem provers exist [20]–[23]. Theorem provers have been applied extensively to proving properties over floating-point algorithms or hardware [24]–[31]. While theorem proving approaches have the potential to be sound and complete, they require substantial manual work, although sophisticated (but incomplete) strategies exist to automate substeps of the proof, e.g., [32]. A preliminary attempt to integrate such techniques with SMT solvers has recently been proposed in [33].

Abstract Interpretation: Analysis of floating-point computations has also been extensively studied in abstract interpretation. An approach to specifying floating-point properties over programs was proposed in [34]. A number of general purpose abstract domains have been constructed for the analysis of floating-point programs [35]–[40]. In addition, specialised approaches exist which target specific problem domains such

[1] FP-ACDCL timed out in 28 instances, whereas Z3 ran out of time or memory in 63 cases. On the subset of benchmarks solved by both tools, the total run time was of 585 seconds for FP-ACDCL, and of 15973 seconds for Z3.

[2] A simple example of this is the formula $x = y \wedge x \neq y$, which requires an abstraction that can express relationships between variables. Intervals are insufficient to efficiently solve this problem.

as numerical filters [41], [42]. The approaches discussed so far mainly aim at establishing the result of a floating-point computation. An orthogonal line of research is to analyse the deviation of a floating-point computation from its real counterpart by studying the propagation of rounding errors [43], [44]. Case studies for this approach are given in [45], [46]. Abstract interpretation techniques provide a soundness guarantee, but may yield imprecise results.

Decision Procedures: In the area of decision procedures, study of floating-point problems is relatively scarce. Work in constraint programming [47] shows how approximation with real numbers can be used to soundly restrict the scope of floating-point values. In [17], a symbolic execution approach for floating-point problems is presented, which combines interval propagation with explicit search for satisfiable floating-point assignments. An SMTLIB theory of FPA was presented in [12]. Recent decision procedures for floating-point logic are based on propositional encodings of floating-point constraints. Examples of this approach are implemented in MATHSAT5 [16], CBMC [48] and Sonolar [49]. A difficulty of this approach is that even simple floating-point formulas can have extremely large propositional encodings, which can be hard for current SAT solvers. This problem is addressed in [1], which uses a combination of over- and underapproximate propositional abstractions in order to keep the size of the search space as small as possible.

B. Lifting Decision Procedures

The practical success of CDCL solvers has given rise to various attempts to lift the algorithmic core of CDCL to new problem domains. This idea is extensively studied in the field of satisfiability modulo theories. The most popular such lifting is the DPLL(T) framework [50], which separates theory-specific reasoning from Boolean reasoning over the structure of the formula. Typically a propositional CDCL solver is used to reason about the Boolean structure while an ad-hoc procedure is used for theory reasoning. The DPLL(T) framework can suffer from some difficulties that arise from this separation. To alleviate these problems, approaches such as *theory decisions on demand* [4] and theory-based decision heuristics [51] have been proposed.

Our work is co-located in the context of natural-domain SMT [5], which aims to lift steps of the CDCL algorithm to operate directly over the theory. Notable examples of such approaches have been presented for equality logic with uninterpreted functions [52], linear real arithmetic and difference logic [5], [6], linear integer arithmetic [7], nonlinear integer arithmetic [9], and nonlinear real arithmetic [8]. The work in [9] is most similar to ours since it also operates over intervals and uses an implication graph construction.

We follow a slightly different approach to generalisation based on abstract interpretation. The work in [10] shows that SAT solvers can naturally be considered as abstract interpreters for logical formulas. Generalisations can then be obtained by using different abstract domains. Our work is an application of this insight. A similar line of research was independently undertaken in [53], [54], which presents an abstract-interpretation based generalisation of Stålmarck's method and an application to computation of abstract transformers.

C. Refining Abstract Analyses

A number of program analyses exist that use decision procedures or decision procedure architectures to refine a base analysis. A lifting of CDCL to program analyses over abstract domains is given in [55]. In [56], a decision-procedure based software model checker is presented that imitates the architecture of a CDCL solver. A lifting of DPLL(T) to refinement of abstract analyses is presented in [57] which combines a CDCL solver with an abstract interpreter.

Modern CDCL solvers can be viewed as refinements of the original DPLL algorithm [58], which is based on case-analysis. Case analysis has been studied in the abstract interpretation literature. The formal basis is given by cardinal power domains, already discussed in [13], in which a base domain is refined with a lattice of cases. The framework of *trace partitioning* [59] describes a systematic refinement framework for programs based on case analysis. The DPLL algorithm can be viewed as a special instance of dynamic trace partitioning applied to the analysis of logical formulas.

VII. Conclusions and Future Work

We have presented a decision procedure for the theory of floating-point arithmetic based on a strict lifting of the conflict analysis algorithm used in modern CDCL solvers to abstract domains. We have shown that, for a certain class of formulas, this approach significantly outperforms current complete solvers based on bit-vector encodings. Both our formalism and our implementation are modular and separate the CDCL algorithm from the details of the underlying abstraction. Furthermore, the overall architecture is not tied to analysing properties over floating-point formulas.

We are interested in a number of avenues of future research. One of these is a comparison of abstract CDCL and DPLL(T)-based architectures, and investigating possible integrations. Another avenue of research is instantiating ACDCL with richer abstractions (e.g., octagons). Combination and refinements of abstractions are well studied in the abstract interpretation literature [13]. Recent work [60] has shown that Nelson-Oppen theory combination is an instance of a product construction over abstract domains. We hope to apply this work to obtain effective theory combination within ACDCL. In addition, product constructions can be used to enhance the reasoning capabilities within a single theory, e.g., by fusing interval-based reasoning over floating-point numbers and propositional reasoning about the corresponding bit-vector encoding.

We see this work as a step towards integrating the abstract interpretation point of view with algorithmic advances made in the area of decision procedures. Black-box frameworks such as DPLL(T) abstract away from the details of their component procedures. Abstract interpretation can be used to express an orthogonal, algebraic "white-box" view which, we believe, has uses in both theory and practice.

Acknowledgements: We gratefully acknowledge the contributions of Vijay D'Silva who helped create the formal framework advocated in this paper.

REFERENCES

[1] A. Brillout, D. Kroening, and T. Wahl, "Mixed abstractions for floating-point arithmetic," in *FMCAD*. IEEE, 2009, pp. 69–76.

[2] C. Michel, "Exact projection functions for floating point number constraints," in *AMAI*, 2002.

[3] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. IOS Press, 2009, pp. 825–885.

[4] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Splitting on demand in SAT modulo theories," in *LPAR*, 2006, pp. 512–526.

[5] S. Cotton, "Natural domain SMT: a preliminary assessment," in *FORMATS*. Springer, 2010, pp. 77–91.

[6] K. McMillan, A. Kuehlmann, and M. Sagiv, "Generalizing DPLL to richer logics," in *CAV*. Springer, 2009, pp. 462–476.

[7] D. Jovanovic and L. de Moura, "Cutting to the Chase: Solving Linear Integer Arithmetic," in *CADE*. Springer, 2011, pp. 338–353.

[8] ——, "Solving non-linear arithmetic," in *IJCAR*. Springer, 2012, pp. 339–354.

[9] 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*, vol. 1, no. 3-4, pp. 209–236, 2007.

[10] V. D'Silva, L. Haller, and D. Kroening, "Satisfiability solvers are static analyzers," in *SAS*, ser. LNCS, vol. 7460. Springer, 2012, pp. 317–333.

[11] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *ICCAD*. ACM, 2001, pp. 279–285.

[12] P. Rümmer and T. Wahl, "An SMT-LIB theory of binary floating-point arithmetic," in *SMT Workshop*, 2010.

[13] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *POPL*, 1979, pp. 269–282.

[14] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*. IOS Press, 2009, pp. 131–153.

[15] L. Zhang, C. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *ICCAD*. IEEE, 2001, pp. 279–285.

[16] A. Griggio, "A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic," *JSAT*, vol. 8, pp. 1–27, January 2012.

[17] B. Botella, A. Gotlieb, and C. Michel, "Symbolic execution of floating-point computations," *STVR.*, vol. 16, no. 2, pp. 97–121, 2006.

[18] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*. Springer, 2008, pp. 337–340.

[19] D. Monniaux, "The pitfalls of verifying floating-point computations," *TOPLAS*, vol. 30, no. 3, 2008.

[20] M. Daumas, L. Rideau, and L. Théry, "A generic library for floating-point numbers and its application to exact computing," in *TPHOLs*. Springer, 2001, pp. 169–184.

[21] G. Melquiond, "Floating-point arithmetic in the Coq system," *Inf. Comput.*, vol. 216, pp. 14–23, 2012.

[22] P. S. Miner, "Defining the IEEE-854 floating-point standard in PVS," NASA, Langley Research, PVS. Technical Memorandum 110167, 1995.

[23] J. Harrison, "A machine-checked theory of floating point arithmetic," in *TPHOLs*. Springer, 1999, pp. 113–130.

[24] ——, "Floating point verification in HOL light: The exponential function," *FMSD*, vol. 16, no. 3, pp. 271–305, 2000.

[25] ——, "Formal verification of square root algorithms," *FMSD*, vol. 22, no. 2, pp. 143–153, 2003.

[26] ——, "Floating-point verification," *J. UCS*, vol. 13, no. 5, pp. 629–638, 2007.

[27] B. Akbarpour, A. Abdel-Hamid, S. Tahar, and J. Harrison, "Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL," *Comput. J.*, vol. 53, no. 4, pp. 465–488, 2010.

[28] R. Kaivola and M. Aagaard, "Divider circuit verification with model checking and theorem proving," in *TPHOLs*. Springer, 2000, pp. 338–355.

[29] J. S. Moore, T. Lynch, and M. Kaufmann, "A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm," *TC*, vol. 47, 1996.

[30] D. Russinoff, "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions," *JCM*, vol. 1, pp. 148–200, 1998.

[31] J. Harrison, "Formal verification of floating point trigonometric functions," in *FMCAD*, 2000, pp. 217–233.

[32] A. Ayad and C. Marché, "Multi-prover verification of floating-point programs," in *IJCAR*. Springer, 2010, pp. 127–141.

[33] S. Conchon, G. Melquiond, C. Roux, and M. Iguernelala, "Built-in treatment of an axiomatic floating-point theory for SMT solvers," in *SMT Workshop*, 2012.

[34] S. Boldo and J. Filliâtre, "Formal verification of floating-point programs," in *ARITH*. IEEE, 2007, pp. 187–194.

[35] A. Miné, "Relational abstract domains for the detection of floating-point run-time errors," in *ESOP*. Springer, 2004, pp. 3–17.

[36] L. Chen, A. Miné, and P. Cousot, "A sound floating-point polyhedra abstract domain," in *APLAS*. Springer, 2008, pp. 3–18.

[37] L. Chen, A. Miné, J. Wang, and P. Cousot, "Interval polyhedra: An abstract domain to infer interval linear relationships," in *SAS*. Springer, 2009, pp. 309–325.

[38] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *CAV*. Springer, 2009, pp. 661–667.

[39] A. Chapoutot, "Interval slopes as a numerical abstract domain for floating-point variables," in *SAS*. Springer, 2010, pp. 184–200.

[40] L. Chen, A. Miné, J. Wang, and P. Cousot, "An abstract domain to discover interval linear equalities," in *VMCAI*. Springer, 2010, pp. 112–128.

[41] J. Feret, "Static analysis of digital filters," in *ESOP*. Springer, 2004, pp. 33–48.

[42] D. Monniaux, "Compositional analysis of floating-point linear numerical filters," in *CAV*. Springer, 2005, pp. 199–212.

[43] E. Goubault, "Static analyses of the precision of floating-point operations," in *SAS*. Springer, 2001, pp. 234–259.

[44] K. Ghorbal, E. Goubault, and S. Putot, "The zonotope abstract domain Taylor1+," in *CAV*. Springer, 2009, pp. 627–633.

[45] E. Goubault, S. Putot, P. Baufreton, and J. Gassino, "Static analysis of the accuracy in control systems: Principles and experiments," in *FMICS*. Springer, 2007, pp. 3–20.

[46] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, "Towards an industrial use of FLUCTUAT on safety-critical avionics software," in *FMICS*, 2009, pp. 53–69.

[47] C. Michel, M. Rueher, and Y. Lebbah, "Solving constraints over floating-point numbers," in *CP*, 2001, pp. 524–538.

[48] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.

[49] E. V. Jan Peleska and F. Lapschies, "Automated test case generation with SMT-solving and abstract interpretation," in *NFM*. Springer, 2011, pp. 298–312.

[50] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," in *CAV*. Springer, 2004, pp. 175–188.

[51] D. Goldwasser, O. Strichman, and S. Fine, "A theory-based decision heuristic for DPLL(T)," in *FMCAD*, 2008, pp. 1–8.

[52] B. Badban, J. van de Pol, O. Tveretina, and H. Zantema, "Generalizing DPLL and satisfiability for equalities," *Inf. Comput.*, vol. 205, no. 8, pp. 1188–1211, 2007.

[53] A. Thakur and T. Reps, "A method for symbolic computation of abstract operations," in *CAV*. Springer, 2012.

[54] ——, "A generalization of Stålmarck's method," in *SAS*. Springer, 2012.

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

[56] K. L. McMillan, "Lazy annotation for program testing and verification," in *CAV*, 2010, pp. 104–118.

[57] W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Program analysis via satisfiability modulo path programs," in *POPL*, 2010, pp. 71–82.

[58] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *CACM*, vol. 5, pp. 394–397, July 1962.

[59] X. Rival and L. Mauborgne, "The trace partitioning abstract domain," *TOPLAS*, vol. 29, no. 5, 2007.

[60] P. Cousot, R. Cousot, and L. Mauborgne, "The reduced product of abstract domains and the combination of decision procedures," in *FoSSaCS*, 2011, pp. 456–472.