Abstract Satisfaction

Leopold Carl Robert Haller

Magdalen College

A thesis submitted for the degree of

Doctor of Philosophy

Trinity Term, 2013

Abstract

This dissertation shows that satisfiability procedures are abstract interpreters. This insight provides a unified view of program analysis and satisfiability solving and enables technology transfer between the two fields. The framework underlying these developments provides systematic recipes that show how intuition from satisfiability solvers can be lifted to program analyzers, how approximation techniques from program analyzers can be integrated into satisfiability procedures and how program analyzers and satisfiability solvers can be combined. Based on this work, we have developed new tools for checking program correctness and for solving satisfiability of quantifier-free first-order formulas. These tools outperform existing approaches.

We introduce abstract satisfaction, an algebraic framework for applying abstract interpretation to obtain sound, but potentially incomplete satisfiability procedures. The framework allows the operation of satisfiability procedures to be understood in terms of fixed point computations involving deduction and abduction transformers on lattices. It also enables satisfiability solving and program correctness to be viewed as the same algebraic problem.

Using abstract satisfaction, we show that a number of satisfiability procedures can be understood as abstract interpreters, including Boolean constraint propagation, the DPLL and CDCL algorithms, Stålmarck's procedure, the DPLL(T) framework and solvers based on congruence closure and the Bellman-Ford algorithm. Our work leads to a novel understanding of satisfiability architectures as refinement procedures for abstract analyses and allows us to relate these procedures to independent developments in program analysis. We use this perspective to develop Abstract Conflict-Driven Clause Learning (ACDCL), a rigorous, lattice-based generalization of CDCL, the central algorithm of modern satisfiability research. The ACDCL framework provides a solution to the open problem of lifting CDCL to new problem domains and can be instantiated over many lattices that occur in practice. We provide soundness and completeness arguments for ACDCL that apply to all such instantiations.

We evaluate the effectiveness of ACDCL by investigating two practical instantiations: FP-ACDCL, a satisfiability procedure for the first-order theory of floating point arithmetic, and CDFPL, an interval-based program analyzer that uses CDCL-style learning to improve the precision of a program analysis. FP-ACDCL is faster than competing approaches in 80% of our benchmarks and it is faster by more than an order of magnitude in 60% of the benchmarks. Out of 33 safe programs, CDFPL proves 16 more programs correct than a mature interval analysis tool and can conclusively determine the presence of errors in 24 unsafe benchmarks. Compared to bounded model checking, CDFPL is on average at least 260 times faster on our benchmark set.

Credit

In the following, I clarify which parts of the work presented in this thesis are not wholly my own. I use all such work with permission of my collaborators.

Much of the formal work presented in this dissertation was developed in close collaboration with Vijay D'Silva. This includes the following parts of this dissertation: (i) the methodology for applying abstract interpretation to logical satisfiablity in Section 3.1 which is based on material jointly published in [58]; (ii) the definition of the bottom-everywhere problem described in Section 3.2, which was jointly published in [59], (iii) the abstract-interpretation characterizations of BCP and CDCL in Sections 4.1 and 4.4, which were jointly published in [58]; (iv) the characterization of DPLL in Section 4.2, which is similar to an account that was jointly published in [58]; (v) part of the material in Sections 5.1, 5.2 and 5.3 which were published in [59]; (vi) the idea for the CDFPL algorithm described Section 7.4, which was conceived jointly.

Section 4.5.3 is based on a section of [25] that was developed jointly with Martin Brain who provided the figures and initial drafts of the proofs. Chapter 6 is based on the work described in [89]; all implementation work for this chapter was done by Alberto Griggio based on an unpublished prototype tool that I wrote; experimental results were obtained jointly by Alberto Griggio and Martin Brain. Chapter 7 is based on the work in [60]; Michael Tautschnig helped with obtaining benchmarks and experimental results.

Acknowledgements

I would like to thank Manfred Wagenhuber, my high-school mathematics teacher, for giving me my first glimpse of mathematical rigor. Armin Biere, for teaching me about SAT solvers and Uwe Egly for giving me the opportunity to write one. My supervisor, Daniel Kröning, for providing me with a level of support and freedom that not many graduate students get to experience. The group at Oxford and at ETH Zürich, Vijay D'Silva, Georg Weissenbacher, Yury Chebiryak, Gérard Basler, Christoph Wintersteiger, Nicolas Blanc, Michele Mazzucchi, Mitra Purandare, Angelo Brillout, Alexander Kaiser, Thomas Wahl, Philipp Ruemmer, Alastair Donaldson, Michael Tautschnig, Vincent Nimal, Jade Alglave, Ajitha Rajan, Peter Schrammel, Martin Brain, Nassim Seghir, Ganesh Narayanaswamy, Matt Lewis, David Landsberg and Pamela Farris for the company, many interesting discussions and a sense of family. Satnam Singh, my mentor at Microsoft Research Cambridge, for widening my horizons regarding hardware design and single-malt whisky. My friends, Reinier Van Straten and Anton Baker for offering distractions when I needed them. Greta Yorsh, for supporting this work in its early stages and for helping me spread the word. Radhia and Patrick Cousot, for their warmth, support and encouragement.

Special mention is due to my parents Christine Haller and Herbert Haller and to my siblings Anna and Christina, for their ongoing love and support, which is the basis for everything I have achieved; to my wife, Fazilat Nassiri, who makes me unreasonably happy and has kept me reasonably sane throughout the writing of this document; to Vijay D'Silva, who walked this path with me, for being a great friend, an invaluable teacher, a walking library, and one of the most inspired and clear thinkers I have met.

Publications on Abstract Satisfaction

V. D'Silva, L. Haller, and D. Kroening. Abstract satisfaction. In *Proceedings of Principles of Programming Languages*, to appear. ACM Press, 2014.

V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. In *Proceedings of Principles of Programming Languages*, pages 143–154. ACM Press, 2013.

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

M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening. Interpolation-based verification of floating-point programs with Abstract CDCL. In *Proceedings of Static Analysis Symposium*, pages 412–432. Springer, 2013.

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

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

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

Contents

Chapter 1

Introduction

This dissertation argues the following thesis:

> Decision procedures for satisfiability are abstract interpreters. This insight allows technology transfer between program analysis and satisfiability research and can be used to define algebraic generalizations of satisfiability architectures. The resulting generalizations have practical applications.

Approaches to program verification can be classified as performing bug-finding or proof construction. Bug-finding approaches search for program behaviors that are counterexamples to correctness. Proof-construction approaches find program invariants which imply correctness. These two approaches are enabled by two technologies: abstract domains, which offer approximate representations of sets of program behaviors, and satisfiability procedures for propositional and first-order logics.

We relate three approaches in program verification to these categories [101, 40] in Figure 1.1: (i) Symbolic execution is a bug-finding methodology where program paths are encoded as logical formulas. A satisfiability procedure is used to check if counterexample exist. (ii) Software model checking focuses on building tools that can prove both correctness of programs and find bugs. The use of abstract domains in software model checkers focuses on powerset abstractions that are able to precisely express sets of abstract states [35]. Decision procedures are used to find counterexamples and synthesize abstractions [35] and to incrementally construct logical invariants [122, 32]. (iii) Static analysis approximates program semantics by computing fixed points in abstract domains. In contrast to software model checking, which focuses on powerset abstractions, static analysis typically operates over lattices that lack distributivity. These lattices are more efficient but potentially less precise compared to those used in model checking.[1]

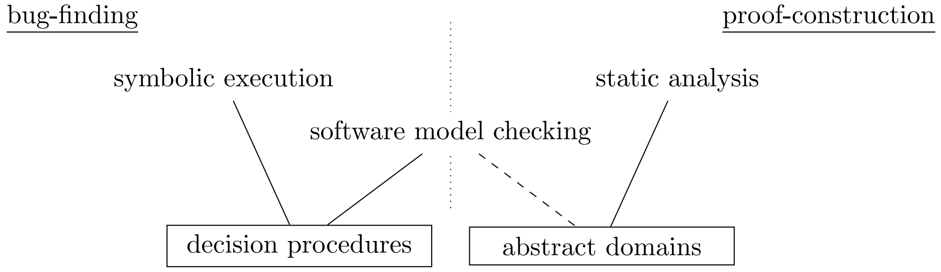


Figure 1.1: Approaches and technologies in program verification.

[1] Software model checking and symbolic execution can also be viewed as instances of static analysis over abstract domains. The taxonomy employed here is therefore somewhat artificial, but it corresponds to a

Since the year 2000, there has been a convergence of research and a number of approaches have been developed which combine techniques developed in different areas [101]. The combination of model checking and symbolic execution forms the basis for the well-known CEGAR model checking framework [35] which finds error candidates using an approximate powerset analysis and checks whether these candidates are feasible using symbolic execution. Combinations of software model checking and static analysis use static analyzers to compute program invariants to increase efficiency of model checking algorithms [97]. Static analysis and symbolic execution are combined, for example, in [171], where the fixed point computation in a static analysis engine is guided by the results of symbolic execution runs.

To increase the effectiveness of such combinations, techniques have been proposed which directly access the underlying technologies and integrate decision procedures into abstract static analyses or use abstract domains inside model checkers. Decision procedures are used to synthesize transformers for abstract domains [134, 7], or to enumerate potentially unsafe program fragments to be proven correct with static analysis [91]. Modern software model checking frameworks [15] allow flexible integrations of abstract domains. The work in [150, 166, 167] explores the use of abstract domains to approximate the meaning of logical formulas and the use of decision procedures in computing these approximations.

All of the approaches discussed so far take a black-box approach to combining abstract domains and decision procedures. From a static analysis point of view, decision procedures are oracles that give binary answers to semantic queries. From the decision procedure point of view, static analyzers are considered invariant generation procedures. This black-box perspective limits the potential of combinations and makes it difficult to lift successful algorithms from one area and implement them in the other.

To progress beyond simple black-box combinations, a number of open methodological questions need to be solved.

(i) How can ideas from decision procedures, such as clause learning[2], be lifted to abstract domains and static analyzes?

(ii) How can approximation techniques used in abstract domains and static analysis be lifted to decision procedures?

(iii) How can static analyzers and decision procedures be combined in more effective, semantically aware ways that go beyond simple black-box combinations?

(iv) How can existing research in static analysis and decision procedures be formally related?

To address these questions, we introduce a novel framework called *abstract satisfaction* which is based on abstract interpretation. Abstract satisfaction characterizes logical inference in terms of lattices, transformers and fixed points and provides a mathematical foundation for satisfiability solvers. It also provides a methodology for deriving sound but incomplete procedures for deciding the satisfiability and validity problems for a logic.

We use the abstract satisfaction framework to prove the central claim of this thesis: that satisfiability solvers are abstract interpreters. To this end, we show that existing satisfiability solvers can be decomposed into fixed point computations over abduction and deduction transformers that use extrapolation and interpolation operators. This perspective provides the same benefits in the analysis and construction of solvers that abstract interpretation provides in static analysis and allows us to view satisfiability solvers as iteration engines, parametrized by abstract domains. We show that this view is universal and faithful by characterizing a number of different satisfiability procedures in this way, including Boolean constraint propagation, the DPLL and CDCL algorithms, Stålmarck's procedure, the DPLL(T)

perspective that is common among verification practitioners.

[2] Clause learning is a form of guided inference, and we use the term "learning" in this sense throughout this document.

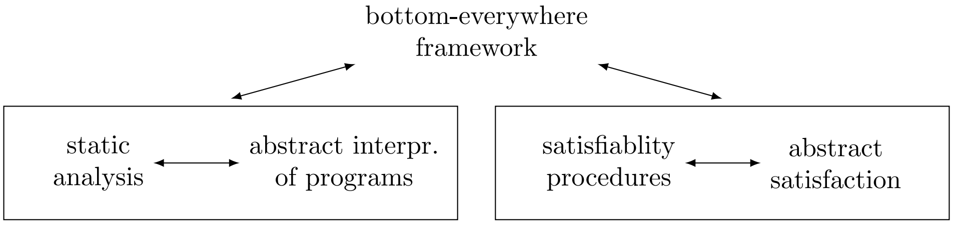


Figure 1.2: The Bottom-Everywhere Framework

framework and solvers based on congruence closure and the Bellman-Ford algorithm. The result of this analysis is a new understanding of satisfiability algorithms as refinement procedures for analyses that operate on non-distributive lattices. These refinements can be lifted to new problem domains and are fundamentally different from existing refinement techniques such as CEGAR, which only apply to Boolean powerset lattices.

Abstract satisfaction is a framework for approximate inference about logical formulas. Static analysis frameworks based on abstract interpretation are concerned with approximating the semantics of a program. To rigorously relate these two applications of abstract interpretation we introduce the *bottom-everywhere framework*. Logical satisfiability and program correctness are both instances of the *bottom-everywhere problem*, a decision problem that involves closure operators on Boolean algebras. The relation between abstract interpretation, abstract satisfaction and the bottom-everywhere framework is depicted in Figure 1.2. The bottom-everywhere framework generalizes logical inference and shows that strongest postcondition and weakest precondition transformers are the program analysis counterparts of deduction and abduction transformers. The bottom-everywhere framework also allows us to create generalizations of satisfiability architectures that can be applied equally to problems in satisfiability and program correctness. This constitutes a significant contribution, since it provides refinement architectures for non-distributive domains, whereas common refinement techniques only apply to Boolean domains.

One such refinement architecture presented in this dissertation and is *Abstract Conflict Driven Clause Learning* (ACDCL), an algebraic generalization of the CDCL algorithm. CDCL is the central algorithm in decision procedure research today and has had significant impact on the field of satisfiability checking in the past 15 years. We lift the algorithm to lattices, prove soundness and completeness and characterize a generalization of clause learning in an abstract, lattice-theoretic setting.

The ACDCL algorithm shows how CDCL can be lifted to new problems and domains in a way that is both mathematically rigorous and conceptually simple. New algorithmic instantiations can be systematically derived from abstract domains that satisfy certain complementation properties. These properties hold in many lattices used in practice. ACDCL provides an answer to the open methodological problem of lifting CDCL to new problem domains, which has been a focus of active and ongoing research in the satisfiability community. We present ACDCL as an algorithmic framework to solve the bottom-everywhere problem, which allows instantiations of both program analyzers and satisfiability checkers.

In order to demonstrate that instantiations of ACDCL are effective in practice, we present two tools: the satisfiability solver FP-ACDCL for the first-order theory of floating-point arithmetic and the interval-analysis based program verifier CDFPL. Both instantiate the ACDCL framework. FP-ACDCL is the state-of-the-art solver for its logic. It is faster than competing approaches 80% of the time and faster by at least an order of magnitude in 60% of cases. Out of 33 safe programs, CDFPL proves 16 more programs correct than a mature interval analysis tool and can conclusively determine the presence of errors in 24 unsafe benchmarks. Compared to bounded model checking, CDFPL is on average at least 260 times faster on our benchmark set.

Outline and Contributions

We now outline the structure of this document and list the contributions of each chapter.

Chapter 2 introduces formal preliminaries. This includes basic background on order and lattice theory, an introduction to abstract interpretation which covers the concepts we will use throughout this thesis and a brief section on logic, including propositional logic and the first-order theories studied in the area of Satisfiability Modulo Theories solving (SMT). Chapter 2 contains no new material.

Chapter 3 introduces *abstract satisfaction* and the *bottom-everywhere problem*, which form the formal framework used throughout this thesis. The contributions of this chapter are as follows.

- We identify transformers that correspond to various modes of logical inference. We give fixed point characterizations of the satisfiability and validity problem of a logic.
- We show that abstract interpretation can be used to approximate these fixed points, which provides sound but incomplete methodologies for satisfiability and validity.
- We introduce a novel, purely lattice-theoretic problem called bottom-everywhere, which generalizes the satisfiability problem for a logic and the problem of program correctness. Any techniques developed to solve the bottom-everywhere problem can be instantiated equally to solve satisfiability or program correctness.

Chapter 4 argues that SAT solvers are abstract interpreters by giving abstract interpretation accounts of a number of common satisfiability procedures and relating them to research in program analysis. We make the following contributions.

- We show that abstract satisfaction can be used to give accounts of decision procedures in terms of basic algebraic components such as deduction transformers, abduction transformers, extrapolation and interpolation.
- We show that partial assignments in the DPLL algorithm are an abstract domain that is well-known in program analysis, that the unit rule is an abstract transformer and that Boolean Constraint Propagation (BCP), the workhorse deduction procedure of DPLL, computes a greatest fixed point over this abstract domain. DPLL and abstract interpretation were proposed more than 30 years ago yet we are, to our knowledge, the first to make these observations.
- We show that DPLL and Stålmarck's procedures are case-based refinement procedures for non-distributive lattices and we identify the algebraic structure necessary to instantiate them. We give general algebraic recipes that show how these procedures can be instantiated over a range of problems and abstract domains.
- We show that CDCL computes fixed points in overapproximate and underapproximate lattices equipped with acceleration operators. Specifically, CDCL alternates between a greatest fixed point computation with extrapolation and a least fixed point computation with interpolation.
- We show that the DPLL(T) algorithm is an abstract interpreter for first-order formulas and that it computes a reduced product between a propositional abstraction and a theory abstraction. We characterize two theory solvers that are commonly used within DPLL(T) as abstract interpreters, namely, the congruence closure and Bellman-Ford algorithms.

Chapter 5 presents ACDCL, an abstract-interpretation based lifting of CDCL to the bottom-everywhere problem, and makes the following contributions.

- A conceptual framework for conflict-driven learning procedures in a lattice-theoretic setting.
- A characterization of learning as an overapproximation of negation and the definition of a best learning transformer for an abstract domain.
- A mathematically rigorous lifting of the CDCL algorithm to abstract domains and the bottom-everywhere problem.
- Soundness, completeness and termination proofs for ACDCL in a lattice-theoretic setting.
- A lifting of the FIRSTUIP conflict analysis algorithm to lattice-based abstractions.

Chapters 6 and 7 demonstrate the effectiveness and generality of ACDCL by showing how ACDCL can be applied to derive novel program analyses and satisfiability procedures. The contributions of Chapter 6 are as follows.

- The FP-ACDCL tool, a sound and complete satisfiability solver for the first-order theory of *Floating-Point Arithmetic* (FPA) based on the abstract domain of floating-point intervals. Reasoning in FPA is considered a hard problem since standard algebraic techniques for the reals do not apply [133]. FP-ACDCL is currently the state-of-the-art solver for that logic.
- An extensive empirical evaluation using more than 200 benchmark formulas which shows that FP-ACDCL is faster than competing procedures in 80% of cases and is faster by more than an order of magnitude in 60% of cases.
- A lattice-independent conflict analysis heuristic called *trail-guided choice.*
- An empirical study of the efficacy of heuristics for decision making and conflict analysis.
- An empirical exploration of the trade-off between efficiency and precision in lattice-based conflict analysis.

Chapter 7 makes the following contributions.

- A methodology for instantiating ACDCL to reason about the static analysis equation of a program.
- The CDFPL tool, a tool that extends interval analysis with learning to reason precisely about disjunction in programs that contain machine integer and floating-point variables.
- A comparison to the mature abstract interpreter ASTRÉE and the bounded model checker CBMC, which shows that CDFPL is more precise than standard interval analysis and orders of magnitude faster than CBMC on bounds checking problems.
- An analysis of CDFPL as an automatic analysis refinement approach; we show that CDFPL automatically induces a program and property driven refinement of a non-distributive abstract analysis and we demonstrate empirically that the precision of this refinement is adjusted dynamically to match the hardness of the problem.

In this dissertation, we show that satisfiability solvers can be understood as abstract interpreters. We provide a formal framework for applying abstract interpretation to solve the satisfiability problem and we give a mathematical generalization that enables us to give equal treatment to logical satisfiability and program correctness. We demonstrate the power of our formalization by providing characterizations of a number of decision procedures in terms of transformers, fixed points and acceleration operators on lattices and we extract general algebraic recipes for refinement algorithms for static analyses. As a result we are able to give a generalization of CDCL, the central algorithm in modern decision procedure research. We demonstrate the practical effectiveness of our approach by instantiating state-of-the-art satisfiability procedures and program analyzers based on ACDCL and we show that they outperform competing approaches. These ideas are carried out within a rigorous framework and are the result of significant new insights and developments.

The work in this thesis eliminates the conceptual boundary between static analysis and satisfiability research in both theory and practice. It is a significant step towards enabling technology transfer between the two fields and towards allowing practical combinations of procedures that go beyond black-box integration.

Chapter 2

Lattices, Abstraction and Logic

In this chapter, we introduce material relating to order theory, lattice theory, abstract interpretation and logic. The material in this section is not new.

Outline Section 2.1 introduces mathematical notation used in this document. We provide background on order theory and lattices in Section 2.2, a survey of relevant concepts in abstract interpretation theory in Section 2.3 and background on logic in Section 2.4.

2.1 Formal Preliminaries

We now introduce notation used throughout this thesis.

Definitions We use the symbol "$\hat{=}$" to denote definitions, and "$=$" to express equality. For example, $f(x) \hat{=} g(x)$ defines the function f, whereas $f(x) = g(x)$ expresses that f and g map x to the same element. Function definitions that include multiple cases are read top-to-bottom, that is, the order of the definition defines the precedence in cases of ambiguity. Consider the following example.

$$f(x) \hat{=} \begin{cases} 1 & \text{if } x \text{ is even} \\ 1 & \text{if } x = 3 \\ 0 & \text{if } x \text{ is odd} \end{cases}$$

The function defined above maps 3 to 1, as prescribed by the second case, even though the third case would map 3 to 0.

Sets The *powerset* of a set S contains all subsets of S and is denoted $\mathcal{P}(S)$. The *set-theoretic complement* of S is written $\neg S$. For two sets A and B, we write $A \setminus B$ to denote the set $A \cap \neg B$.

Functions and Relations Consider a function $f : A \to B$ with *domain* A and *codomain* B. The *image of a subset* $X \subseteq A$ under f is the set $\{f(x) \mid x \in X\}$. The *image of the function* f is the image of A under f. Consider a function $f \hat{=} g(x) + 3$. In cases where we wish to avoid explicitly naming the function denoted by f, we write $\lambda x.\ g(x) + 3$ or simply $x \mapsto g(x) + 3$. For $a \in A$ and $b \in B$ we write $f[a \mapsto b]$ to denote the function that maps a to b and every other element $a' \neq a$ to $f(a')$. Every set S is associated with an identity function $id_S : S \to S$, given by $s \mapsto s$. Given functions $f : A \to B$ and $g : B' \to C$ where $B \subseteq B'$, their *composition* $g \circ f$ is the function $\lambda a.\ g(f(a))$.

Sequences Given a set A we denote by A^* and A^+ the set of finite sequences and the set of finite, non-empty sequences over elements of A, respectively. The empty sequence is written ϵ. For two sequences π and ρ, we denote their concatenation by $\pi\rho$ or by $\pi \cdot \rho$. The sequence π is a prefix of ρ if $\rho = \pi \cdot \rho'$ for some sequence ρ', and a suffix if $\rho = \rho' \cdot \pi$ for some ρ'. A set of sequences Π is prefix-closed if for every $\pi \in \Pi$ and prefix ρ of π, ρ is in Π. The *length* of the sequence π is defined as usual and denoted $|\pi|$. The ith element of the sequence π is denoted π_i, that is, π_1 is the first element of π. For a sequence π and an element a we write $a \in \pi$ if a occurs in π, i.e., that there exists an i with $1 \leq i \leq |\pi|$ such that $\pi_i = a$.

2.2 Lattices and Order

In this section, we recall basic concepts in order and lattice theory, and fix notation. We discuss ways in which lattices can be derived from partially ordered sets, and conversely, ways of decomposing lattices into their partially ordered building blocks. The interested reader may refer to [52] for a more in-depth treatment.

2.2.1 Partial Orders and Lattices

Definition 2.2.1 (Partial Order)**.** A *partial order* over a set P is a binary relation $\sqsubseteq$ over P such that for all $a, b, c \in P$, it holds that

(i) $a \sqsubseteq a$ (*reflexivity*),

(ii) if $a \sqsubseteq b$ and $b \sqsubseteq c$ hold, then $a \sqsubseteq c$ (*transitivity*) and

(iii) if $a \sqsubseteq b$ and $b \sqsubseteq a$ hold, then $a = b$ (*anti-symmetry*).

We refer to $(P, \sqsubseteq)$ (or simply to P, if the order $\sqsubseteq$ is clear from context) as a *partially ordered set* or simply *poset*. If the order is clear, then we say for $a, b \in P$ that a is *less than* b and that b is *greater than* a if $a \sqsubseteq b$ holds. If at least one of these conditions holds, then a and b are comparable. If a is less than b and there is no element between a and b, i.e., $\forall p \in P \setminus \{a, b\}.\ a \not\sqsubseteq p \vee p \not\sqsubseteq b$ then b *covers* a.

We now define special types of maps between posets.

Definition 2.2.2. Let $(P, \sqsubseteq)$ and $(S, \leq)$ be posets. A function $f : P \to Q$ is

- *monotone* or *order-preserving* if whenever $p \sqsubseteq q$ then $f(p) \leq f(q)$,
- an *order embedding* if $p \sqsubseteq q \iff f(p) \leq f(q)$ and
- an *order isomorphism* if it is a surjective order embedding.

Note that an order embedding is necessarily injective, therefore, order isomorphisms are bijective functions. We define two posets to be order isomorphic if there is an order isomorphism between them. Order isomorphism provides a useful notion of equivalence between ordered structures.

Definition 2.2.3 (Greatest Lower Bound)**.** Let $(P, \sqsubseteq)$ be a poset and Q be a subset of P. An element $p \in P$ is a *lower bound* of Q if every $q \in Q$ is greater than p. It is the *greatest lower bound* or *meet* of Q, denoted $\bigsqcap Q$, if every lower bound of Q is less than p.

Duality We refer to $\sqsupseteq$ as the *dual order* with respect to the order $\sqsubseteq$. For any order-theoretic statement or structure, its dual is obtained by replacing all components of its definition with their dual statement. For example, $(P, \sqsupseteq)$ is the $\sqsubseteq$-dual poset to $(P, \sqsubseteq)$. The dual of the greatest lower bound is the least upper bound. We give an explicit definition of the least upper bound, but from now on, we will sometimes refrain from doing so for dual concepts. The duality principle asserts that if a statement is true in all ordered structures, then its $\sqsubseteq$-dual statement is true is all ordered structures. We invoke this principle when we say that a statement follows from duality.

Definition 2.2.4 (Least Upper Bound)**.** Let $(P, \sqsubseteq)$ be a poset and Q be a subset of P. An element $p \in P$ is an *upper bound* of Q if every $q \in Q$ is less than p. It is the *least upper bound* or *join* of Q, denoted $\bigsqcup Q$, if every *upper bound* is greater than p.

Some comments regarding notation follow. For two elements a, b we denote their meet by $a \sqcap b$ and their join by $a \sqcup b$. The relations $\sqsupseteq$, $\sqsubset$ and $\sqsupset$ are defined as usual. Throughout this document, we will use a number of symbols to denote orderings, for example, $\sqsubseteq$, $\dot{\sqsubseteq}$ and $\preceq$. We appropriately match the symbols for meets and joins. In the above examples, we would use, respectively, $\sqcap$ and $\sqcup$, $\dot{\sqcap}$ and $\dot{\sqcup}$, $\curlywedge$ and $\curlyvee$ to denote meets and joins. To simplify notation and where the meaning is clear from context, we will sometimes use an ordering symbol $\sqsubseteq$ to denote more than one ordering relation.

We now introduce some terminology regarding functions that preserve meets and joins.

Definition 2.2.5. A function $f : P \to Q$ between posets $(P, \sqsubseteq)$ and $(Q, \preceq)$ is

- *multiplicative* (or *meet-preserving*) if whenever a and b have a meet $a \sqcap b$, then $f(a)$ and $f(b)$ have a meet $f(a) \curlywedge f(b)$ and $f(a \sqcap b) = f(a) \curlywedge f(b)$.
- *completely multiplicative* (or *completely meet-preserving*) if whenever $R \subseteq P$ has a meet $\bigsqcap R$, then the set $S = \{f(r) \mid r \in R\}$ has a meet $\bigcurlywedge\{f(r) \mid r \in R\}$ and $f(\bigsqcap R) = \bigcurlywedge\{f(r) \mid r \in R\}$.

The dual concept is that of a (completely) *additive* or *join-preserving* function.

Functions that map into partially ordered sets can themselves be partially ordered by pointwise lifting.

Definition 2.2.6 (Pointwise Lifting)**.** Let $(P, \sqsubseteq)$ be a poset. The *pointwise order* of $\sqsubseteq$ lifted to the set $A \to P$ is the partial order $\dot{\sqsubseteq}$ on $A \to P$, defined as $f \mathrel{\dot{\sqsubseteq}} g$ if for all a in A, $f(a) \sqsubseteq f(g)$. The *pointwise lifting* of an n-ary function $h : P^n \to P$ to the set $A \to P$ is the function $\dot{h} : (A \to P)^n \to (A \to P)$ given as $\dot{h}(f_1, \ldots, f_n) \mathrel{\hat{=}} \lambda a.\ h(f_1(a), \ldots, f_n(a))$.

In cases where ambiguities do not arise, we will use the same symbols to denote operators or orders and their pointwise liftings.

Lattices We now introduce lattices.

Definition 2.2.7 (Lattice)**.** A *lattice* is a poset $(L, \sqsubseteq)$ such that for every two elements $a, b \in L$ the join $a \sqcup b$ and the meet $a \sqcap b$ exists.

We denote lattices either by their underlying poset $(L, \sqsubseteq)$ or by $(L, \sqsubseteq, \sqcup, \sqcap)$. If order, meets and joins are clear from context we denote a lattice by the carrier set L.

In a lattice, only the existence of binary joins and meets is guaranteed. A lattice in which every set has a join and a meet is called *complete*.

Definition 2.2.8 (Complete Lattice)**.** A lattice $(L, \sqsubseteq)$ is *complete* if every set $Q \subseteq P$ has a meet $\bigsqcap Q$ and a join $\bigsqcup Q$.

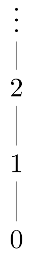


Figure 2.1: The poset $(\mathbb{N}_0, \leq)$ is a lattice with $\bigsqcap Q$ defined as the minimal element in Q. It is not a complete lattice; for example, $\bigsqcup \mathbb{N}_0$ does not exist.

Figure 2.1 shows an example of a lattice that is not complete. It is easy to see that every finite lattice is complete. Furthermore, it is possible to express arbitrary meets in terms of arbitrary joins and vice versa, therefore, in a lattice, the existence of arbitrary joins *or* arbitrary meets implies the existence arbitrary joins *and* arbitrary meets. For example, the element $\bigsqcap Q$ can be equivalently expressed as $\bigsqcup \{a \in L \mid \forall q \in Q.\ a \sqsubseteq q\}$.

A related concept is *boundedness* which asserts the existence of a least or greatest element.

Definition 2.2.9 (Top and Bottom)**.** Let $(P, \sqsubseteq)$ be a poset. An element $p \in P$ is the *top element* if every element of P is less than p. The *bottom element* is defined dually. We denote the top and bottom elements of P, if they exist, by $\top$ and $\bot$, respectively.

When discussing multiple bounded posets, if the meaning is clear from context, we will sometimes use the symbols $\top$ and $\bot$ to denote the top and bottom element of more than one lattice. We also sometimes denote the top and bottom element of a poset P by $\top_P$, respectively, $\bot_P$ to avoid this ambiguity.

Definition 2.2.10 (Bounded Lattice)**.** A *bounded lattice* is a lattice with a top and a bottom element.

Complete lattices are bounded, with $\bigsqcap \emptyset = \top$ and $\bigsqcup \emptyset = \bot$.

Chains are totally ordered subsets of partial orders that are important for characterizing finiteness conditions, called the chain conditions.

Definition 2.2.11 (Chains)**.** A *chain* in an ordered set P is a subset Q of pairwise comparable elements of P. The set of chains in P is denoted $\mathcal{C}(P)$.

Increasing sequences of elements of the form $s_0 \sqsubseteq s_1 \sqsubseteq \cdots$ define a chain $\{s_i \mid i \in \mathbb{N}\}$. The same holds for decreasing sequences. To simplify exposition, we will sometimes refer to such sequences as chains.

Chains allow us to define a notion of height in a partial order.

Definition 2.2.12 (Height)**.** The *height* of a poset is the cardinality of its longest chain.

A lattice has finite height exactly if it satisfies both of the chain conditions, which are introduced next.

Definition 2.2.13 (Chain Conditions)**.** A poset P satisfies the ascending chain condition (ACC) if every chain with a minimum is finite. The descending chain condition (DCC) is defined dually.

Other important structures in lattices are downsets and upsets, which obey certain closure properties.

Definition 2.2.14 (Downsets and Upsets)**.** Let $(P, \sqsubseteq)$ be a poset. The *downwards closure* of a set $Q \subseteq P$, denoted $\downarrow Q$, is the set $\{p \in P \mid \exists q \in Q.\ p \sqsubseteq q\}$. The set Q is *downwards closed*, or is a *downset* if $Q = \downarrow Q$. The *downset* of an element $p \in P$, denoted $\downarrow p$, is the downwards closure $\downarrow\{p\}$. The set of all downsets of P is denoted $\mathcal{D}(P)$. The notion of *upward closure* and *upsets* is dual, denoted $\uparrow Q$ and $\uparrow p$. The set of all *upsets* of P is denoted $\mathcal{U}(P)$.

2.2.2 Distributive Lattices and Boolean Algebras

We will now introduce distributive lattices, lattice-theoretic complements and Boolean algebras. In this dissertation, we will define semantics in terms of Boolean algebras. Many of the techniques presented in later chapters of this dissertation can be understood as restoring information lost by approximating these Boolean algebras using non-distributive lattices.

Definition 2.2.15 (Distributivity)**.** A lattice $(L, \sqsubseteq, \sqcap, \sqcup)$ is *distributive* if for all $a, b, c \in L$, it holds that $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$.

Distributivity is self-dual, i.e., the above condition is equivalent to the condition $a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$.

Definition 2.2.16 (Complements)**.** Given an element a in bounded lattice L, an element b is defined to be a *complement* of a if $a \sqcap b = \bot$ and $a \sqcup b = \top$. A *complementation* is a function $\neg\cdot : L \to L$ that maps every element to one of its complements. The lattice L is complemented if it admits a complementation.

In distributive lattices, complements are unique. This is not true in arbitrary lattices.

Definition 2.2.17 (Boolean Algebra)**.** A *Boolean algebra* is a complemented, distributive lattice.

In a Boolean algebra, complements are related to meets and joins via De Morgan's laws.

Proposition 2.2.1 (De Morgan's Laws)**.** *For any elements a, b of a Boolean algebra B with complementation $\neg\cdot$, the following equivalences hold.*

$$\neg(a \sqcup b) = \neg a \sqcap \neg b \qquad\qquad \neg(a \sqcap b) = \neg a \sqcup \neg b$$

Infinitary versions of De Morgan's laws hold in complete Boolean algebras.

Proposition 2.2.2. *For any subset Q of a complete Boolean algebra L and its unique orthocomplementation $\neg\cdot$, the following equivalences hold:*

$$\neg \bigsqcup Q = \bigsqcap_{q \in Q} \neg q \qquad\qquad \neg \bigsqcap Q = \bigsqcup_{q \in Q} \neg q$$

The relationship between meets and joins expressed by De Morgan's laws will be of special significance to us in later chapters. We now introduce a concept that lifts the relationship between the binary operators $\sqcap$ and $\sqcup$ to arbitrary operators over Boolean algebra.

Definition 2.2.18 (De Morgan Duality)**.** For $k \in \mathbb{N}$, let $f : L^k \to L$ be a function over a Boolean algebra L with complementation $\neg$. The *De Morgan dual* of f is the function $\tilde{f} = \lambda q_1, \ldots, q_k.\ \neg \circ f(\neg q_1, \ldots, \neg q_k)$.

2.2.3 Constructing Lattices from Ordered Sets

In this section, we discuss two constructions for deriving complete lattices from sets: the powerset algebra of a set and the downset completion of a poset.

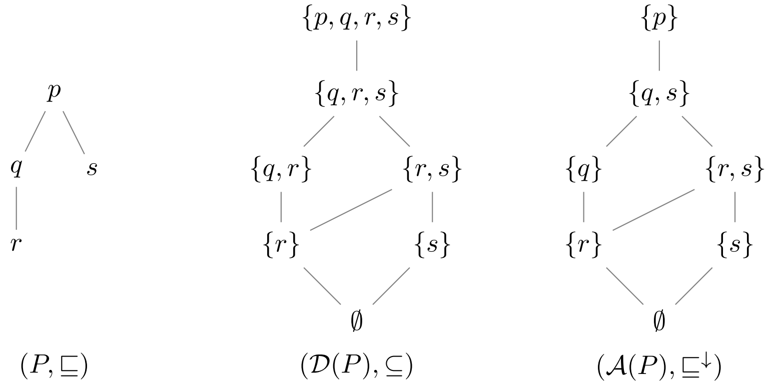


Figure 2.2: A poset (left), its downset completion (center) and antichain lattice (right).

Definition 2.2.19 (Powerset Algebra)**.** The *powerset algebra* of a set S is the complete lattice $(\mathcal{P}(S),\subseteq,\bigcap,\bigcup)$.

If the set S is itself partially ordered, we can complete it in a way that incorporates order information.

Definition 2.2.20 (Downset Completion)**.** The *downset completion* of a poset $(P,\sqsubseteq)$ is the complete lattice $(\mathcal{D}(P),\subseteq,\bigcap,\bigcup)$. The *upset completion* is defined dually.

The downset completion is a sublattice of the powerset algebra that is closed under arbitrary meets and joins. An example is shown in Figure 2.2.

Proposition 2.2.3. *The lattice $\mathcal{D}(P)$ for a poset $(P,\sqsubseteq)$ is complete and distributive.*

In fact, the lattice $\mathcal{D}(P)$ is completely distributive, that is, it satisfies an infinitary variant of the law of distributivity.

In practical terms, an interesting property of downsets is that under certain conditions, they have finite representations as antichains.

Definition 2.2.21 (Antichains)**.** An *antichain* is a subset of poset P such that no pair of elements in Q is comparable. The set of antichains of P is $\mathcal{A}(P)$.

Based on the element ordering $\sqsubseteq$, we can define two partial orders on antichains, the *Hoare ordering* $\sqsubseteq^{\downarrow}$ and the *Smyth ordering* $\sqsubseteq^{\uparrow}$, defined below.

$$A \sqsubseteq^{\downarrow} B \text{ iff } \forall a \in A\ \exists b \in B.\ a \sqsubseteq b \qquad\qquad A \sqsubseteq^{\uparrow} B \text{ iff } \forall b \in B\ \exists a \in A.\ a \sqsubseteq b$$

In general, the antichain posets generated by the above order may form neither lattices nor complete lattices and can therefore not be used to represent downsets. This is the case, for example, when some non-maximal elements of a downset are part of an infinite ascending chain. When the ascending chain condition is met, antichains are order-isomorphic to downsets.

Proposition 2.2.4. *For a poset $(P,\sqsubseteq)$, the function $f : \mathcal{A}(P) \to \mathcal{D}(P)$, $f(A) = \downarrow A$ is an order embedding from the poset $(\mathcal{A}(P),\sqsubseteq^{\downarrow})$ to the downset completion $(\mathcal{D}(P),\subseteq)$.*

Proof. Let $A, B \in \mathcal{A}(P)$ with $A \sqsubseteq^{\downarrow} B$. Then $\forall a \in A\ \exists b \in B\ a \sqsubseteq b$. Therefore, every a in A is in the downset $\downarrow b$ for some $b \in B$, and as a consequence also in $\downarrow B$. It follows that every element of $\downarrow A$ is in $\downarrow B$, therefore $\downarrow A \subseteq \downarrow B$.

Now assume that $\downarrow A \subseteq \downarrow B$ and let $a \in A$. Since $a \in \downarrow B$, there is an element $b \in B$ with $a \sqsubseteq B$. It follows that $A \sqsubseteq^{\downarrow} B$, therefore f is an order embedding. □

Proposition 2.2.5. *If $(P, \sqsubseteq)$ satisfies* ACC*, then $(\mathcal{D}(P), \subseteq)$ and $(\mathcal{A}(P), \sqsubseteq^{\downarrow})$ are order-isomorphic.*

Proof. The function f defined in Proposition 2.2.4 is an order embedding. We show that f is surjective and therefore an order isomorphism, since every set Q in $\mathcal{D}(P)$ can be expressed as $\downarrow A$ for some antichain $A \in \mathcal{A}(P)$. Let R be the set of subset-maximal chains in Q. By ACC, we have that each chain $C \in R$ has a maximum element. Because each chain $C \in R$ is subset-maximal, the set $M = \{c \mid c \text{ is maximum of some } C \in R\}$ is an antichain. It is easy to see that $\downarrow M \subseteq Q$ since M itself is a subset of Q. Further, every element $q \in Q$ is part of some maximal chain C and therefore smaller than the maximum element c of C, which is contained within M. Therefore also $Q \subseteq \downarrow M$. □

2.3 Abstract Interpretation

Abstract interpretation [42, 38, 43, 45] is a mathematical theory of sound approximation and representation. The theory was developed as a mathematical approach to the semantic analysis of computer programs, but the underlying formal framework is more general. As a consequence, the term "abstract interpretation" is both used to describe the mathematical theory and as a short-hand for the varied efforts of applying the theory towards a wide range of program analysis problems. We use the term in the former sense, to denote the theory of sound approximation and representation. This section briefly introduces abstract interpretation from this perspective.

Basics of Sound Approximation Many objects of interest, including the semantics of programs, can be expressed as fixed points of monotone functions on partially ordered sets.

Example 2.3.1. *Consider a program which initializes a variable x to zero and then increases the value of x by two in each execution step. We can give semantics to this program by defining the set of values that x may reach. The set of all values that x takes during execution is an element of the lattice $(\mathcal{P}(\mathbb{Z}), \subseteq)$. Given a set of reachable states, we can extend via the following function application.*

$$f(Q) \mathrel{\hat{=}} \{0\} \cup \{q + 2 \mid q \in Q\}.$$

The set of all values that x may take during program execution is given as follows.

$$Q \mathrel{\hat{=}} \{q \mid q \geq 0 \land q \text{ is even } \}.$$

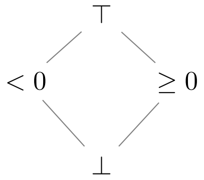
The set Q is the smallest set such that $f(Q)$ is equal to Q.

The set Q in the above example is the least fixed point of f.

Definition 2.3.1 (Fixed Points)**.** A *fixed point* of a function $f : P \to P$ is an element $p \in P$ such that $f(p) = p$. We denote the least fixed point and greatest fixed point, if they exist, respectively, by $\mathsf{lfp}\, X.\ f(X)$ and $\mathsf{gfp}\, X.\ f(X)$ or simply by $\mathsf{lfp}\, f$ and $\mathsf{gfp}\, f$.

The ordered set and functions that model the object of interest form a *concrete domain.* Abstract interpretation provides a systematic methodology for deriving algorithms that provide approximate answers to problems posed in the concrete domain.

Example 2.3.2. *Consider the program from Example 2.3.1. Rather than compute a fixed point in the domain $(\mathcal{P}(\mathbb{Z}), \subseteq)$, we may compute a fixed point in the following lattice A.*



The lattice above can represent information about the sign of the variable x. We associate elements with sets in $\mathcal{P}(\mathbb{Z})$ that they soundly approximate:

$$\begin{aligned} \top &\textit{ approximates every set } Q \subseteq \mathbb{Z} \\ < 0 &\textit{ approximates every set } Q \subseteq \{q \in \mathbb{Z} \mid q \textit{ is negative } \} \\ \geq 0 &\textit{ approximates every set } Q \subseteq \{q \in \mathbb{Z} \mid q \textit{ is zero or positive}\} \\ \bot &\textit{ approximates only the set } \emptyset \end{aligned}$$

Furthermore, we can approximate the concrete function $f : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ with an abstract function $g : A \to A$:

$$\begin{aligned} g(\top) &\mathrel{\hat{=}} \top & g(< 0) &\mathrel{\hat{=}} \top \\ g(\geq 0) &\mathrel{\hat{=}} \geq 0 & g(\bot) &\mathrel{\hat{=}} \geq 0 \end{aligned}$$

In the definition above, $g(\bot)$ is equal to ≥ 0 to indicate that the initial state 0 is always reachable and $g(< 0)$ is equal to $\top$, since from a set of negative numbers, 0 may be reachable.

2.3.1 The Galois Connection Framework

There are a number of frameworks available to formalize the notion of sound approximation, including soundness relations, closure operators and Galois connections [45]. In this dissertation, we use the Galois connection framework, in which a *concrete lattice* $(C, \preceq)$ is related to an *abstract lattice* $(A, \sqsubseteq)$ via two functions: an *abstraction function* $\alpha : C \to A$, which maps concrete elements to their best abstract representation, and a *concretization function* $\gamma : C \to A$, which maps abstract elements to the weakest elements that they soundly approximate. We first consider overapproximation, where we consider an abstract element $a \in A$ a sound approximation of a concrete element $c \in C$ if $\gamma(a)$ is greater than c. We then discuss how this discussion dualises to underapproximation.

We first introduce the lattice of intervals, which serve as an example for Galois-connection based approximation.

Definition 2.3.2 (Interval Lattice)**.** Let $\mathbb{Z}_\infty$ denote the set $\mathbb{Z} \cup \{-\infty, \infty\}$ where for all $n \in \mathbb{Z}$, it holds that $-\infty \leq n \leq \infty$. The set of *intervals* is $\mathit{Itv} \mathrel{\hat{=}} \{(l, u) \in \mathbb{Z}_\infty^2 \mid l \leq u\} \cup \{\bot\}$. The element $\bot$ is the *empty interval*. We use square brackets $[l, u]$ to denote an interval (l, u). We refer to l as the *lower bound* $\mathit{lb}(i)$ and to u as the *upper bound* $\mathit{ub}(i)$.

The *interval lattice* is given as follows.

$$(\mathit{Itv}, \sqsubseteq, \sqcap, \sqcup)$$

$$i \sqsubseteq i' \text{ exactly if } i = \bot \text{ or if both } \mathit{lb}(l) \geq \mathit{lb}(l') \text{ and } \mathit{ub}(i) \leq \mathit{ub}(i')$$

$$\bot \sqcap i \mathrel{\hat{=}} \bot \quad i \sqcap i' \mathrel{\hat{=}} \rho(\max\{\mathit{lb}(i), \mathit{lb}(i')\}, \min\{\mathit{ub}(i), \mathit{ub}(i')\})$$

$$\bot \sqcup i \mathrel{\hat{=}} i \quad i \sqcup i' \mathrel{\hat{=}} \rho(\min\{\mathit{lb}(i), \mathit{lb}(i')\}, \max\{\mathit{ub}(i), \mathit{ub}(i')\})$$

$$\text{where } \rho([l, u]) \mathrel{\hat{=}} \begin{cases} \bot & l > u \\ [l, u] & \text{otherwise} \end{cases}$$

The interval lattice is depicted in Figure 2.3. An interval $[l, u]$ can be viewed as representing the set $\{x \in \mathbb{Z} \mid l \leq x \leq u\}$. The empty interval $\bot$ represents the empty set. We can define a concretization function $\gamma : \mathit{Itv} \to \mathcal{P}(\mathbb{Z})$ which maps an interval to the set it represents.

Clearly, not every set of integers is represented by an interval. On the other hand, every set of integers included in some unique, maximally precise interval, which we call the *best approximation* of that set. The best approximation of the empty set is the empty interval $\bot$. For a non-empty set $S \subseteq \mathbb{Z}$, the best approximation is given by $[\min(S), \max(S)]$. The approximation function $\alpha : \mathcal{P}(\mathbb{Z}) \to \mathit{Itv}$ maps a set to its best approximation as an interval.

The existence of best approximations constitutes a special case, which is formalized by the Galois connection framework.

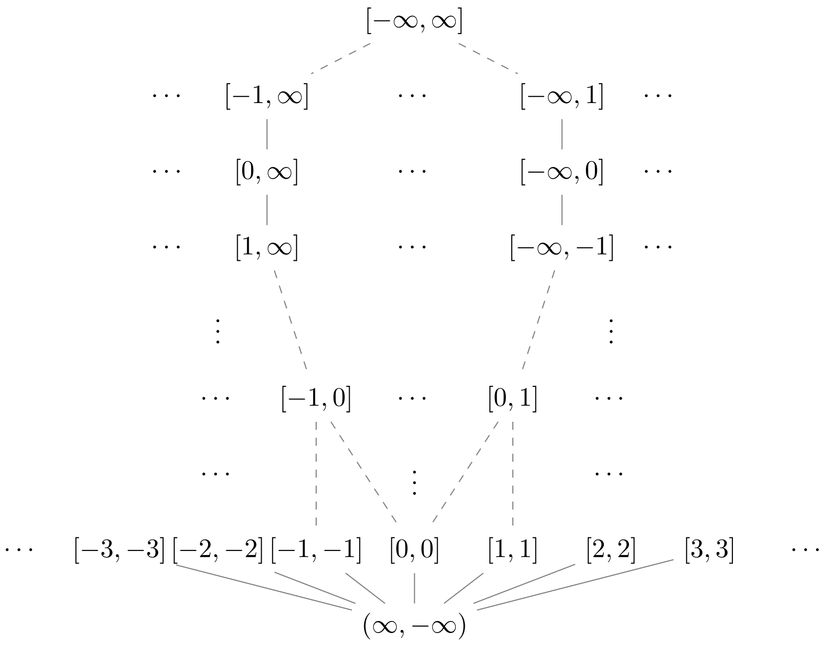


Figure 2.3: Lattice of intervals.

Definition 2.3.3 (Galois Connection)**.** Let $(C, \preceq)$ and $(A, \sqsubseteq)$ be posets. A pair of order-preserving functions $(\alpha : C \to A, \gamma : A \to C)$ is a *Galois connection* with *lower adjoint* α and *upper adjoint* γ, denoted

$$(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq),$$

if it holds for all $c \in C$ and $a \in A$ that

$$\alpha(c) \sqsubseteq a \text{ exactly if } c \preceq \gamma(a).$$

Abstraction and concretization function for the intervals form a Galois connection.

Proposition 2.3.1. *The pair* (α, γ) *defined below forms a Galois connection.*

$$(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathit{Itv}, \sqsubseteq)$$

$$\begin{array}{rr} \alpha(\emptyset) \mathrel{\hat{=}} \bot & \alpha(S) \mathrel{\hat{=}} [\min(S), \max(S)] \\ \gamma(\bot) \mathrel{\hat{=}} \emptyset & \gamma([l, u]) \mathrel{\hat{=}} \{s \in \mathbb{N} \mid l \leq s \leq u\} \end{array}$$

We now state some basic properties of Galois connections.

Proposition 2.3.2. *For a Galois connection* $(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$*, the following properties hold.*

(i) The lower adjoint α *uniquely determines the upper adjoint* γ*, and conversely, the upper adjoint* γ *uniquely determines the lower adjoint* α*.*

(ii) α *is completely additive.*

(iii) γ *is completely multiplicative.*

(iv) Galois connections compose, i.e.,

$$\text{if } A \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} B \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} C, \text{ then } A \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} C.$$

In the case where the abstract and the concrete are complete lattices, any completely multiplicative function between the two lattices induces a Galois connection. Informally, this is the case because we can define the best approximation of an element as the meet of all of its approximations.

Proposition 2.3.3. *Let $(C, \preceq)$ and $(A, \sqsubseteq)$ be complete lattices. If there is a completely multiplicative function $\gamma : A \to C$, then γ is the upper adjoint of a Galois connection with lower adjoint $\alpha : C \to A$ as below.*

$$\alpha(c) = \bigsqcap \{a \in A \mid c \preceq \gamma(a)\}$$

Dually, if there is a completely additive function $\alpha : C \to A$, then α is the lower adjoint of a Galois connection with upper adjoint $\gamma : A \to C$ as below.

$$\gamma(a) = \bigvee \{c \in C \mid \alpha(c) \sqsubseteq a\}$$

2.3.2 Closure Operators

Closure operators are special order-preserving functions. In Chapter 3, we generalize the satisfiability problem to a decision problem over closure operators.

Definition 2.3.4 (Closure Operators)**.** An *upper closure operator* on a poset $(P, \sqsubseteq)$ is an order-preserving function $\xi : P \to P$ that is (i) *inflationary*, i.e., for all $p \in P$, $p \sqsubseteq \xi(p)$ and (ii) *idempotent*, i.e., for all $p \in P$, $\xi \circ \xi(p) = \xi(p)$. A *lower closure operator* is dually defined as an order-preserving and idempotent function that is *deflationary*, i.e., for all $p \in P$, $\xi(p) \sqsubseteq p$.

Closure operators provide an alternate characterization of Galois connections.

Proposition 2.3.4. *For two posets $(C, \preceq)$ and $(A, \sqsubseteq)$ and order-preserving functions $\alpha : C \to A$ and $\gamma : A \to C$ it holds that $(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ exactly if $\gamma \circ \alpha$ is an upper closure operator and $\alpha \circ \gamma$ is a lower closure operator.*

2.3.3 Over- and Underapproximation

Our discussion so far has focused on overapproximation. We can dualise the notion of approximation described so far to obtain underapproximation. Chapters 4 and 5 will discuss logical satisfiability algorithms that simultaneously utilize overapproximations and underapproximations.

Definition 2.3.5 (Over- and Underapproximation)**.** For posets $(C, \preceq)$ and $(A, \sqsubseteq)$ and w.r.t. (α, γ), we define that

(i) *A overapproximates C* w.r.t. (α, γ) if $(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ holds,

(ii) *A underapproximates C* w.r.t. (α, γ) if $(C, \succeq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsupseteq)$ holds and

(iii) *A approximates C* w.r.t. (α, γ) if A overapproximates C or if A underapproximates C.

For an underapproximation, this means that $\gamma \circ \alpha$ is a lower closure, that is, moving to the abstract and back to the concrete returns a smaller element. Similarly, $\alpha \circ \gamma$ is an upper closure, i.e., moving to the concrete and back to the abstract may return a property that is strictly more general than the original one but describes the same concrete element.

2.3.4 Approximating Transformers

Monotone functions on ordered sets are central enough in abstract interpretation to deserve their own name.

Definition 2.3.6 (Transformers)**.** A *transformer* on a poset $(P, \sqsubseteq)$ is a order-preserving function $f : P \to P$.

We now define a notion of approximation for transformers.

Definition 2.3.7 (Sound Abstract Transformers)**.** Let $(C, \preceq)$ and $(A, \sqsubseteq)$ be two posets with transformers $f : C \to C$ and $g : A \to A$. The transformer g is a *sound approximation* of f w.r.t. (α, γ) if one of the following conditions holds.

- A overapproximates C w.r.t. (α, γ) and for all $a \in A$, $\gamma \circ g(a) \succeq f \circ \gamma(a)$.
- A underapproximates C w.r.t. (α, γ) and for all $a \in A$, $\gamma \circ g(a) \preceq f \circ \gamma(a)$.

The above notion of approximation can be equivalently expressed in the Galois connection framework.

Proposition 2.3.5. *Let $(C, \preceq)$ be overapproximated by $(A, \sqsubseteq)$ w.r.t. (α, γ). Then the pointwise lifted poset $(C \to C, \preceq)$ is overapproximated by the pointwise lifted poset $(A \to A, \sqsubseteq)$ w.r.t. (α_T, γ_T) where $\alpha_T(f) \mathrel{\hat{=}} \alpha \circ f \circ \gamma$ and $\gamma_T(g) \mathrel{\hat{=}} \lambda g.\gamma \circ g \circ \alpha$.*

The dual statement holds for underapproximation. We can now see that our characterization of sound approximations of transformers can be characterized in terms of the Galois connection (α_T, γ_T).

Proposition 2.3.6. *Given the Galois connection $(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, the transformer $g : A \to A$ soundly approximates $f : C \to C$ exactly if $\alpha_T(f) \sqsubseteq g$ for $\alpha_T(f) = \alpha \circ f \circ \gamma$.*

Since the Galois connection (α, γ) extends to the transformer lattices $C \to C$ and $A \to A$, we may conclude that every concrete transformer $f : C \to C$ has a unique best sound approximation $\alpha_T(f)$ that provides maximal precision. We will define this object separately due to its importance.

Definition 2.3.8 (Best Abstract Transformer)**.** Let $(C, \preceq)$ be approximated by $(A, \sqsubseteq)$ w.r.t. (α, γ), and let $f : C \to C$ and $g : A \to A$ be two transformers. The *best abstract transformer* of f is $\alpha \circ f \circ \gamma$.

Throughout this dissertation, we will be working in the canonical abstract interpretation setting of Galois connections between complete lattices. A domain, in the abstract interpretation sense, is then a complete lattice together with a set of transformers.

Definition 2.3.9 (Concrete and Abstract Domains)**.** The tuple $(C, \preceq, \curlywedge, \curlyvee, \{f_i : C \to C\}_{i \in I})$ is the *concrete domain* and $(A, \sqsubseteq, \sqcap, \sqcup, \{g : A \to A\}_{i \in I})$ the *abstract domain* w.r.t. (α, γ) if the following hold.

(i) $(C, \preceq, \curlywedge, \curlyvee)$ and $(A, \sqsubseteq, \sqcap, \sqcup)$ are complete lattices,

(ii) A approximates C w.r.t. the Galois connection (α, γ) and

(iii) for all $i \in I$, the transformer g_i soundly approximates the transformer f_i.

In the above, $(C, \preceq, \curlywedge, \curlyvee)$ is the *concrete lattice* and $(A, \sqsubseteq, \sqcap, \sqcup)$ is the *abstract lattice.* If we do not explicitly define the Galois connection, then abstraction and concretization function are denoted α and γ. We assume throughout this dissertation that $\gamma(\bot_A) = \bot_C$ and that $\gamma(\top_A) = \top_C$. This is not true in general, but it is simple to extend any abstract domain to satisfy this property.

2.3.5 Semantic Reduction and Reduced Products

We now recall some concepts from abstract-interpretation theory that relate to the representation of abstract elements: semantic reduction and reduced products [43]. Both of these ideas will be important in Chapter 4, where we give abstract interpretation accounts of satisfiability procedures.

In an abstract domain, two distinct abstract element a and b may concretize to the same concrete element, that is, it may hold that $a \neq b$ but $\gamma(a) = \gamma(b)$. When this is not the case and $a \neq b$ implies that $\gamma(a) \neq \gamma(b)$, then the pair (α, γ) is a Galois insertion.

Definition 2.3.10 (Galois Insertion)**.** A *Galois insertion* is a Galois connection $(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ such that $\alpha \circ \gamma$ is the identity function.

The Galois insertion setting formalizes the case where there is no redundancy in the abstraction. If redundancy is present during an analysis certain difficulties may arise. For example, it can become non-trivial to decide whether two abstract elements have the same concretization or whether a given abstract element represents the empty set. In such cases, reduction operators can be applied to make an element more precise without changing the semantics of the element.

Definition 2.3.11 (Reduction Operator)**.** A *reduction operator* for a Galois connection $(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ is a deflationary transformer $\rho : A \to A$. A reduction is *sound* if for all $a \in A$ it holds that $\gamma(\rho(a)) = \rho(a)$.

The *best reduction operator* is the transformer $\alpha \circ \gamma$, which first determines the concrete meaning of the abstract element and then finds the best abstract representation.

We now introduce product lattices. The product operation between lattices lifts to a combination operation between domains [43]. In practical terms, the product of two abstract domains represents the result of running two analyses in parallel.

Definition 2.3.12 (Cartesian Product)**.** Consider two lattices $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ with

$$C \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} A \quad \text{and} \quad C \underset{\alpha_B}{\overset{\gamma_B}{\leftrightarrows}} B.$$

The Cartesian product of A and B is the following lattice.

$$(A \times B, \sqsubseteq_{A\times B}, \sqcap_{A\times B}, \sqcup_{A\times B})$$
$$(a, b) \sqsubseteq_{A\times B} (a', b') \iff a \sqsubseteq_A a' \wedge b \sqsubseteq_B b'$$
$$(a, b) \sqcup_{A\times B} (a', b') \mathrel{\hat{=}} (a \sqcup_A a', b \sqcup_B b') \quad (a, b) \sqcap_{A\times B} (a', b') \mathrel{\hat{=}} (a \sqcap_A a', b \sqcap_B b')$$

The abstraction and concretization functions are defined as follows.

$$\alpha_{A\times B}(c) \mathrel{\hat{=}} (\alpha_A(c), \alpha_B(c)) \quad \gamma_{A\times B}(c) \mathrel{\hat{=}} \gamma_A(c) \cap \gamma_B(c))$$

Proposition 2.3.7. *Let $\alpha_{A\times B}$ and $\gamma_{A\times B}$ be defined as in Definition 2.3.12. The two functions form a Galois connection as follows.*

$$C \underset{\alpha_{A\times B}}{\overset{\gamma_{A\times B}}{\leftrightarrows}} A \times B$$

Product construction may introduce redundancy as the following example will illustrate.

Example 2.3.3. *Consider the* abstract parity lattice $(Par, \sqsubseteq_{Par})$ *defined below.*

$$Par \mathrel{\hat{=}} \{\bot, \mathsf{even}, \mathsf{odd}, \top\} \qquad a \sqsubseteq_{Par} b \textit{ exactly if } a = \bot,\ b = \top \textit{ or } a = b$$

The lattice $(Par, \sqsubseteq_{Par})$ approximates the lattice $(\mathcal{P}(\mathbb{Z}), \subseteq)$ with the concretization function $\gamma_{Par} : Par \to \mathcal{P}(\mathbb{Z})$. This function is defined as follows.

$$\gamma_{Par}(\bot) \mathrel{\hat{=}} \emptyset \qquad \gamma_{Par}(\mathsf{even}) \mathrel{\hat{=}} \{x \in \mathbb{Z} \mid x \textit{ is even }\}$$

$$\gamma_{Par}(\top) \mathrel{\hat{=}} \mathbb{Z} \qquad\qquad \gamma_{Par}(\mathsf{odd}) \mathrel{\hat{=}} \{x \in \mathbb{Z} \mid x \textit{ is odd } \}$$

The Cartesian product lattice $Par \times Itv$ approximates the concrete lattice $\mathcal{P}(\mathbb{Z})$ using both range and parity information. For example, the concretization function $\gamma_{Par\times Itv}$ of the product maps the pair $(\mathsf{even}, [0, 6])$ to the set $\{0, 2, 4, 6\}$. Note that abstraction and concretization functions for the interval and parity lattice are Galois insertions, i.e., in each lattice, no two distinct abstract elements represent the same element. This is not the case for the product. Consider the following two elements and their concretization.

$$p \mathrel{\hat{=}} (\mathsf{even}, [0, 5]) \quad q \mathrel{\hat{=}} (\mathsf{even}, [0, 4])$$
$$\gamma_{Par\times Itv}(p) = \gamma_{Par\times Itv}(q) = \{0, 2, 4\}$$

Reduction can be used to map p to the semantically equivalent but more precise element q.

2.3.6 Existence and Sound Approximation of Fixed Points

Transformers on complete lattices are guaranteed to have fixed points. Moreover, the set of fixed points form a complete sub-lattice.

Theorem 2.3.8 (Knaster-Tarski Theorem [164, 110])**.** *Let L be a complete lattice and $f : L \to L$ be an order-preserving function. Then the set of fixed points of f in L is also a complete lattice, with a least fixed point $\mathsf{lfp}\, f$ and a greatest fixed point $\mathsf{gfp}\, f$ characterized as below.*

$$\mathsf{lfp}\, f = \bigsqcap\{a \in L \mid f(a) \sqsubseteq a\} \qquad \mathsf{gfp}\, f = \bigsqcup\{a \in L \mid a \sqsubseteq f(a)\}$$

We are now ready to state the central soundness result of abstract interpretation, which states that concrete fixed points can be soundly computed in the abstract.

Theorem 2.3.9 (Fixed Point Transfer [38, 43])**.** *Let $(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ be a Galois connection between complete lattices. If $f : C \to C$ is soundly approximated by $g : A \to A$, then $\mathsf{lfp}\, f \preceq \gamma(\mathsf{lfp}\, g)$ and $\mathsf{gfp}\, f \preceq \gamma(\mathsf{gfp}\, g)$.*

Computing Fixed Points Precisely and Approximately

We now address the question of how fixed points can be computed in the abstract. If a transformer preserves certain joins or meets this can be accomplished via an iterative procedure. We specialize the following statements for lattices.

Definition 2.3.13 (Kleene Iteration Sequence)**.** Let f be a function on a poset P. The *upwards Kleene iteration sequence* of f is given by $F_i^{\bot} \mathrel{\hat{=}} f^i(\bot)$ where $f^0(p) = p$ and $f^{i+1}(p) = f(f^i(p))$. The *downwards Kleene iteration sequence* f $(F_i^{\top})_{i\in\mathbb{N}_0}$ is defined dually.

To state the conditions under which Kleene iteration leads to a fixed point, we need to introduce some further notions.

Definition 2.3.14 (Directed Sets and Continuous Functions)**.** A subset S of a partially ordered set $(P, \sqsubseteq)$ is *directed* if every pair of elements in S has an upper bound in S. A function $f : P \to Q$ between partially ordered sets $(P, \sqsubseteq)$ and $(Q, \preceq)$ is *Scott-continuous* if for every directed set $S \subseteq Q$ with join $\bigsqcup S$ it holds that $\bigcurlyvee_{s\in S} f(s)$ is defined and $f(\bigsqcup S) = \bigcurlyvee_{s\in S} f(s)$.

Scott-continuous functions are order-preserving and therefore have least fixed points on complete lattices. These fixed points can be computed as the limit of the Kleene iteration sequence.

Theorem 2.3.10 (Kleene's Fixed Point Theorem [109] for Complete Lattices)**.** *Let L be a complete lattice and $f : L \to L$ be a Scott-continuous function. Then $\mathsf{lfp}\, f$ exists and is equal to $\bigsqcup_{i\in\mathbb{N}_0} F_i^{\bot}$.*

The statement dualises for greatest fixed points.

If the underlying poset satisfies the ascending chain condition (ACC), then the upwards Kleene iteration sequence converges in finitely many steps. Under the assumption that f is computable, the fixed point of f can be computed by iterating the function f finitely many times until the result converges. If ACC is not satisfied, or if one is willing to sacrifice precision for increased efficiency, widening and narrowing [42] operators may be used to accelerate the computation. We first define two related *acceleration operators* which we call extrapolation and interpolation and then define widenings and narrowings as restrictions of these operators with additional convergence properties. Widening and narrowing have been defined in different ways throughout the literature. We adapt the definitions from [45] and specialize them to complete lattices.

Extrapolation aims to jump ahead of the trajectory of an ascending iteration sequence and can be viewed as an overapproximation of the join.

Definition 2.3.15 (Extrapolation)**.** An *upwards extrapolation* operator is a function $\nabla_\uparrow : \mathcal{P}(L) \to L$ on a complete lattice $(L, \sqsubseteq, \sqcap, \sqcup)$ such that for any $Q \subseteq L$ it holds that

$$\forall q \in Q.\ q \sqsubseteq \bigsqcup Q \sqsubseteq \nabla_\uparrow Q.$$

Clearly it is always the case that $\forall q \in Q.\ q \sqsubseteq \bigsqcup Q$ holds. The definition above is redundant, but shows the relation to interpolation. Interpolation may be used to accelerate a descending iteration sequence, and corresponds to an overapproximation of the meet.

Definition 2.3.16 (Interpolation)**.** A *downwards interpolation* operator on a complete lattice $(L, \sqsubseteq, \sqcap, \sqcup)$ is a function $\Delta_\downarrow : \mathcal{P}(L) \to L$ such that for any non-empty set $Q \subseteq L$, the following holds.

$$\exists q \in Q.\ \bigsqcap Q \sqsubseteq \Delta_\downarrow Q \sqsubseteq q$$

Instead of the set-based operators with signature $\mathcal{P}(L) \to L$ defined above we will sometimes use unary or binary versions of these operators. A unary extrapolation is a function $\nabla_\uparrow : L \to L$ such that the function $\lambda\{a\}.\ \nabla_\uparrow a$ is an extrapolation operator that is partially defined for arguments of cardinality one. For binary operators, we use infix notation, e.g., $a \,\Delta_\downarrow\, b$ for $\Delta_\downarrow(a, b)$. Binary extrapolation is a function $\nabla_\uparrow\, L \times L \to L$ such that the function $\lambda\{a, b\}.\ a \nabla_\uparrow b$ is an extrapolation operator that is partially defined for arguments of cardinality 1 or 2. Binary interpolation is defined similarly.

Binary interpolation satisfies the following property which coincides with another common definition of interpolation.

Proposition 2.3.11 (Binary Interpolation)**.** *Let $\Delta_\downarrow$ be a binary interpolation operator over a complete lattice L. Then for $a, b \in L$ the following statement holds.*

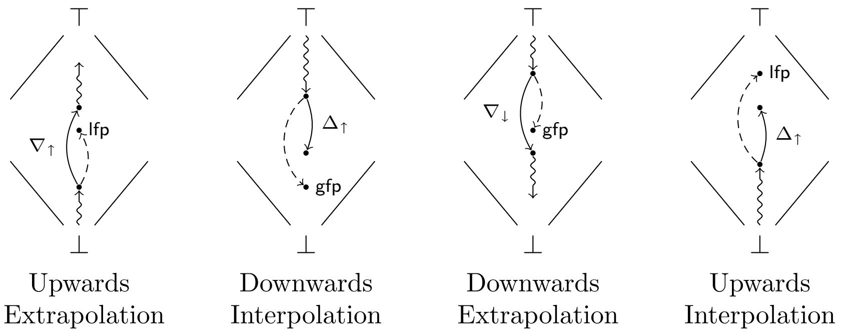
$$a \sqsupseteq b \implies a \sqsupseteq a \,\Delta_\downarrow\, b \sqsupseteq b$$

Extrapolation and interpolation can be used to approximate Kleene iteration sequences. This involves constructing an iteration sequence in which function application is alternated with application of the interpolation and extrapolation operators. Different applications of interpolation and extrapolation are possible depending on the number of iterates that are considered when applying the operators. We consider only the case where two iterates are considered and binary operators suffice. Considering more than the above number of iterates can be viewed as a practical refinement.

Definition 2.3.17 (Iteration Sequence with Extrapolation)**.** Given a transformer $f : L \to L$ on a complete lattice L and an upwards extrapolation $\nabla_\uparrow$, the *upwards iteration sequence with extrapolation* is given as follows.

$$F_0^{\nabla_\uparrow} = \bot \qquad F_{i+1}^{\nabla_\uparrow} \mathrel{\hat{=}} F_i^{\nabla_\uparrow} \nabla_\uparrow f(F_i^{\nabla_\uparrow})$$

Downwards extrapolation $\nabla_\downarrow$ and the *downwards iteration with extrapolation* are defined dually.



Upwards Extrapolation

Downwards Interpolation

Downwards Extrapolation

Upwards Interpolation

Figure 2.4: Extrapolation and interpolation.

Definition 2.3.18 (Iteration Sequence with Interpolation)**.** For a transformer $f : L \to L$ on a complete lattice L and a downwards interpolation $\Delta_\downarrow$, the *downwards iteration sequence with interpolation* is given as follows.

$$F_0^{\Delta_\downarrow} \mathrel{\hat{=}} \top \qquad F_{i+1}^{\Delta_\downarrow} \mathrel{\hat{=}} F_i^{\Delta_\downarrow} \, \Delta_\downarrow \, f(F_i^{\Delta_\downarrow})$$

Upwards interpolation and the *upwards iteration with interpolation* are defined dually.

Iteration sequences with interpolation and extrapolation are depicted in Figure 2.4. The use of extrapolation and interpolation may lead to finite convergence of an otherwise infinite iteration sequence, but this is not guaranteed. In order to guarantee finite convergence, stronger constraints need to be imposed on the operators. The following definitions are adapted from [45].

Definition 2.3.19 (Widening)**.** A *upwards widening* is an upwards extrapolation $\nabla_\uparrow$ such that for any increasing sequence $s_0 \sqsubseteq s_1 \sqsubseteq \ldots$ the sequence $a_i = \nabla_\uparrow$ converges in a finite number of steps. *Downwards widening* is defined dually.

Definition 2.3.20 (Narrowing)**.** A *downwards narrowing* is a downwards interpolation $\Delta_\downarrow$ such that for any sequence $(s_i)_{i \in \mathbb{N}_0}$ the sequence $a_i = \nabla_\uparrow\{a_0, \ldots, a_{i-1}, s_i\}$ converges in a finite number of steps. *Upwards narrowing* is defined dually.

The use of widening and narrowing allows fixed points to be approximated by finite iteration. We specialize the soundness results from [45] to the case of complete lattices.

Theorem 2.3.12 (Soundness of Widening and Narrowing [45])**.** *Let $f : L \to L$ be a transformer on a complete lattice $(L, \sqsubseteq)$, let $\nabla_\uparrow$ be an upwards widening and $\nabla_\downarrow$ be a downwards narrowing, then the sequences $(F_i^{\nabla_\uparrow})_{i \in \mathbb{N}_0}$ and $(F_i^{\Delta_\downarrow})_{i \in \mathbb{N}_0}$ converge in a finite number of steps and the following holds.*

$$\mathsf{lfp}\, f \sqsubseteq \bigsqcup_{i \in \mathbb{N}_0} F_i^{\nabla_\uparrow} \qquad\qquad \mathsf{gfp}\, f \sqsubseteq \bigsqcap_{i \in \mathbb{N}_0} F_i^{\Delta_\downarrow}$$

2.3.7 Precision and Completeness

Abstract interpretation in the Galois connection framework is sound by design. On the other hand, the result of an abstract computation may lose precision compared to the concrete result which can make it impossible to determine certain properties of interest. For this section, fix $g : A \to A$ to be a sound approximation of $f : C \to C$ w.r.t. the Galois connection (α, γ) between the posets $(C, \preceq)$ and $(A, \sqsubseteq)$.

There are three possible sources of imprecision when approximating the concrete fixed point $\mathsf{lfp}\, f$ by computing the abstract iteration sequence $(G_i^{\nabla_\uparrow})_{i \in \mathbb{N}_0}$.

- The abstract transformer g may be less precise than the best approximation $\alpha \circ f \circ \gamma$.
- The abstract lattice may not be precise enough to capture the same information as the concrete, even when best abstract transformers are used.
- The use of acceleration operators such as extrapolation and interpolation may lose precision.

In logic, completeness notions characterize deductive systems with respect to their ability to derive certain statements. Similarly, completeness notions in abstract interpretation characterize the precision that can be obtained by applying an abstract transformer, compared to applying its concrete counterpart.

Definition 2.3.21 (Completeness of Transformers)**.** We define for $a \in A$ and $c \in C$ that that g is

- γ-complete at a if $\gamma \circ g(a) = f \circ \gamma(a)$ [68] and
- α-complete at c if $g \circ \alpha(c) = \alpha \circ f(c)$ [43, 70].

The transformer g is complete for one of the completeness notions above if it is complete at every element.

The completeness notions above are independent. An abstract transformer can be γ-complete but not α-complete. This is the case when the abstract transformer loses no precision compared to the concrete, but returns elements that are not best representations. On the other hand a transformer that is α-complete but not γ-complete returns best representations, but may still lose precisions.

2.3.8 Abstractly Interpreting Programs

Abstract interpretation was originally introduced as an approach for semantic analysis of computer programs. Program semantics can be characterized as a fixed point on a complete lattice. Approximate semantics can be obtained via abstract interpretation and used to soundly decide program properties such as correctness.

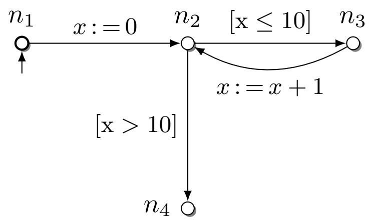
Control-Flow Graphs and State Transition Systems

We model programs as *control-flow graphs* (CFG). A CFG is a directed graph where nodes represent control locations and edges between nodes are labeled with program statements. An execution of a program describes a path through the control-flow graph starting at a unique initial control location; each edge traversal corresponds to a transformation of the program state that respects the statement associated with the edge. Figure 2.5 shows an example of a CFG for a simple program that counts from 0 to 10. The initial node n_1 is marked by an incoming edge with no source node.

Definition 2.3.22 (Control-Flow Graph)**.** A *control-flow graph* (CFG) is a tuple (N, E, n_I, st), where N is the set of *control locations*, $E \subseteq N \times N$ is the set of *control-flow edges*, $n_I \in N$ is the *initial location* and $st : E \to \mathcal{P}$ is a labeling of edges with *program statements*.

A *path* through a control-flow graph is a finite, non-empty sequence of control-locations $n_1 \dots n_k$, such that there is an edge $(n_i, n_{i+1}) \in E$ for all $1 \leq i \leq k-1$. A path $n_1 n_2 \dots n_k$ is *initialized* if $n_1 = n_I$. Every program execution describes an initialized path, but not every initialized path corresponds to some program execution.

We assume that each statement $s \in \mathcal{P}$ is associated with a transition relation $T_s^{\Omega} \subseteq \Omega \times \Omega$ over a set of *memory states* Ω. We can define the operational semantics of a CFG in terms of a state transition system.



Control locations	$N \mathrel{\hat{=}} \{n_1, n_2, n_3, n_4\}$
Control-flow edges	$E \mathrel{\hat{=}} \{(n_1, n_2), (n_2, n_3), (n_2, n_4), (n_3, n_2)\}$
Initial location	$n_i \mathrel{\hat{=}} n_1$
Labelling	$st \mathrel{\hat{=}} \{(n_1, n_2) \mapsto x := 0, \ldots\}$
Memory States	$M \mathrel{\hat{=}} (\{x\} \mapsto \mathbb{N})$
States	$S \mathrel{\hat{=}} N \times (\{x\} \mapsto \mathbb{N})$

Figure 2.5: A simple program represented by a control-flow graph.

Definition 2.3.23 (State Transition System)**.** A *state transition system* (or *transition system*) is a tuple $(\Sigma, \rightarrow, I)$ where Σ is a set of *states*, $\rightarrow \;\subseteq \Sigma \times \Sigma$ is a *transition relation* and $I \subseteq \Sigma$ is a set of initial states.

For $(\sigma, \sigma') \in \rightarrow$ we write $\sigma \rightarrow \sigma'$ and call σ a *predecessor* of σ' and σ' a *successor* of σ. We denote by $\rightarrow^*$ the reflexive, transitive closure of $\rightarrow$. If $\sigma \rightarrow^* \sigma'$ then we say σ' is *reachable* from σ. We denote the reverse transition by $\leftarrow$, given as $b \leftarrow a$ exactly if $a \rightarrow b$. For a CFG $G = (N, E, n_I, st)$ we define below a corresponding state transition system $(\Sigma_G, \rightarrow_G, I_G)$ in which states are pairs of control locations and memory states.

$$\begin{aligned}
\Sigma_G &= (N \times \Omega) \\
\rightarrow_G &= \{((n_1, \omega_1), (n_2, \omega_2)) \mid (n_1, n_2) \in E \text{ and } (\omega_1, \omega_2) \in T^{\Omega}_{st(n_1,n_2)}\} \\
I_G &= \{(n_I, \omega) \mid \omega \in \Omega\}
\end{aligned}$$

Transition systems can be given a semantics in terms of the sets of states that are reachable from the initial state.

Definition 2.3.24 (State Semantics [47])**.** The *state semantics* $\|(\Sigma, \rightarrow, I)\| \subseteq \Sigma$ of a transition system $(\Sigma, \rightarrow, I)$ is the set of states σ reachable from some initial state:

$$\|(\Sigma, \rightarrow, I)\| \mathrel{\hat{=}} \{\sigma \mid \exists i \in I.\ i \rightarrow^* \sigma\}$$

The *state semantics* $\|G\|$ of a CFG G is the state semantics of its associated transition system $(\Sigma_G, \rightarrow_G, I_G)$.

State semantics are called reachability semantics in [47].

We now discuss semantics in terms of execution traces. A well-formed trace through a state transition system is a sequence of states that starts in some initial state and respects the transition relation. We do not consider infinite traces.

Definition 2.3.25 (Trace)**.** A *trace* π through a state transition system $\mathcal{T} = (\Sigma, \rightarrow, I)$ is a finite sequence of states. The set of all traces is $\Pi \mathrel{\hat{=}} \Sigma^+$. A trace $\pi = \pi_1 \ldots \pi_k \in \Pi$

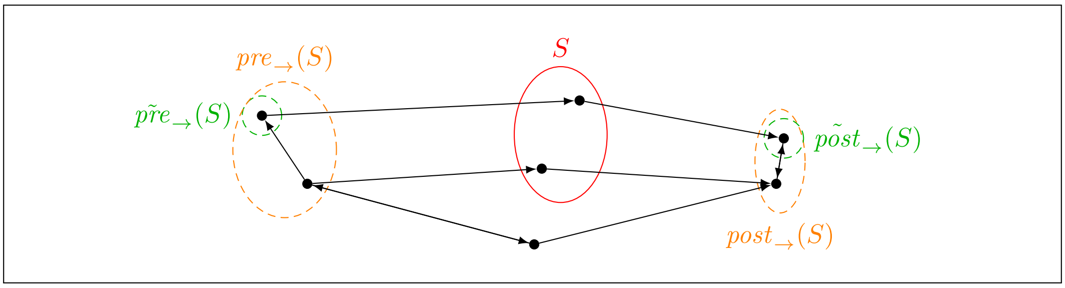

$pre_{\rightarrow}(S)$
S
$\widetilde{pre}_{\rightarrow}(S)$
$\widetilde{post}_{\rightarrow}(S)$
$post_{\rightarrow}(S)$

Figure 2.6: State transformers.

respects $\rightarrow$ if $\pi_1 \rightarrow \pi_2 \rightarrow \ldots \rightarrow \pi_k$ and is *initialized* if $\pi_1 \in I$. A trace is *well-formed* if it is initialized and respects $\rightarrow$. A (well-formed) trace over a CFG G is a (well-formed) trace over its state transition system $(\Sigma_G, \rightarrow_G, I_G)$.

Definition 2.3.26 (Trace Semantics [47])**.** The *trace semantics* $\|(\Sigma, \rightarrow, I)\|^* \subseteq \Pi$ of a transition system $(\Sigma, \rightarrow, I)$ is the set of its well-formed traces. The *trace semantics* $\|G\|^*$ of a CFG G is the trace semantics of its associated transition system $(\Sigma_G, \rightarrow_G, I_G)$.

Transformer and Fixed Point Semantics

In order to apply abstract interpretation to programs, we specify the semantics of a transition system in terms of fixed points over complete lattices. We first define classic transformers over the powerset lattice of states $\mathcal{P}(\Sigma)$. These notions are standard in the literature.

Definition 2.3.27 (State Transformers)**.** For a transition relation $\rightarrow \subseteq \Sigma \times \Sigma$, we define the following transformers over $\mathcal{P}(\Sigma)$.

Strongest Postcondition (Existential Postcondition)

$$post_{\rightarrow}(S) \mathrel{\hat{=}} \{\sigma' \mid \exists \sigma \in \Sigma.\ \sigma \in S \wedge \sigma \rightarrow \sigma'\}$$

Weakest Precondition (Universal Precondition)

$$\widetilde{pre}_{\rightarrow}(S) \mathrel{\hat{=}} \{\sigma \mid \forall \sigma' \in \Sigma.\ \sigma' \in S \vee \sigma \not\rightarrow \sigma'\}$$

Universal Postcondition

$$\widetilde{post}_{\rightarrow}(S) \mathrel{\hat{=}} \widetilde{pre}_{\leftarrow}(S)$$

Existential Precondition

$$pre_{\rightarrow}(S) \mathrel{\hat{=}} post_{\leftarrow}(S)$$

The above are classic transformers defined by transition systems. An example of applying these transformers is shown in Figure 2.6. The strongest postcondition maps a set of states S to the set of all successor states that may be reached in one step by starting from an element $\sigma \in S$, whereas the weakest precondition maps S to the set of all states which can *only* reach elements of S (if any).

We will sometimes omit the subscript and write *post* instead of $post_{\rightarrow}$, $\widetilde{post}$ instead of $\widetilde{post}_{\rightarrow}$ and so on, if $\rightarrow$ is clear from context. The following are well-known properties of these transformers.

Proposition 2.3.13. *The following holds in a transition system* $(\Sigma, \rightarrow, I)$.

- *The existential and universal postconditions, and the existential and universal preconditions are De Morgan dual.*

$$post(S) = \neg \widetilde{post}(\neg S) \qquad\qquad \widetilde{pre}(S) = \neg pre(\neg S)$$

- *The state transformers form a Galois connections as follows.*

$$(\mathcal{P}(\Sigma), \subseteq) \underset{post}{\overset{\widetilde{pre}}{\leftrightarrows}} (\mathcal{P}(\Sigma), \subseteq) \qquad\qquad (\mathcal{P}(\Sigma), \subseteq) \underset{pre}{\overset{\widetilde{post}}{\leftrightarrows}} (\mathcal{P}(\Sigma), \subseteq)$$

- *$post$ and pre are completely additive.*
- *$\widetilde{pre}$ and $\widetilde{post}$ are completely multiplicative.*

We can define a strongest postcondition and weakest precondition over traces rather than states, which operate by extending and shortening traces, respectively.

Definition 2.3.28 (Trace Transformers)**.** For a transition relation $\rightarrow \;\subseteq \Sigma \times \Sigma$, we define the following transformers over $\mathcal{P}(\Pi)$ where the variables σ and σ' range over states in Σ and the variable π ranges over the set of (possibly empty) sequences of states in Σ^*.

Strongest Postcondition (Existential Postcondition)

$$tpost_{\rightarrow}(P) \mathrel{\hat{=}} \{\pi\sigma\sigma' \mid \pi\sigma \in P \wedge \sigma \rightarrow \sigma'\}$$

Weakest Precondition (Universal Precondition)

$$\widetilde{tpre}_{\rightarrow}(P) \mathrel{\hat{=}} \{\pi\sigma \mid \forall \sigma' \in \Sigma.\ \pi\sigma\sigma' \in P \vee \sigma \not\rightarrow \sigma'\}$$

Existential Precondition

$$tpre_{\rightarrow}(P) \mathrel{\hat{=}} \{\sigma'\sigma\pi \mid \sigma'\pi \in P \wedge \sigma' \rightarrow \sigma\}$$

Universal Postcondition

$$\widetilde{tpost}_{\rightarrow}(P) \mathrel{\hat{=}} \{\sigma\pi \mid \forall \sigma' \in \Sigma.\ \sigma'\sigma\pi \in P \vee \sigma' \not\rightarrow \sigma\}$$

We now characterize the trace and state semantics as fixed points over the transformers above.

Proposition 2.3.14. *For a transition system $(S, \rightarrow, I)$, trace semantics and state semantics can be characterized as the following fixed points.*

$$\|(S, \rightarrow, I)\|^* = \mathsf{lfp}\, X.\ tpost(X) \cup I \qquad\qquad \|(S, \rightarrow, I)\| \ = \mathsf{lfp}\, X.\ post(X) \cup I$$

Fixed Point Characterizations of Safety

In formal verification, program semantics are computed to check adherence to some specification. We are concerned with specifications that are given in terms of a set of allowed states $\Xi \subseteq S$. A state $\sigma \in \neg\Xi$ is called an *error state*. This is a restricted variant of the safety problem.

Definition 2.3.29 (Safety Checking)**.** A transition system $\mathcal{T} = (\Sigma, \rightarrow, I)$ is *safe* with respect to a specification $\Xi \subseteq \Sigma$, if every well-formed trace contains only states σ in Ξ, and *unsafe* otherwise. The *safety problem* is for $\mathcal{T}$ and Ξ is to decide whether $\mathcal{T}$ is safe w.r.t. Ξ.

We can determine safety using fixed point computations. For this we introduce some notation for a transition system $(\Sigma, \rightarrow, I)$ and a safety specification Ξ below.

$$I_* \mathrel{\hat{=}} \{\pi \in P \mid \exists \sigma \in \pi.\ \sigma \in I\} \qquad\qquad \Xi_* \mathrel{\hat{=}} \{\pi \in P \mid \forall \sigma \in \pi.\ \sigma \in \Xi\}$$

The following proposition summarizes well-known conditions for safety.

Proposition 2.3.15. *The following statements hold exactly if the transition system $\mathcal{T} \mathrel{\hat{=}} (\Sigma, \rightarrow, I)$ is safe w.r.t.* Ξ.

(i) $\mathsf{lfp}\, X.\ I \cup \mathit{tpost}(X) \subseteq \Xi_*$

(ii) $\mathsf{lfp}\, X.\ \neg\Xi \cup \mathit{tpre}(X) \subseteq \neg I_*$

(iii) $(\mathsf{lfp}\, X.\ I \cup \mathit{tpost}(X)) \cap (\mathsf{lfp}\, X.\ \neg\Xi \cup \mathit{tpre}(X)) = \emptyset$

(iv) $\mathsf{gfp}\, X.\ \Xi_* \cap \mathit{t\widetilde{pre}}(X) \supseteq I$

(v) $\mathsf{gfp}\, X.\ \neg I_* \cap \mathit{t\widetilde{post}}(X) \supseteq \neg\Xi_*$

(vi) $(\mathsf{gfp}\, X.\ \Xi_* \cap \mathit{t\widetilde{pre}}(X)) \cup (\mathsf{gfp}\, X.\ \neg I_* \cap \mathit{t\widetilde{post}}(X)) = \Pi$

Proof. For (i), (ii) and (iii), first consider that $\mathsf{lfp}\, X.\ I \cup \mathit{tpost}(X)$ is the set of traces π s.t. π respects $\rightarrow$ and is initialized and that $\mathsf{gfp}\, X.\ I \cup \mathit{tpre}(X)$ is the set of traces π s.t. π respect $\rightarrow$ and ends in an error state $\sigma \not\in \Xi$. We skip the proofs for these characterizations since they are standard. Condition (i) then states that all initialized traces that respect $\rightarrow$ only contain states in Ξ. This is equivalent to stating that every well-formed trace contains only states in Ξ, which is equivalent to safety. Condition (ii) states that every trace that respects the transition relation $\rightarrow$ and ends in an error state does not contain the initial state. This is equivalent to stating that all well-formed traces do not end in an error state. Since the set of well-formed traces is prefix-closed, this is equivalent to stating that no well-formed trace contains an error state, which is equivalent to safety. Condition (iii) states that there is no trace that is initialized, respects $\rightarrow$ and ends in an error state. Again, due to prefix closure of the set of well-formed traces, this is equivalent to stating that there is no trace that is well-formed and contains an error state.

For (iv), (v) and (vi), consider that $\mathsf{gfp}\, X.\ \Xi_* \cap \mathit{t\widetilde{pre}}(X)$ is the set of traces π that are not the prefix of any trace $\pi \cdot \pi'$ that respects $\rightarrow$ and contains an error state, and that $\mathsf{gfp}\, X.\ \neg I_* \cap \mathit{t\widetilde{post}}(X)$ is the set of traces π that are not the suffix of some trace $\pi'\pi$ that respects $\rightarrow$ and contains an initial state. Condition (iv) then states that for all $i \in I$, there is no trace $\pi = i \cdot \pi'$ such that π respects $\rightarrow$ and contains an error state. This is equivalent to stating that there is no well-formed trace that contains an error state, which is equivalent to safety of $\mathcal{T}$. Condition (v) states that for all $e \in \neg\Xi$, there is no trace $\pi = \pi' e$ such that π contains an initial state and respects $\rightarrow$. Since the set of traces that respects $\rightarrow$ is suffix-closed, this is equivalent to stating that there is no trace $\pi = \pi' e$ such that π is initialized and respects $\rightarrow$. This is equivalent to stating that there is no trace that is well-formed and ends in an error state. Since the set of well-formed traces is prefix closed this is equivalent to stating that no well-formed trace contains an error state, therefore, condition (v) is equivalent to $\mathcal{T}$ being safe. Finally, condition (vi) states that for every trace $\pi \in \Pi$, π is not the prefix of any trace that respects $\rightarrow$ and contains an error state or that π is not the suffix of any trace that respects $\rightarrow$ and contains an initial state. Therefore, no trace respects the transition relation $\rightarrow$ and contains an initial state and an error state. Since the set of traces that respect $\rightarrow$ is suffix closed, this is equivalent to stating that no well-formed trace contains an error state, which is equivalent to safety of $\mathcal{T}$. □

Checking the above properties using overapproximate and underapproximate abstract domains, gives us various conditions that are either sufficient or necessary for safety, but not both. The fixed point expressions Proposition 2.3.15 can be soundly approximated using abstract interpretation. The results can be checked against the conditions in Proposition 2.3.15 to determine safety.

2.4 Logic

We now provide preliminary material on logic. We start with our definition of a logical semantics.

Definition 2.4.1 (Logical Semantics)**.** A *logical semantics* is a triple $\mathcal{L} = (\mathcal{S}, \models, \mathcal{F})$ where (i)-(iii) below hold.

(i) $\mathcal{S}$ is a set, called the *set of structures of* $\mathcal{L}$ and

(ii) $\mathcal{F}$ is formal language, called the *set of formulas of* $\mathcal{L}$,

(iii) $\models$, called the *semantic entailment relation of* $\mathcal{L}$, is a binary relation between $\mathcal{S}$ and $\mathcal{F}$.

In this dissertation, we take a semantic approach. For our purposes, we abstract away from the way the set of formulas is constructed (typically, it is inductively generated) and how the semantics are defined (typically inductively, on the structure of the formula). From now on, we will call a logical semantics a *logic*, which deviates from standard practice in the logic literature, but is in line with informal usage of the term in the satisfiability context. Furthermore, for some logics, the set of structures $\mathcal{S}$ may be a proper class rather than a set. We will not consider such cases and therefore the above definition suffices.

We establish some basic terminology. If $\sigma \models \varphi$ holds, then σ *models* φ, and σ is called a *model* of φ. If σ does not model φ, it is a *countermodel.* A formula φ is *satisfiable* if it has a model; it is *valid* if it does not have a countermodel. The semantics of a formula $\|\varphi\|$ of a formula φ is the set of models of the formula.

2.4.1 Propositional Logic and SAT

We now define propositional logic, which will play a central role in this thesis.

Syntax Let P be a finite set of *propositions.* The set of formulas $\mathcal{F}_P$ of propositional logic is defined inductively as the smallest set such that (a)-(d) below hold.

(a) For any $p \in P$, $p \in \mathcal{F}_P$.

(b) For any $\varphi \in \mathcal{F}_P$, the *negation of* φ, $\neg\varphi$ is in $\mathcal{F}_P$.

(c) For any $\varphi_1, \ldots, \varphi_k \in \mathcal{F}_P$ for $k \geq 0$ the disjunction $\varphi_1 \lor \ldots \lor \varphi_k$ is in $\mathcal{F}_P$. The empty disjunction (where $k = 0$) is denoted by the truth constant f.

(d) For any $\varphi_1, \ldots, \varphi_k \in \mathcal{F}_P$ for $k \geq 0$ the conjunction $\varphi_1 \land \ldots \land \varphi_k$ is in $\mathcal{F}_P$. The empty conjunction (where $k = 0$) is denoted by the truth constant t.

We use parenthesis to clarify formula structure where necessary. We also use the following shorthands.

$$\varphi \implies \psi \text{ for } \neg\varphi \lor \psi \qquad\qquad \varphi \iff \psi \text{ for } (\varphi \implies \psi) \land (\psi \implies \varphi)$$

For $p \in P$, a formula of the form p is *positive phase literal* and a formula of the form $\neg p$ is a *negative phase literal.* A *clause* C is a (possibly empty) disjunction over literals $l_1 \lor \ldots \lor l_k$ and a formula φ in *Conjunctive Normal Form* (CNF) is a (possibly empty) conjunction of clauses $C_1 \land \ldots \land C_k$. As is common in the literature, we will sometimes treat clauses as sets of literals and CNF formulas as sets of clauses. We denote the set of propositional clauses and CNF formulas, by $\mathit{Clauses}_P$ and $\mathcal{F}_P^{\text{CNF}}$, respectively.

Semantics The semantics of propositional logic are given in terms of the set of *propositional assignments*, $\mathcal{S}_P \mathrel{\hat{=}} P \to \mathbb{B}$, which map propositions to the *Boolean truth values* true and false, denoted t and f, respectively. The set of Boolean truth values is $\mathbb{B} \mathrel{\hat{=}} \{\mathsf{t}, \mathsf{f}\}$. The relation $\models$ between $\mathcal{S}_P$ and $\mathcal{F}_P$ is then defined inductively on the structure of the formulas as follows, where ρ is a propositional assignment.

(a) For a proposition $p \in P$, it holds that $\rho \models p$ exactly if $\rho(p) = \mathsf{t}$.

(b) For a negation $\neg\varphi \in \mathcal{F}_P$, it holds that $\rho \models \neg\varphi$ exactly if $\rho \not\models \varphi$.

(c) For a disjunction $\varphi_1 \vee \ldots \vee \varphi_k \in \mathcal{F}_P$, it holds that $\rho \models \varphi_1 \vee \ldots \vee \varphi_k$ exactly if there is an i with $1 \leq i \leq k$ such that $\rho \models \varphi_i$. As a consequence, it holds universally that $\rho \not\models \mathsf{f}$.

(d) For a conjunction $\varphi_1 \wedge \ldots \wedge \varphi_k \in \mathcal{F}_P$, it holds that $\rho \models \varphi_1 \wedge \ldots \wedge \varphi_k$ exactly if for all i with $1 \leq i \leq k$ it holds that $\rho \models \varphi_i$. As a consequence, it holds universally that $\rho \models \mathsf{t}$.

Since we will be mainly concerned with studying sets of clauses, we give the following restricted definition of propositional logic. *Propositional logic* is the logic $\mathcal{L}_P \mathrel{\hat{=}} (\mathit{Clauses}_P, \models_P, \mathcal{S}_P)$. The propositional satisfiability problem (SAT) is the problem of deciding whether there is a propositional assignment that models each clause in a given, finite set $\varphi \subseteq \mathit{Clauses}_P$.

The satisfiability of any single propositional formula of arbitrary structure can be decided by transforming it into an equisatisfiable SAT instance of the form above [168, 146].

2.4.2 Quantifier-Free First Order Logic and SMT

The area of Satisfiability Modulo Theories (SMT) [9] is concerned with satisfiability of some fragment of first-order logic with respect to some background theory. Formulas considered in SMT are *ground*, that is, they do not contain first-order variables and they are *quantifier-free*, meaning that they do not contain the first-order quantifiers $\forall$ and $\exists$.

Syntax Formulas in first-order logic are defined over a *signature* Σ, which is a set of *function symbols* and *predicate symbols*. Each symbol f in Σ is associated with an *arity* $ar(f)$ in $\mathbb{N}_0$. Predicate and function symbols with arity zero are called *propositions* and *constants*, respectively.

We first define the set $\mathcal{T}_\Sigma$ of *ground first-order terms* over the signature Σ inductively as the smallest set that satisfies the conditions (a) and (b) below.

(a) Let p be a constant in Σ, then p is also a term in $\mathcal{T}_\Sigma$.

(b) Let f be a function symbol in Σ, and let $t_1, \ldots, t_{ar(f)}$ be terms in $\mathcal{T}_\Sigma$, then $f(t_1, \ldots, t_{ar(f)})$ is a term in $\mathcal{T}_\Sigma$.

We now define the set $\mathcal{F}_\Sigma$ of *ground, quantifier-free first-order formulas* over the signature Σ (or simply, Σ-formulas), inductively as the smallest set satisfying (a) to (d) below.

(a) Let p be a predicate symbol and let $t_1, \ldots, t_{ar(p)} \in \mathcal{T}_\Sigma$ be terms, then $p(t_1, \ldots, t_{ar(p)})$ is in $\mathcal{F}_\Sigma$.

(b) For any $\varphi \in \mathcal{F}_\Sigma$, the *negation of* φ, $\neg\varphi$ is in $\mathcal{F}_\Sigma$.

(c) For any $\varphi_1, \ldots, \varphi_k \in \mathcal{F}_\Sigma$ for $k \geq 0$ the disjunction $\varphi_1 \vee \ldots \vee \varphi_k$ is in $\mathcal{F}_\Sigma$. The empty disjunction (where $k = 0$) is denoted by the truth constant f.

(d) For any $\varphi_1, \ldots, \varphi_k \in \mathcal{F}_\Sigma$ for $k \geq 0$ the conjunction $\varphi_1 \wedge \ldots \wedge \varphi_k$ is in $\mathcal{F}_\Sigma$. The empty conjunction (where $k = 0$) is denoted by the truth constant t.

Semantics First-order formulas are evaluated with respect to a first-order structure. A Σ-*structure* for a signature Σ is a pair (U, ξ) consisting of a non-empty set U called the *universe* and an *interpretation function* ξ with the signature as below.

$$\xi : \Sigma \to (\bigcup_{i \in \mathbb{N}_0} \mathcal{P}(U^i)) \cup (\bigcup_{i \in \mathbb{N}_0} U^i \to U)$$

Interpretation functions provide the meaning of the symbols defined in the signature by assigning each predicate symbol p in Σ to a relation in $\mathcal{P}(U^{ar(p)})$ and each function symbol f to a function $U^{ar(p)} \to U$. We extend interpretation functions to terms as follows.

$$\xi(f(t_1, \ldots, t_k)) \mathrel{\hat{=}} \xi(f)(\xi(t_1), \ldots, \xi(t_k))$$

Given a structure $\sigma = (U, \xi)$, we will for convenience denote $\xi(p)$ as $\sigma(p)$ for any symbol $p \in \Sigma$.

The entailment relation $\models_\Sigma$ of a ground, quantifier-free formula $\varphi \in \mathcal{F}_\Sigma$ can be defined as follows, where σ is some first-order Σ-structure.

(a) For a predicate symbol p of arity k and terms $t_1, \ldots, t_k$, it holds that $\sigma \models_\Sigma p(t_1, \ldots, t_k)$ exactly if $(\sigma(t_1), \ldots, \sigma(t_k)) \in \sigma(p)$.

(b) For a negation $\neg\varphi \in \mathcal{F}_\Sigma$, it holds that $\sigma \models_\Sigma \neg\varphi$ exactly if $\sigma \not\models_\Sigma \varphi$.

(c) For a disjunction $\varphi_1 \vee \ldots \vee \varphi_k \in \mathcal{F}_\Sigma$, it holds that $\sigma \models_\Sigma \varphi_1 \vee \ldots \vee \varphi_k$ exactly if there is an i s.t. $1 \leq i \leq k$ and $\sigma \models_\Sigma \varphi_i$. As a consequence, it holds universally that $\sigma \not\models_\Sigma \mathsf{f}$.

(d) For a conjunction $\varphi_1 \wedge \ldots \wedge \varphi_k \in \mathcal{F}_\Sigma$, it holds that $\sigma \models_\Sigma \varphi_1 \wedge \ldots \wedge \varphi_k$ exactly if for all i with $1 \leq i \leq k$ it holds that $\sigma \models_\Sigma \varphi_i$. As a consequence, it holds universally that $\sigma \models_\Sigma \mathsf{t}$.

We denote the set of terms occurring in a formula φ by $\mathsf{H}(\varphi)$ (this set is also called the *Herbrand universe*). A ground, quantifier free-formula φ is *atomic* if φ is a predicate $p(t_1, \ldots, t_k)$ over a possibly empty set of terms $t_1, \ldots, t_k$, and a *first-order literal* if it is atomic or a negation $\neg\varphi$ of an atomic formula φ. In the former case, it is in *positive phase*, in the latter, in *negative phase*. We denote the set of atomic subformulas of a formula φ as $\mathsf{A}(\varphi)$, and the set of literals occurring in φ as $\mathsf{L}(\varphi)$. The formula φ is a *first-order clause* if it is a disjunction $l_1, \ldots, l_k$ of first-order literals. We denote by $\mathit{Clauses}_\Sigma$ the set of ground, quantifier-free first-order clauses over a signature Σ.

In SMT, formulas are analyzed with respect to a background theory, which fixes the interpretation of certain functions and predicates. Following convention [9], we define an SMT *theory* as a set of Σ-structures $\mathcal{S}_\Sigma$. Also following convention, we refer to a constant v as a *variable* if v may take more than one value in the interpretations in $\mathcal{S}_\Sigma$. For example, consider a theory in which integer constants are given their standard interpretation. The symbol 2 occurring in a formula is then a constant, but since it is interpreted as the number 2 in every interpretation, it is not a variable. An uninterpreted constant symbol v that is assigned to different numbers in $\mathbb{Z}$ by different interpretations, on the other hand, is a variable. We denote the set of variables occurring in a formula φ as $\mathsf{V}(\varphi)$. We use the term SMT *logic* to denote ground, quantifier-free fragments of first-order logic, evaluated with respect to an SMT theory $\mathcal{S}_\Sigma$, i.e., a logic of the form

$$\mathcal{L} = (\mathcal{S}_\Sigma, \models_\Sigma, \mathcal{F}_\Sigma),$$

where $\mathcal{S}_\Sigma$ is an SMT theory, $\mathcal{F}_\Sigma$ is a set of ground, quantifier-free Σ-formulas and $\models_\Sigma$ is defined as above. The Satisfiability Modulo Theory problem (SMT) is to decide, given an SMT logic $(\mathcal{S}_\Sigma, \models_\Sigma, \mathcal{F}_\Sigma)$ and a formula $\varphi \in \mathcal{F}_\Sigma$, whether φ has a model in $\mathcal{S}_\Sigma$.

Chapter 3

An Algebraic Satisfiability Framework

This dissertation generalizes logical satisfiability and presents a new understanding of SAT algorithms in the framework of abstract interpretation. The consequences of this understanding include the ability to generalize existing SAT algorithms and to relate procedures in program analysis and satisfiability research. We derive these results within a novel formal framework called *abstract satisfaction.* This chapter introduces the framework; the contributions are as follows.

- A fixed point characterization of logical satisfiability.
- A methodology for applying abstract interpretation to approximate the resulting fixed points.
- A generalization of the satisfiability problem to a decision problem involving closure operators, which we call the *bottom-everywhere problem.*
- A lifting of our abstract-interpretation based satisfiability framework to the bottom-everywhere problem to obtain the *abstract satisfaction framework.*

The main application of abstract satisfaction is to allow a joint treatment of program analysis and satisfiability, and to enable technology transfer between the two fields. This is reflected in the coming chapters. In Chapter 5, we generalize an existing satisfiability procedure using the framework of abstract satisfaction. Chapters 6 and 7 show how the resulting abstract algorithm can be instantiated to yield both novel satisfiability solvers and program analyzers.

Motivation In the satisfiability literature, two styles of presentation are common for algorithms. One is the proof-theoretic perspective, wherein the algorithm is viewed as the systematic application of a set proof rules for some logic. The second is a detailed discussion of low-level data structures and procedures to facilitate effective implementations.

The algebraic perspective brought to bear by abstract interpretation provides a map of the landscape of sound satisfiability procedures, which complements the proof-theoretic and low-level algorithmic perspectives that are common. For a single algorithm, an algebraic analysis allows for a new formal understanding, including new proofs of existing results, but most importantly, generalizations of the algorithm to novel application domains. Studying multiple algorithms, the analysis provides the formally rigorous conceptual vocabulary necessary to study similarities and differences between procedures. Finally, an algebraic understanding of satisfiability procedures can guide the search for novel procedures, and provide a formal blueprint for modular and extensible software architectures.

The abstract satisfaction framework presented in this section forms the basis of the remainder of all work in this thesis. In Chapter 4, we use abstract satisfaction to give algebraic characterizations of existing satisfiability procedures. In Chapter 5, we use it to generalize satisfiability procedures to new problem domains. Concrete instantiations of the resulting generalized algorithms are presented in Chapters 6 and 7.

Outline Section 3.1 introduces the abstract satisfaction framework, a framework for applying abstract interpretation to the satisfiability problem. Section 3.2 introduces the bottom-everywhere problem, which generalizes abstract satisfaction to a framework for checking properties of closure operators.

3.1 Applying Abstract Interpretation to Logic

We now present a framework for applying abstract interpretation to logic. For a logic $\mathcal{L} = (\mathcal{S}, \models, \mathcal{F})$, we consider the concrete lattice as the semantic universe, that is, the powerset domain of structures $\mathcal{P}(\mathcal{S})$. Section 3.1.1 introduces transformers on this lattice that are semantic counterparts to various inference tasks. Section 3.1.2 discusses how approximations of these transformers can be applied to determine satisfiability of formulas.

3.1.1 Structure Transformer

The use of logic for problem solving can be viewed as a two-step process. The first step is to construct a logical formula to reflect some property of an object of interest. The second step is to apply logical inference to obtain additional information about the problem. We use the term *inference* to denote mappings between logical assertions. Assume a background theory. We distinguish four modes of inference.

- *Deduction*, which is concerned with finding necessary consequences of a statement.
- *Abduction*, which derives sufficient reasons for a statement.
- *Counterdeduction*, which models contrapositive reasoning and finds necessary consequences of a statement under the assumption that the theory is false.
- *Counterabduction*, which finds sufficient reason for a statement under the assumption that the theory is false.

The following *structure transformers* represent concrete, semantic versions of these logical inference tasks. The transformers are defined with respect to some formula φ, which takes the role of the background theory.

Definition 3.1.1 (Structure Transformers)**.** Let $\mathcal{L} = (\mathcal{S}, \models, \mathcal{F})$ be a logic, and let $\varphi \in \mathcal{F}$ be a formula. The structure transformers of φ are defined below.

Deduction transformer

$$ded_{\varphi}(S) \mathrel{\hat{=}} \{\sigma \in \mathcal{S} \mid \sigma \in S \wedge \sigma \models \varphi\}$$

Abduction transformer

$$abd_{\varphi}(S) \mathrel{\hat{=}} \{\sigma \in \mathcal{S} \mid \sigma \in S \vee \sigma \not\models \varphi\}$$

Counterdeduction transformer

$$cded_{\varphi}(S) \mathrel{\hat{=}} \{\sigma \in \mathcal{S} \mid \sigma \in S \wedge \sigma \not\models \varphi\}$$

Counterabduction transformer

$$cabd_{\varphi}(S) \mathrel{\hat{=}} \{\sigma \in \mathcal{S} \mid \sigma \in S \vee \sigma \models \varphi\}$$

The deduction transformer removes structures from the input set which do not satisfy the formula φ. To illustrate that the deduction transformer is a semantic counterpart of deduction, consider two formulas φ_S and φ_R such that the sets S and R are, respectively, the models of the two formulas. Applying the deduction transformer $ded_{\varphi_R}(S)$ computes the strongest semantically expressed consequence of S, assuming the truth of φ_R. The result is $ded_{\varphi_R}(S) = \|\varphi_R\| \cap S = R \cap S$. This is the semantic equivalent of deriving the statement $\varphi_S \wedge \varphi_R$.

The abduction transformer adds structures to the input set that do not satisfy a given formula. Abduction denotes, informally speaking, the act of extracting from a statement ψ a reason φ whose truth suffices to ensure the truth of ψ with respect to some background theory. Computing the transformer $abd_\varphi(S)$ can be viewed as finding the weakest semantically expressed reason for S, assuming the truth of the statement φ. Defining S and R as above, the result of applying abd_{φ_R} to a set of models S is the set $\neg R \cup S$, which is the set of models of the formula $\varphi_R \implies \varphi_S$.

Abduction and deduction provide dual perspectives on logical inference. If extracting ψ starting from φ is sound deductive inference, then extracting φ from ψ is sound abductive inference. This relationship manifests mathematically as a Galois connection between abduction and deduction transformers.

There are two justifications for using the definition of abduction provided in this chapter. As shown in Chapters 4 and 5, our notion of abduction formalizes operations in SAT solvers that are commonly described as “finding conflict reasons”. These operations are used to find logical statements that are sufficient for deriving a contradiction. The notion of abduction in logic [120] does not cover such operations since it requires that the result of abduction must be consistent with the background theory and conflict reasons do not satisfy this requirement.

In contrast to logic-based abduction, our notion of abduction is a symmetric counterpart to deduction, without any additional restrictions. Just as in deduction, anything follows from a contradiction, in our conception of abduction, a contradiction is a sufficient explanation for anything. The following statement illustrates the fundamental connection between abduction and deduction.

Proposition 3.1.1. *The deduction and abduction transformers form the Galois connection below.*

$$\begin{tikzcd}(\mathcal{P}(\mathcal{S}), \subseteq) \arrow[r, shift right, "ded_\varphi"'] & (\mathcal{P}(\mathcal{S}), \subseteq) \arrow[l, shift right, "abd_\varphi"']\end{tikzcd}$$

Proof. Monotonicity of abd_φ and ded_φ is easy to see. We have that $abd_\varphi \circ ded_\varphi(S) = S \cup \{\sigma \in \mathcal{S} \mid \sigma \not\models \varphi\}$, therefore $abd_\varphi \circ ded_\varphi$ is inflationary. Similarly, $ded_\varphi \circ abd_\varphi(S) = S \cap \{\sigma \in \mathcal{S} \mid \sigma \models \varphi\}$, therefore $ded_\varphi \circ abd_\varphi$ is deflationary. The result follows from Proposition 2.3.4. □

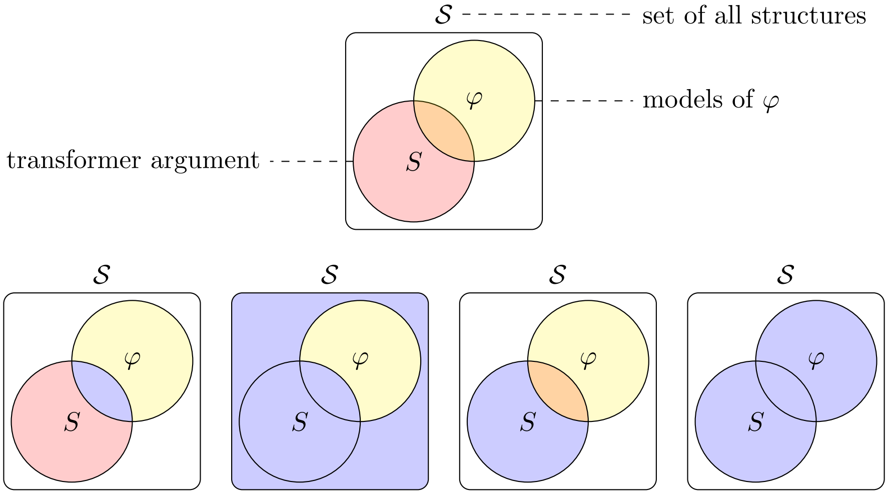
For some logics, approximations of the structure transformers are available in existing abstract domains. The reason is that approximate state transformers over guard statements correspond to approximations of the structure transformers.

Proposition 3.1.2. *Let $\mathcal{S}$ be the set of memory states associated with some* CFG *G, and let $T_{[\varphi]} \subseteq \mathcal{S} \times \mathcal{S}$ be the transition relation of some guard statement where $(\sigma, \sigma') \in T_{[\varphi]}$ exactly if $\sigma = \sigma'$ and $\sigma \models \varphi$. Then the following equivalences hold.*

$$\text{(i) } post_{T_{[\varphi]}} = ded_\varphi \qquad\qquad \text{(ii) } \widetilde{pre}_{T_{[\varphi]}} = abd_\varphi$$

Proof. We have that $post_{T_{[\varphi]}}(S) = \{\sigma \in \mathcal{S} \mid \exists \sigma' \in S.\ \sigma' = \sigma \wedge \sigma' \models \varphi\}$. This set is equal to $\{\sigma \in \mathcal{S} \mid \sigma \in S \wedge \sigma \models \varphi\} = ded_\varphi(S)$. Conversely, $\widetilde{pre}_{T_{[\varphi]}}(S) = \{\sigma \in \mathcal{S} \mid \forall \sigma' \in \mathcal{S}. \sigma' \in S \vee \neg(\sigma' = \sigma \wedge \sigma' \models \varphi)\}$, which is equal to $\{\sigma \in \mathcal{S} \mid \forall \sigma' \in \mathcal{S}. \sigma' \in S \vee \neg(\sigma' = \sigma \wedge \sigma' \models \varphi)\}$. □

The following proposition shows that the relationship between abduction and deduction is an instance of a more general pattern.



(i) $ded_\varphi(S)$ (ii) $abd_\varphi(S)$ (iii) $cded_\varphi(S)$ (iv) $cabd_\varphi(S)$

Figure 3.1: Schematic depictions of the structure transformers. The set labeled φ is the set of models of φ. The result of transformer application is in blue.

Proposition 3.1.3. *The functions ded_φ and abd_φ are De Morgan duals.*

Proof.

$$\begin{aligned}\neg \circ abd_\varphi(\neg S) &= \neg\{\sigma \mid \sigma \in \neg S \vee \sigma \not\models \varphi\}\\ &= \{\sigma \mid \sigma \notin \neg S \wedge \sigma \models \varphi\}\\ &= \{\sigma \mid \sigma \in S \wedge \sigma \models \varphi\}\\ &= ded_\varphi(S)\end{aligned}$$

□

Counterdeduction and counterabduction correspond, respectively, to deduction and abduction w.r.t. to the complement of the semantics of the formula. If the underlying logic admits a logical negation operator $\neg$ that complements semantics, then we have that $cded_\varphi = ded_{\neg\varphi}$ and $cabd_\varphi = abd_{\neg\varphi}$. We note that while many logics have such a negation operator, many logical algorithms internally use restricted problem representations and logical data structures that lack negation. Similarly, precise negation operators are rare in abstract domains.

We illustrate the transformers in Figure 3.1 and with the following example from propositional logic.

Example 3.1.1. *Consider the following propositional formula.*

$$\varphi \mathrel{\hat{=}} (p \vee q)$$

In the following, we denote a structure $\{p \mapsto \mathsf{t}, q \mapsto \mathsf{f}\}$ simply by tf. Recall that $\|\varphi\|$ denotes the set of models of φ.

$ded_\varphi(\top)$ returns the models of the following formula.

$$ded_\varphi(\top) = ded_p(\top) \cup ded_q(\top) = \{\mathsf{tt}, \mathsf{tf}, \mathsf{ft}\}$$

$abd_\varphi(\bot)$ finds conditions that make the formula false.

$$abd_\varphi(\bot) = abd_p(\bot) \cap abd_q(\bot) = \{\mathsf{ff}\}$$

The following examples show the relation of these transformers to logical inference.

$$\begin{aligned} ded_\varphi(\|\neg p\|) &= ded_\varphi(\{\mathsf{ff}, \mathsf{ft}\}) = \{\mathsf{ft}\} = \|\neg p \wedge q\| \\ abd_\varphi(\|p\|) &= abd_\varphi(\{\mathsf{tf}, \mathsf{tt}\}) == \{\mathsf{tf}, \mathsf{tt}, \mathsf{ff}\} = \|p \vee \neg q\| \end{aligned}$$

Transformers for logics can typically be defined inductively on the structure of the formula. We demonstrate this via the case of propositional logic.

Proposition 3.1.4 (Structure Transformers for Propositional Logic)**.** *Consider a propositional formula φ over a finite set of propositions P and the set of propositional structures $\mathcal{S}_P = P \to \mathbb{B}$. The structure transformers ded_φ, abd_φ, $cded_\varphi$, $cabd_\varphi$ have the inductive characterization given below.*

If $\varphi = p$ for $p \in P$, then:

$$\begin{aligned} ded_\varphi(S) &= \{\sigma \mid \sigma \in S \wedge \sigma(p) = \mathsf{t}\} & abd_\varphi(S) &= \{\sigma \mid \sigma \in S \vee \sigma(p) = \mathsf{f}\} \\ cded_\varphi(S) &= \{\sigma \mid \sigma \in S \wedge \sigma(p) = \mathsf{f}\} & cabd_\varphi(S) &= \{\sigma \mid \sigma \in S \vee \sigma(p) = \mathsf{t}\} \end{aligned}$$

if $\varphi = \neg\psi$, then:

$$\begin{aligned} ded_\varphi(S) &= cded_\psi(S) & abd_\varphi(S) &= cabd_\psi(S) \\ cded_\varphi(S) &= ded_\psi(S) & cabd_\varphi(S) &= abd_\psi(S) \end{aligned}$$

if $\varphi = \bigvee \Psi$, then:

$$\begin{aligned} ded_\varphi(S) &= \bigcup_{\psi \in \Psi} ded_\psi(S) & abd_\varphi(S) &= \bigcap_{\psi \in \Psi} abd_\psi(S) \\ cded_\varphi(S) &= \bigcup_{\psi \in \Psi} cded_\psi(S) & cabd_\varphi(S) &= \bigcap_{\psi \in \Psi} cabd_\psi(S) \end{aligned}$$

if $\varphi = \bigwedge \Psi$, then:

$$\begin{aligned} ded_\varphi(S) &= \bigcap_{\psi \in \Psi} ded_\psi(S) & abd_\varphi(S) &= \bigcup_{\psi \in \Psi} abd_\psi(S) \\ cded_\varphi(S) &= \bigcap_{\psi \in \Psi} cded_\psi(S) & cabd_\varphi(S) &= \bigcup_{\psi \in \Psi} cabd_\psi(S) \end{aligned}$$

Proof. Remember that for a proposition p, we have that $\sigma \models p$ exactly if $\sigma(p) = \mathsf{t}$. It is therefore easy to see that the transformers ded_p, abd_p, $cded_p$ and $cabd_p$ may be characterized as above.

We show the remaining cases only for the deduction transformer.

$$\begin{aligned} \text{For } \varphi = \neg\psi, \qquad & ded_\varphi(S) = \{\sigma \mid \sigma \in S \wedge \sigma \models \neg\psi\} \\ & = \{\sigma \mid \sigma \in S \wedge \sigma \not\models \psi\} = cded_\psi(S) \\ \text{For } \varphi = \bigvee \psi, \qquad & ded_\varphi(S) = \{\sigma \mid \sigma \in S \wedge \sigma \models \bigvee \psi\} \\ & = \{\sigma \mid \exists \psi \in \Psi .\ \sigma \in S \wedge \sigma \models \psi\} \\ & = \bigcup_{\psi \in \Psi} \{\sigma \mid \sigma \in S \wedge \sigma \models \psi\} = \bigcup_{\psi \in \Psi} ded_\psi(S) \\ \text{For } \varphi = \bigwedge \psi, \qquad & ded_\varphi(S) = \{\sigma \mid \sigma \in S \wedge \sigma \models \bigwedge \psi\} \end{aligned}$$

$$
\begin{aligned}
&= \{\sigma \mid \forall \psi \in \Psi.\ \sigma \in S \wedge \sigma \models \psi\} \\
&= \bigcap_{\psi \in \Psi} \{\sigma \mid \sigma \in S \wedge \sigma \models \psi\} = \bigcap_{\psi \in \Psi} ded_\psi(S)
\end{aligned}
$$

□

The structure transformers are closure operators that add or remove certain structures from a given set.

Proposition 3.1.5. *The transformers ded_φ and $cded_\varphi$ are lower closure operators. The transformers abd_φ and $cabd_\varphi$ are upper closure operators.*

Proof. We show the case for ded_φ, the other cases are similar. It is easy to see that the transformer is deflationary and monotone. We now show that it is idempotent. We have that $ded_\varphi(S) = S \cap \|\varphi\|$. The expression $ded_\varphi \circ ded_\varphi(S)$ is then equivalent to $S \cap \|\varphi\| \cap \|\varphi\|$ which is equivalent to $S \cap \|\varphi\|$ and $ded_\varphi(S)$. □

3.1.2 Satisfiability via Abstract Fixed Points

As a first step towards applying abstract interpretation logic, we characterize satisfiability and validity in terms of fixed points. This may seem odd, since the greatest fixed point of an idempotent function such as ded_φ is the element $ded_\varphi(\top)$. Since the abstraction of a lower closure is not always a lower closure, computing a fixed point with the abstract transformer yields strictly more precision than applying it once. The next chapter will provide examples. Approximation of closure operators via abstract fixed point computations is well-known [38].

Theorem 3.1.6 (Fixed Point Characterization of Satisfiability)**.** *The following two conditions are equivalent to unsatisfiability of φ.*

(i) $\mathsf{gfp}\, ded_\varphi = \bot$ *(ii)* $\mathsf{lfp}\, abd_\varphi = \top$

The following two conditions are equivalent to validity of φ.

(iii) $\mathsf{gfp}\, cded_\varphi = \bot$ *(iv)* $\mathsf{lfp}\, cabd_\varphi = \top$

Proof. Since ded_φ is a closure operator, we have that $\mathsf{gfp}\, ded_\varphi = ded_\varphi(\top)$, which is the set of models of φ. The formula φ is satisfiable exactly if this set is non-empty. The second statement follows from De Morgan duality: $\mathsf{lfp}\, abd_\varphi = abd_\varphi(\bot) = \neg ded_\varphi(\top)$, therefore φ is satisfiable exactly if $\mathsf{lfp}\, abd_\varphi(\bot)$.

For statement (iii), it's easy to see by a similar argument that $\mathsf{gfp}\, cded_\varphi = \bot$ holds exactly if there is no countermodel, and for (iv) that $\mathsf{lfp}\, cabd_\varphi = \top$ holds exactly if every formula is a model. □

We are now ready to state the main soundness theorem, which shows how abstract interpretation can be applied to determine satisfiability and validity.

Theorem 3.1.7 (Abstract Satisfaction)**.** *Let $oded_\varphi$, $oabd_\varphi$, $ocded_\varphi$ $ocabd_\varphi : O \to O$ be sound overapproximations of the structure transformers w.r.t. (α_O, γ_O) and let $uded_\varphi$, $uabd_\varphi$, $uded_\varphi$ and $ucabd_\varphi : U \to U$ be sound underapproximations of the structure transformers w.r.t. (α_U, γ_U).*

Then φ is unsatisfiable if one of the following conditions hold:

(i) $\gamma_O(\mathsf{gfp}\, oded_\varphi) = \bot$ *(ii)* $\gamma_U(\mathsf{lfp}\, uabd_\varphi) = \top$

Then φ is valid if one of the following conditions hold:

(iii) $\gamma_O(\mathsf{gfp}\, ocded_\varphi) = \bot$ *(iv)* $\gamma_U(\mathsf{lfp}\, ucabd_\varphi) = \top$

Proof. The results follow from fixed point transfer and Theorem 3.1.6. □

The above theorem gives conditions for determining unsatisfiability and validity. It is also possible to determine whether a formula is satisfiable or invalid in the abstract using completeness properties of transformers. We discuss this in a generalized setting in Section 3.2.

3.2 The Bottom-Everywhere Problem

This section generalizes the satisfiability problem to an algebraic problem called the *bottom-everywhere problem*. The bottom-everywhere problem is to check whether a deflationary, completely additive function on a Boolean algebra maps every element to the bottom of the lattice. This generalization has a number of appealing properties: It preserves the duality between satisfiability and validity, which will be algorithmically relevant for fixed point combination procedures discussed in Chapters 4 and 5. It also allows us to view logical satisfiability and program correctness as the same problem. We explore the consequences of this in Chapters 6 and 7 which discuss two practical instantiations of the same algebraic algorithm, one as an SMT solver and one as a program verifier.

The motivation for this generalization is to provide a framework that is broad enough to enable description of satisfiability algorithms and program analyzers in a common framework.

Besides defining the bottom-everywhere problem, this section establishes fixed point characterizations of the bottom everywhere problem and describes techniques for approximating these fixed points in abstractions. We show that satisfiability the satisfiability problem for logical formulas and the error reachability problem for programs are instances of the bottom-everywhere problem. We also provide generalized notions of abstract and concrete witnesses and counterwitnesses, which generalize notions such as models, counterexamples and invariants.

3.2.1 Bottom-Everywhere

We first formally define the bottom-everywhere problem. The following sections will provide intuition by showing how satisfiability and reachability can be viewed as instances of this problem.

Definition 3.2.1 (Bottom-Everywhere)**.** Let B be a bounded lattice. A function $f : B \to B$ is *bottom everywhere* if for all $b \in B$, $f(b) = \bot$. A function $g : B \to B$ is *top everywhere* if for all $b \in B$, $g(b) = \top$.

Definition 3.2.2 (The Bottom-Everywhere Problem)**.** Let B be a complete Boolean algebra. The *bottom-everywhere problem* for a completely additive, deflationary function $f : B \to B$ is to decide whether f is bottom everywhere. Dually, the *top-everywhere problem* for a completely multiplicative, inflationary function $g : B \to B$ is to decide whether g is top-everywhere.

We provide an alternative characterization for functions such as f and g above.

Proposition 3.2.1. *A function f on a complete Boolean algebra $(B, \subseteq, \cap, \cup)$, is completely additive and deflationary exactly if there is some $a \in A$ such that for any $b \in B$:*

$$f(b) = b \cap a$$

Proof. Assume f is completely additive and deflationary. It follows that $b \cap f(\top) = b \cap f(b \cup \neg b)$ and due to additivity of f, this is equal to $b \cap (f(b) \cup f(\neg b))$. By distributivity, we get $(b \cap f(b)) \cup (b \cap f(\neg b))$. Since $f(\neg b) \subseteq \neg b$ we have that $b \cap f(\neg b) = \bot$ and $b \cap f(b) = f(b)$, therefore $b \cap f(b \cup \neg b) = f(b)$. It follows that the element $f(\top)$ is an a such as required above.

Now consider a function $f = \lambda b.\ b \cap a$ for some $a \in B$. Clearly, f is deflationary. Now consider a set $Q \subseteq B$, then we have that $f(\bigcup Q) = (\bigcup Q) \cap a$. Complete Boolean algebras satisfy a infinitary distributive law which equates the above statement to $\bigcup_{q \in Q}(q \cap a)$, which is in turn equal to $\bigcup_{q \in Q} f(q)$. Therefore, f is completely additive. □

We can now see that the functions we study are closure operators.

Proposition 3.2.2. *Every completely additive, deflationary function $f : B \to B$ on a Boolean algebra $(B, \subseteq, \cap, \cup)$ is a lower closure operator.*

Proof. We first show that f is order-preserving. Let $a, b \in B$ such that $a \subseteq b$. By Proposition 3.2.1, it then holds that $f(a) = a \cap f(\top)$ and $f(b) = b \cap f(\top)$, therefore, $f(a) \subseteq f(b)$.

We now show that f is idempotent. We have that $f(f(a)) = f(a \cap f(\top)) = a \cap f(\top) \cap f(\top) = a \cap f(\top) = f(a)$. □

Dually, every completely multiplicative, inflationary function over B is an upper closure. The De Morgan dual of a completely additive lower closure is a completely multiplicative upper closure.

Proposition 3.2.3. *Let $f : B \to B$ be a completely additive, deflationary function over a complete Boolean algebra B. Then $\tilde{f} \mathrel{\hat{=}} \neg \circ f \circ \neg$ is a completely multiplicative, inflationary function.*

Proof. If f is completely additive and deflationary, then by Proposition 3.2.1, we have that $f(b) = b \cap f(\top)$ for any $b \in B$. Then, for any $b \in B$, it holds that $\tilde{f}(b) = \neg f(\neg b) = \neg(\neg b \cap f(\top)) = b \cup \neg f(\top)$. Therefore, by the dual of Proposition 3.2.1, we have that $\tilde{f}$ is inflationary and completely additive. □

In the following, fix a function $f : B \to B$ as above and its De Morgan dual $\tilde{f}$. The functions f and $\tilde{f}$ form a Galois connection.

Proposition 3.2.4. *The transformers $(f, \tilde{f})$ form the Galois connection below.*

$$\begin{tikzcd} (B, \subseteq) \arrow[r, shift right, "f"'] & (B, \subseteq) \arrow[l, shift right, "\tilde{f}"'] \end{tikzcd}$$

Proof. Consider $\tilde{f} \circ f(b)$ for some $b \in B$. We can rewrite the above to $(b \cap f(\top)) \cup \neg f(\top)$, by Proposition 3.2.1 and since $\tilde{f} = \lambda a.a \cup \neg f(\top)$ (see proof of Proposition 3.2.3). We apply distributivity to obtain $(b \cup \neg f(\top)) \cap (f(\top) \cup \neg f(\top)) = (b \cup \neg f(\top) = \tilde{f}(b)$. We have shown that $\tilde{f} \circ f(b) = \tilde{f}(b)$ which is inflationary. Similarly, we can show that $f \circ \tilde{f}$ is deflationary. Since f and $\tilde{f}$) are closures (by Proposition 3.2.2 and its dual) and hence order-preserving, it follows that the pair forms a Galois connection. □

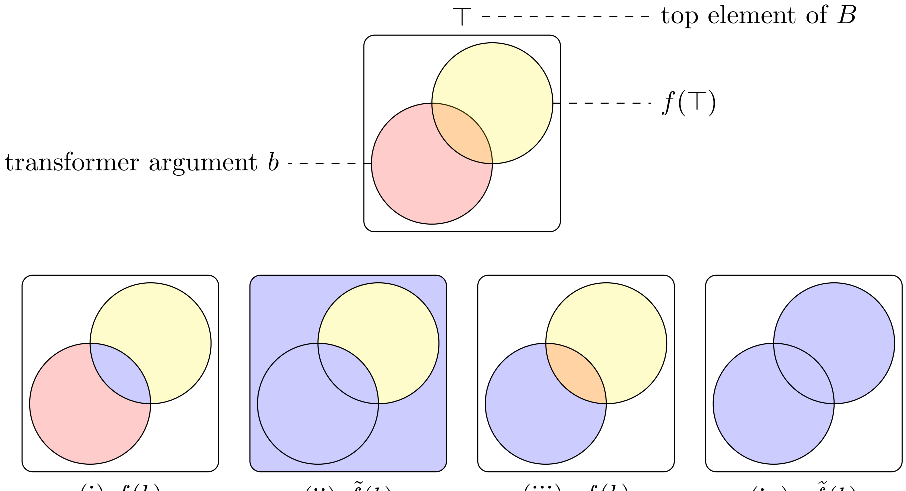
Since B is a complete Boolean algebra, f is also completely multiplicative.

Proposition 3.2.5. *The transformer f is completely multiplicative, the transformer $\tilde{f}$ is completely additive.*

Proof. Consider $f(\bigcap Q)$, then by Proposition 3.2.1, $f(\bigcap Q) = (\bigcap Q) \cap f(\top)$ which is equal to $\bigcap_{q \in Q}(q \cap f(\top)) = \bigcap_{q \in Q} f(q)$. The proof for $\tilde{f}$ is similar. □

Checking if a function is bottom everywhere is equivalent to checking if the De Morgan dual of that function is top everywhere. The following statement generalizes the duality between satisfiability and validity.

Proposition 3.2.6. *An additive, deflationary function $f : B \to B$ on a complete Boolean algebra is bottom everywhere exactly if its De Morgan dual $\tilde{f}$ is top-everywhere.*



(i) $f(b)$ (ii) $\tilde{f}(b)$ (iii) $cf(b)$ (iv) $c\tilde{f}(b)$

Figure 3.2: Schematic depiction of f, $\tilde{f}$, cf and $c\tilde{f}$ as operators on sets. The result of transformer application is in blue.

Recall that in the case of logical inference, we define countertransformers for deduction and abduction. Analogously, we defined two transformers in addition to f and $\tilde{f}$, which we call countertransformers. If the Boolean algebra underlying f is a powerset algebra, then f may be viewed as removing a number of elements from the input set, whereas $\tilde{f}$ would add them. The countertransformers remove, respectively, add exactly those elements which aren't removed, respectively, added by f and $\tilde{f}$.

Definition 3.2.3 (Countertransformers)**.** Let $f : B \to B$ be a deflationary, completely additive function on a complete Boolean algebra, and $\tilde{f}$ be its De Morgan dual. The *countertransformers* of f and $\tilde{f}$ are $cf : B \to B$, respectively, $c\tilde{f} : B \to B$, defined below.

$$cf(b) \mathrel{\hat{=}} b \cap \neg f(\top) \qquad\qquad c\tilde{f}(b) \mathrel{\hat{=}} b \cup \neg\tilde{f}(\bot)$$

Proposition 3.2.7. *Let the transformers be defined as in Definition 3.2.3.*

(i) The function cf is completely additive and deflationary.

(ii) The function $c\tilde{f}$ is completely multiplicative and inflationary.

(iii) The functions cf and $c\tilde{f}$ are De Morgan duals.

Proof. Property (i) follows from Proposition 3.2.1. Property (ii) then follows from De Morgan duality. For (iii), consider that $\neg cf(\neg b) = \neg(\neg b \cap \neg f(\top)) = b \cup f(\top) = b \cup \neg\tilde{f}(\bot) = c\tilde{f}(b)$. □

Each of the four transformers f, $\tilde{f}$, cf and $c\tilde{f}$ may be used to check bottom everywhere.

Theorem 3.2.8. *Let $f : B \to B$ be a deflationary, completely additive function on a Boolean algebra B. The following statements are equivalent.*

(i) f is bottom everywhere.

(ii) $\tilde{f}$ is top everywhere.

(iii) cf is the identity function.

(iv) $\tilde{cf}$ is the identity function.

Proof. We show that (i) and (ii) are equivalent. The transformer f is bottom everywhere exactly if for all $b \in B$ it holds that $f(b) = \bot$, which is true exactly if for all $b \in B$, $f(\neg b) = \bot$ (since complementation is a bijection). This is the case exactly if for all $b \in B$, $\neg f(\neg b) = \neg\bot$, which is equivalent to $\tilde{f}(b) = \top$.

We now show that (iii) is equivalent to (i). The transformer f is bottom everywhere exactly if for all $b \in B$, $f(b) = \bot$, which holds exactly if for all $b \in B$, $\neg f(b) = \top$. This is true exactly if for all $b \in B$, $b \cap \neg f(b) = b$, which is equivalent to $cf(b) = b$.

The equivalence between (iv) and (ii) can be shown by a symmetric argument. □

For convenience, we define a Boolean algebra with the above operators as a separate mathematical object.

Definition 3.2.4 (Bottom-Everywhere Algebra)**.** A *bottom-everywhere algebra* is the tuple

$$(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, \tilde{cf})$$

where all of the following hold.

(i) $(B, \subseteq, \cap, \cup)$ is a Boolean algebra.

(ii) f is a deflationary, completely additive function.

(iii) $\tilde{f}$ is the inflationary, completely multiplicative De Morgan dual of f.

(iv) cf is the deflationary, completely additive countertransformer of f.

(v) $\tilde{cf}$ is the inflationary, completely multiplicative countertransformer of $\tilde{f}$.

By Proposition 3.2.1, we may alternatively characterize conditions (ii)-(v) in a more strictly algebraic fashion, by requiring the existence of a *restriction* r and asserting that the following equalities hold for any $b \in B$.

$$\text{(ii) } f(b) = b \cap r \qquad \text{(iii) } \tilde{f}(b) = b \cup \neg r$$
$$\text{(iv) } cf(b) = b \cap \neg r \qquad \text{(v) } \tilde{cf}(b) = b \cup r$$

We may now give an expanded, equivalent definition of the bottom-everywhere problem.

The Bottom-Everywhere Problem

Let $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, \tilde{cf})$ be a bottom-everywhere algebra. The bottom-everywhere problem is to decide whether one of the following equivalent conditions hold.

$$\text{(i) } \forall b \in B.\ f(b) = \bot \qquad \text{(ii) } \forall b \in B.\ \tilde{f}(b) = \top$$
$$\text{(iii) } \forall b \in B.\ cf(b) = b \qquad \text{(iv) } \forall b \in B.\ \tilde{cf}(b) = b$$

We will see shortly that the bottom-everywhere problem generalizes logical satisfiability. The characterization above shows that there are many alternative formulations of bottom-everywhere. SAT is itself a special case of an entailment problem. We introduce an algebraic generalization of entailment below.

Definition 3.2.5 (r-Boundedness)**.** A function $f : B \to B$ is r *bounded* for an element $r \in B$, if $\forall b \in B.\ f(b) \subseteq r$.

The r-Boundedness Problem

Let $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, c\tilde{f})$ be a bottom-everywhere algebra and r be an element of B. The r-boundedness problem is to decide whether one of the following conditions hold.

$$\begin{array}{ll} \text{(i) } \forall b \in B.\ f(b) \subseteq r & \text{(ii) } \forall b \in B.\ \tilde{f}(b) \supseteq \neg r \\ \text{(iii) } \forall b \in B.\ cf(b) \supseteq b \cap \neg r & \text{(iv) } \forall b \in B.\ c\tilde{f}(b) \subseteq b \cup r \end{array}$$

Proposition 3.2.9. *The following conditions in the r-boundedness problem are equivalent.*

$$\begin{array}{ll} \textit{(i) } \forall b \in B.\ f(b) \subseteq r & \textit{(ii) } \forall b \in B.\ \tilde{f}(b) \supseteq \neg r \\ \textit{(iii) } \forall b \in B.\ cf(b) \supseteq b \cap \neg r & \textit{(iv) } \forall b \in B.\ c\tilde{f}(b) \subseteq b \cup r \end{array}$$

Proof.

$$\begin{array}{ll@{\qquad}ll} & \underline{\text{(i) iff (ii)}} & & \underline{\text{(iii) iff (iv)}} \\ & \forall b \in B.\ f(b) \subseteq r & & \forall b \in B.\ cf(b) \supseteq b \cap \neg r \\ \Longleftrightarrow & \forall b \in B.\ f(\neg b) \subseteq r & \Longleftrightarrow & \forall b \in B.\ cf(\neg b) \supseteq \neg b \cap \neg r \\ \Longleftrightarrow & \forall b \in B.\ \neg f(\neg b) \supseteq \neg r & \Longleftrightarrow & \forall b \in B.\ \neg cf(\neg b) \subseteq b \cup r \\ \Longleftrightarrow & \forall b \in B.\ \tilde{f}(b) \supseteq \neg r & \Longleftrightarrow & \forall b \in B.\ c\tilde{f}(b) \subseteq b \cup r \end{array}$$

$$\begin{array}{ll@{\qquad}ll} & \underline{\text{(iii) implies (i)}} & & \underline{\text{(i) implies (iii)}} \\ & \forall b \in B.\ cf(b) \supseteq b \cap \neg r & & \forall b \in B.\ f(b) \subseteq r \\ \Longrightarrow & cf(\top) \supseteq \neg r & \Longrightarrow & f(\top) \subseteq r \\ \Longrightarrow & \neg f(\top) \supseteq \neg r & \Longrightarrow & cf(\top) \supseteq \neg r \\ \Longrightarrow & f(\top) \subseteq r & \Longrightarrow & \forall b \in B.\ cf(\top) \cap b \supseteq b \cap \neg r \\ \Longrightarrow & \forall b \in B.\ f(b) \subseteq r & \Longrightarrow & \forall b \in B.\ cf(b) \supseteq b \cap \neg r \end{array}$$

□

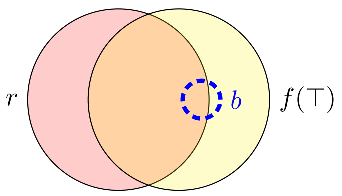
The r-boundedness problem is an algebraic entailment check. The bottom-everywhere problem is a special instance of checking r boundedness where r is $\bot$. Furthermore, checking r boundedness can be reduced to checking bottom-everywhere. In logic, checking whether a formula φ entails ψ reduces to checking the satisfiability of $\varphi \wedge \neg\psi$. Similarly, the r-boundedness problem for a function f can be reduced to checking if the transformer $\lambda b.\ f(b) \cap \neg r$ is bottom everywhere.

Theorem 3.2.10 (r-Boundedness via Bottom-Everywhere)**.** *Consider the bottom-everywhere algebra $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, c\tilde{f})$ and an element $r \in B$. Then $(B, \subseteq, \cap, \cup, g, \tilde{g}, cg, c\tilde{g})$, defined below, is a bottom-everywhere algebra.*

$$\begin{array}{ll} g(b) \mathrel{\hat{=}} f(b) \cap \neg r & \tilde{g}(b) \mathrel{\hat{=}} \tilde{f}(b) \cup r \\ cg(b) \mathrel{\hat{=}} cf(b) \cup r & c\tilde{g}(b) \mathrel{\hat{=}} c\tilde{f}(b) \cap \neg r \end{array}$$

The function f is r bounded exactly if g is bottom everywhere.

Proof. We first show that the above is a bottom-everywhere algebra. It is easy to see that g is completely additive and deflationary. The completely multiplicative, inflationary De Morgan dual of g is $\tilde{g}$, since $\neg g(\neg b) = \neg(f(\neg b) \cap \neg r) = \neg f(\neg b) \cup r = \tilde{f} \cup r = \tilde{g}$. The function cg is the



r-everywhere counterwitness b r-everywhere witness b

Figure 3.3: Witnesses and counterwitnesses for r boundedness.

countertransformer of g since $b \cap \neg g(b) = b \cap \neg(f(b) \cap \neg r) = b \cap \neg f(b) \cup r = cf(b) \cup r = cg(b)$. Its De Morgan dual is given by $\neg cg(\neg b) = \neg(cf(\neg b) \cup r) = \neg cf(\neg b) \cap \neg r = \tilde{cf}(b) \cap \neg r = \tilde{cg}(b)$.

Now assume that g is bottom everywhere. This is equivalent to $g(\top) \subseteq \bot$, which can be rewritten as $f(\top) \cap \neg r \subseteq \bot$. The above is equivalent to $f(\top) \subseteq r$, which in turn is equivalent to f being r bounded. □

Witnesses and Counterwitnesses

Often, it is not sufficient to solely determine whether a function is r bounded, but also to find elements of the lattice that may serve as a certificate for the result. We use the term *witness* to denote an element of the abstraction that certifies that r boundedness holds, and the term *counterwitness*, to denote an element that certifies that r boundedness does not hold. An r-boundedness check may be understood as a semantic check whether a property r holds at all elements of the lattice. A counterwitness is an algebraic counterexample, which shows an element that is not r bounded. A witness is an invariant property of all elements of the lattice that is sufficient to show that r holds.

Definition 3.2.6 (Counterwitness)**.** Given a bottom-everywhere algebra B as in Definition 3.2.4, a function $f : B \to B$ and elements $b, r \in B$ then b is an *r-boundedness counterwitness* if both
(i) b is a fixed point of f, i.e., $f(b) = b$ and
(ii) $b \not\subseteq r$.
A counterwitness b is *minimal* if there is no smaller element that is also an r-boundedness counterwitness.

Definition 3.2.7 (Witness)**.** Given a bottom-everywhere algebra B as in Definition 3.2.4, a function $f : B \to B$ and elements $b, r \in B$ then b is an *r-boundedness witness* if both
(i) b is a fixed point of $\tilde{cf}$, i.e., $\tilde{cf}(b) = b$ and
(ii) $b \subseteq r$.
A witness is *minimal*, if there is no smaller element that is also an r-boundedness witness.

Witnesses and counterwitnesses are schematically depicted in Figure 3.3.

Every r-boundedness problem has a witness exactly if f is r bounded and it has a counterwitness exactly if f is not r bounded.

Theorem 3.2.11. *The conditions (i)-(iii) below are equivalent in a bottom-everywhere algebra $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, \tilde{cf})$.*

(i) The function f is r bounded.

(ii) There is an r-boundedness witness in B.

(iii) There is no r-boundedness counterwitness in B.

Proof. We show that (i) and (ii) are equivalent and argue that $a = f(\top)$ is a witness exactly if f is r bounded. Assume that a is a witness. Then $\tilde{cf}(a) = a$ and $a \subseteq r$. We can characterize $\tilde{cf}(a)$ as $a \cup f(\top)$. Therefore, $a \cup f(\top) = a$ and $f(\top) \subseteq a$. Since $a \subseteq r$, it holds that f is r bounded. For the other direction, assume that f is r bounded. Then since $a = f(\top)$ it holds that $a \subseteq r$. Now consider $\tilde{cf}(a)$ which may be characterized as $a \cup f(\top)$. Since $f(\top) = a$, it follows that $\tilde{cf}(a) = a$, that is, a is a fixed point of $\tilde{cf}(a)$. Therefore, a is a witness.

We now show that (i) and (iii) are equivalent by arguing that, $a = f(\top)$ is a counterwitness exactly if f is not r bounded. Assume a is a counterwitness, then $a \not\subseteq r$. Therefore, f is not r bounded, since $f(\top) \not\subseteq r$. If f is not r bounded, then for some $b \in B$, the element $f(b)$ is not smaller than r. Therefore, by monotonicity, $f(\top) \not\subseteq r$ and a is a witness. □

If the underlying Boolean algebra is atomic, that is, if every element sits above some element that covers $\bot$, then minimal counterwitnesses can always be found. Minimal counterwitnesses are an algebraic generalization of the notion of a model of a formula. A minimal counterwitness, if it exists, is not necessarily unique.

The following statement gives conditions for the existence of minimal counterwitnesses.

Proposition 3.2.12. *Given an atomic bottom-everywhere algebra B as in Definition 3.2.4, then every completely additive, deflationary function f that is not r everywhere has a minimal r-boundedness counterwitness.*

Proof. If f is not r bounded, then $a = f(\top)$ is an r-boundedness counterwitness. Every complete atomic Boolean algebra is atomistic, that is, every element can be represented as the join of a set of atoms. Due to atomicity of B and additivity of f, $a = f(\top) = f(\bigcup A) = \bigcup_{q \in A} f(a)$ where A is the set of atoms of B. If for all atoms $q \in A$ it held that $f(q) \subseteq r$, then it would hold that $a \subseteq r$ which contradicts that a is an r-boundedness counterwitness. Therefore, there is some $q \in A$ such that $q \not\subseteq r$. Since q is an atom and hence covers $\bot$, it must hold that $f(q) = q$ and since $f(\bot)$ is equal to $\bot$, it holds that q is a minimal counterwitness. □

3.2.2 Satisfiability as Bottom-Everywhere

We now show that the satisfiability problem for a logic is an instance of the bottom-everywhere problem.

Proposition 3.2.13. *The transformers ded_φ and $cded_\varphi$ are deflationary and completely additive. The transformers abd_φ and $cabd_\varphi$ are inflationary and completely multiplicative.*

Proof. It is easy to see that ded_φ is deflationary. Consider a set $Q \subseteq \mathcal{P}(\mathcal{S})$. Then $ded_\varphi(\bigcup Q) = \{\sigma \mid \sigma \in \bigcup Q \wedge \sigma \models \varphi\} = (\bigcup Q) \cap (\{\sigma \mid \sigma \models \varphi\} = \bigcup_{S \in Q}(S \cap \{\sigma \mid \sigma \models \varphi\} = \bigcup_{S \in Q} ded_\varphi(S)\}$, therefore ded_φ is completely additive. The proof for $cded_\varphi$ is similar.

Dually, it's easy to see that abd_φ is inflationary. We have that $abd_\varphi(\bigcap Q) = \{\sigma \mid \sigma \in \bigcap Q \vee \sigma \not\models \varphi\} = (\bigcap Q) \cup \{\sigma \mid \sigma \not\models \varphi\} = \bigcap_{S \in Q}(S \cup \{\sigma \mid \sigma \not\models \varphi\} = \bigcap_{S \in Q} abd_\varphi(S)$, therefore abd_φ is completely multiplicative. The case for $cabd_\varphi$ is similar. □

Proposition 3.2.14. *The structure transformers defined by a logic $\mathcal{L}$ satisfy the equalities below.*

$$\begin{aligned} abd_\varphi &= \neg \circ ded_\varphi \circ \neg & \qquad \forall S \subseteq \mathcal{S}.\ cded_\varphi(S) &= S \cap \neg ded_\varphi(S) \\ cabd_\varphi &= \neg \circ cded_\varphi \circ \neg & \qquad \forall S \subseteq \mathcal{S}.\ cabd_\varphi(S) &= S \cup \neg abd_\varphi(S) \end{aligned}$$

If $\mathcal{L}$ has negation, then the following holds.

$$cded_\varphi = ded_{\neg\varphi} \qquad\qquad cabd_\varphi = abd_{\neg\varphi}$$

Together the transformers and the lattice $\mathcal{P}(\mathcal{S})$ form a bottom-everywhere algebra. Determining satisfiability and validity corresponds to deciding bottom everywhere and top everywhere.

Theorem 3.2.15. *Let $\mathcal{L} = (\mathcal{F}, \models, \mathcal{S})$ be a logic and let φ be a formula in $\mathcal{F}$. Then the following is a bottom-everywhere algebra.*

The Satisfiability Bottom-Everywhere Algebra for φ

$$(\mathcal{P}(\mathcal{S}), \subseteq, \cap, \cup, ded_\varphi, abd_\varphi, cded_\varphi, cabd_\varphi)$$

Each statement below is true exactly if φ is satisfiable.

(i) ded_φ is not bottom everywhere *(ii) abd_φ is not top everywhere*

(iii) $cded_\varphi$ is not identity *(iv) $cabd_\varphi$ is not identity*

Proof. It follows from Proposition 3.2.13, Proposition 3.1.3 and Proposition 3.2.14 that the above structure is a bottom-everywhere algebra. The transformer ded_φ is not bottom everywhere exactly if the formula φ has some model σ. The conditions (ii) to (iv) are equivalent to (i) by Theorem 3.2.8. □

It is easy to see that a counterwitness to a deduction transformer being bottom everywhere is a set of models.

Proposition 3.2.16. *If $S \subseteq \mathcal{S}$ is a counterwitness in the satisfiability bottom-everywhere algebra for φ, then S is a non-empty set of models. If S is a minimal counterwitness, then $S = \{\sigma\}$ where σ is a model.*

Proof. Assume that S is a counterwitness. Then $ded_\varphi(S) = S$ and S is non-empty. Therefore it holds that for all $\sigma \in S$, $\sigma \models \varphi$, and S is a set of models. It is easy to see that a minimal counterwitness corresponds to a singleton. □

3.2.3 Safety as Bottom-Everywhere

We now characterize the problem of deciding if a transition system $\mathcal{T}$ is safe w.r.t. a specification as an instance of the bottom-everywhere problem. Recall that a specification $\Xi \subseteq \Sigma$ is a subset of the set of states Σ that may not occur on any well-formed trace.

Recall that the powerset lattice of traces $\mathcal{P}(\Pi)$ is the concrete semantic domain for program analysis. We define a number of transformers on this lattice which provide the concrete semantic foundation for reasoning about the existence of safe traces. We recall below from Section 2.3.8 the definitions of the set I_* of initialized traces and the set of traces Ξ_* that satisfy the specification.

$$I_* \mathrel{\hat{=}} \{\pi \in \Pi \mid \exists \sigma \in \pi.\ \sigma \in I\} \qquad \Xi_* \mathrel{\hat{=}} \{\pi \in \Pi \mid \forall \sigma \in \pi.\ \sigma \in \Xi\}$$

We now define the trace safety transformers which can be used to reason about the existence of counterexample traces.

Definition 3.2.8 (Trace Safety Transformers)**.** For a transition system $\mathcal{T} = (\Sigma, \to, I)$ and a safety specification $\Xi \subseteq \Sigma$, we define the following transformers over $\mathcal{P}(\Pi)$.

Unsafe-Trace Transformer

$$utrace_{\mathcal{T},\Xi}(P) \mathrel{\hat{=}} \{\pi \mid \pi \in P \wedge \pi \notin \Xi_* \wedge \pi \text{ is well-formed in } \mathcal{T}\}$$

Safe-Trace Transformer

$$\mathit{strace}_{\mathcal{T},\Xi}(P) \mathrel{\hat{=}} \{\pi \mid \pi \in P \vee \pi \in \Xi_* \vee \pi \text{ is not well-formed in } \mathcal{T}\}$$

Counter Unsafe-Trace Transformer

$$\mathit{cutrace}_{\mathcal{T},\Xi}(P) \mathrel{\hat{=}} \{\pi \mid \pi \in P \wedge (\pi \in \Xi_* \vee \pi \text{ is not well-formed in } \mathcal{T})\}$$

Counter Safe-Trace Transformer

$$\mathit{cstrace}_{\mathcal{T},\Xi}(P) \mathrel{\hat{=}} \{\pi \mid \pi \in P \vee (\pi \notin \Xi_* \wedge \pi \text{ is well-formed in } \mathcal{T})\}$$

The above transformers may be characterized in terms of the trace transformers, which means that they can be approximated via classic program-analysis techniques.

Proposition 3.2.17. *For a transition system $\mathcal{T} = (\Sigma, \rightarrow, I)$ and safety specification Ξ_*, we may define the trace safety transformers as follows.*

$$\begin{aligned} \mathit{utrace}_{\mathcal{T},\Xi}(P) &\mathrel{\hat{=}} \neg\Xi_* \cap \mathsf{lfp}\, X.\ I \cup \mathit{tpost}_{\rightarrow}(X) \\ \mathit{strace}_{\mathcal{T},\Xi}(P) &\mathrel{\hat{=}} \neg I_* \cup \mathsf{gfp}\, X.\ \Xi_* \cap \mathit{t\widetilde{pre}}_{\rightarrow}(X) \end{aligned}$$

Proof. This follows from the characterization of the semantics $\|\mathcal{T}\|^*$ given in Proposition 2.3.14. □

The trace transformers form a bottom-everywhere algebra.

Theorem 3.2.18. *Let $\mathcal{T} = (\Sigma, \rightarrow, I)$ be a transition system and $\Xi_* \subseteq \Pi$ be a safety specification. Then the following is a bottom-everywhere algebra.*

The Trace-Safety Bottom-Everywhere Algebra

$$(\mathcal{P}(\Sigma), \subseteq, \cap, \cup, \mathit{utrace}_{\mathcal{T},R}, \mathit{strace}_{\mathcal{T},R}, \mathit{cutrace}_{\mathcal{T},R}, \mathit{cstrace}_{\mathcal{T},R})$$

Proof. The proof is identical to that of Theorem 3.2.15, only in place of the predicate $\models \varphi$ over the set of structures $\mathcal{S}$, we have the predicate $\pi \notin \Xi_* \wedge (\pi \text{ is well-formed})$ over the set of traces P. □

It is easy to see that $\mathcal{T}$ is safe w.r.t. Ξ_* exactly if the transformer $\mathit{utrace}_{\mathcal{T},\Xi_*}$ is bottom everywhere.

We can equally express safety as an instance of the r-boundedness problem, where r is the specification Ξ. For this purpose, we define the following variants of the above transformers which do not include the specification.

$$\begin{aligned} \mathit{utrace}_{\mathcal{T}}(P) &\mathrel{\hat{=}} \{\pi \mid \pi \in P \wedge \pi \text{ is well-formed}\} \\ \mathit{strace}_{\mathcal{T}}(\pi) &\mathrel{\hat{=}} \{\pi \mid \pi \in P \vee \pi \text{ is not well-formed}\} \\ \mathit{cutrace}_{\mathcal{T}}(P) &\mathrel{\hat{=}} \{\pi \mid \pi \in P \wedge \pi \text{ is not well-formed}\} \\ \mathit{cstrace}_{\mathcal{T}}(\pi) &\mathrel{\hat{=}} \{\pi \mid \pi \in P \vee \pi \text{ is well-formed}\} \end{aligned}$$

The safety of $\mathcal{T}$ w.r.t. Ξ is now equivalent to the condition that $\mathit{utrace}_{\mathcal{T}}$ is Ξ_* bounded.

In this setting, the notions of counterwitness and witness relate to those of inductive invariants and counterexample traces. An *inductive invariant* $\mathcal{I}$ is a subset of P such that $\mathit{tpost}_{\varphi}(\mathcal{I}) \subseteq \mathcal{I}$. It is *safe* if it holds that $\mathcal{I} \subseteq \Xi_*$. A *counterexample trace* is a trace $\pi \in \Pi$ such that π is a well-formed trace of $\mathcal{T}$ that is not in Ξ_*.

Theorem 3.2.19. *For a transition system $\mathcal{T}$, consider the following bottom-everywhere algebra.*

$$(\Pi, \subseteq, \cap, \cup, utrace_{\mathcal{T}}, strace_{\mathcal{T}}, cutrace_{\mathcal{T}}, cstrace_{\mathcal{T}})$$

The following statements hold for a set $P \subseteq \Pi$:

(i) If P is a Ξ_*-boundedness counterwitness then it contains a counterexample trace.

(ii) If P is a safe inductive invariant, then it is a Ξ_*-boundedness witness.

Proof. Recall that a counterwitness is an element P that is a fixed point of $utrace_{\mathcal{T}}$ and such that $P \not\subseteq \Xi_*$. Because a counterwitness P is a fixed point of $utrace_{\mathcal{T}}$, all traces in P are well-formed, and since $P \not\subseteq \Xi_*$, there is a trace $\pi \in P$ such that $\pi \notin \Xi_*$. It follows that π is a counterexample trace.

Let P be a safe inductive invariant. Then $tpost(P) \subseteq P$, therefore $\mathsf{lfp}\, X.\, I \cup tpost_{\rightarrow}(X) \subseteq P$. It then follows that P is a fixed point of $cutrace_{\mathcal{T}}$. Also, since P is a safe inductive invariant, it is contained within Ξ_*. Therefore, P is a Ξ_*-boundedness witness. □

3.2.4 Bottom-Everywhere via Abstract Fixed Points

This section shows how to approximately decide the bottom-everywhere problem. We first show that the bottom-everywhere property has fixed point characterizations. We then apply abstract interpretation to compute abstract fixed points, thereby allowing abstract checks of bottom everywhere.

Theorem 3.2.20 (Fixed Point Characterization of r-Boundedness)**.** *Consider a bottom-everywhere algebra $(B, \subseteq, \cup, \cap, f, \tilde{f}, cf, \tilde{cf})$. The conditions below are equivalent.*

(i) f is r bounded *(ii)* $\mathsf{gfp}\, f \subseteq r$ *(iii)* $\mathsf{lfp}\, \tilde{f} \supseteq \neg r$

Proof. Since f is idempotent, we have that $\mathsf{gfp}\, f = f(\top)$. By Proposition 3.2.1, we have that $f(b) = b \cap f(\top)$ for any $b \in B$. It is then easy to see that $f(b) \subseteq r$ for all $b \in B$ exactly if $f(\top) \subseteq r$. Therefore, (i) and (ii) are equivalent. Since both f and $\tilde{f}$ are closures, we have that $\mathsf{gfp}\, f = f(\top)$ and $\mathsf{lfp}\, \tilde{f} = \tilde{f}(\bot)$. We have that $f(\top) \subseteq r$ exactly if $\tilde{f}(\bot) \supseteq \neg r$. Therefore, (ii) and (iii) are equivalent. □

The fixed point characterization above may seem odd, since computing a greatest fixed point of a lower closure operator is equivalent to applying that operator once. The reason for this characterization is that we intend to work with abstractions of closure operators, which may not necessarily be closure operators themselves. Approximating a fixed point in the abstract leads to strictly more precision than applying the abstract function once.

The approximation technique we use is abstract interpretation. The concrete domain is the bottom-everywhere algebra $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, \tilde{cf})$, and an abstract domain is an abstract lattice, together with sound approximations of the transformers f, $\tilde{f}$, cf and $\tilde{cf}$.

Definition 3.2.9 (Abstract Bottom-Everywhere Algebra)**.** Let $(B, \subseteq, \cap, \sqcup, f, \tilde{f}, cf, \tilde{cf})$ be a bottom-everywhere algebra. An *overapproximate bottom-everywhere algebra* is an abstract domain

$$(O, \sqsubseteq, \sqcap, \sqcup, of, \tilde{of}, cof, \tilde{cof})$$

with functions $(\alpha_O : B \to O, \gamma_O : O \to B)$ such that the following holds.

$$(B, \subseteq) \underset{\alpha_O}{\overset{\gamma_O}{\longleftrightarrow}} (O, \sqsubseteq) \text{ with } \gamma_O(\bot_O) = \bot_B$$

$$of,\ \tilde{of},\ cof \text{ and } \tilde{cof} \text{ are transformers on } O, \text{ s.t.:}$$

$$\gamma_O \circ of \supseteq f \qquad \gamma_O \circ \tilde{of} \supseteq \tilde{f}$$

$$\gamma_O \circ cof \supseteq cf \quad \gamma_O \circ \tilde{cof} \supseteq \tilde{cf}$$

An *underapproximate bottom-everywhere algebra* is an abstract domain

$$(U, \sqsubseteq, \sqcap, \sqcup, uf, \tilde{uf}, cuf, c\tilde{uf})$$

with functions $(\alpha_U : B \to U, \gamma_U : U \to B)$ such that the following holds.

$$(B, \supseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (U, \sqsupseteq) \text{ with } \gamma_U(\top_U) = \top_B$$

$$uf,\ \tilde{uf},\ cuf \text{ and } c\tilde{uf} \text{ are transformers on } U\text{, s.t.:}$$

$$\gamma_U \circ uf \subseteq f \qquad \gamma_U \circ \tilde{uf} \subseteq \tilde{f}$$

$$\gamma_U \circ cuf \subseteq cf \quad \gamma_U \circ c\tilde{uf} \subseteq c\tilde{f}$$

We can now give conditions for using abstractions to determine if a transformer is bottom everywhere or is top everywhere.

Theorem 3.2.21 (Abstract r-Boundedness). *Let B, O and U be defined as in Definition 3.2.9. If one of (i)-(ii) holds, then f is r bounded.*

(i) $\gamma_O(\mathsf{gfp}\, of) \subseteq r$ *(ii)* $\gamma_U(\mathsf{lfp}\, \tilde{uf}) \supseteq \neg r$

Proof. This follows from the soundness result of abstract interpretation. If $\gamma_O(\mathsf{gfp}\, of) = \bot$, then by fixed point transfer (Theorem 2.3.9), we have that $\mathsf{gfp}\, f = \bot$ and by Theorem 3.2.20, f is bottom everywhere. Similarly, if $\gamma_U(\mathsf{lfp}\, \tilde{uf}) = \top$, then $\mathsf{lfp}\, \tilde{f} = \top$ and $\tilde{f}$ is top everywhere. From Theorem 3.2.8, we get that f is top-everywhere. □

The theorem above states that we may attempt to show that a function is bottom everywhere by computing a fixed point of an appropriate abstraction.

Abstract Witnesses and Counterwitnesses

We now show how the notions of witnesses and counterwitnesses translate to abstraction. The abstract loses precision w.r.t. to the concrete, and therefore direct lifting of the definitions of witness and counterwitness is insufficient. To ensure that an abstract witness corresponds to a concrete one we need a separate definition that takes completeness considerations into account. Recall from Section 2.3.7 that an abstraction of of a transformer f is γ-complete at an element a if $\gamma \circ of(a) = f \circ \gamma(a)$.

Definition 3.2.10 (Abstract Counterwitness). Let B, O and U be defined as in Definition 3.2.9.

(i) An element $o \in O$ is an abstract *r-boundedness* counterwitness if

  (a) $\gamma_O(o) \not\subseteq r$,

  (b) o is a fixed point of of, i.e, $of(o) = o$ and

  (c) of is γ-complete at o.

(ii) An element $u \in U$ is an abstract *r-boundedness* counterwitness if

  (a) $\gamma_U(u) \not\subseteq r$ and

  (b) u is a fixed point of uf, i.e, $uf(u) = u$.

Definition 3.2.11 (Abstract Witness). Let B, O and U be defined as in Definition 3.2.9.

(i) An element $o \in O$ is an abstract *r-boundedness* witness if

  (a) $\gamma_O(o) \subseteq r$ and

  (b) o is a fixed point of $c\tilde{of}$, i.e, $c\tilde{of}(o) = o$.

(ii) An element $u \in U$ is an abstract r-*boundedness* witness if

  (a) $\gamma_U(u) \subseteq r$,

  (b) u is a fixed point of $\widetilde{cuf}$, i.e, $\widetilde{cuf}(u) = u$ and

  (c) $\widetilde{cuf}$ is γ-complete at u.

Minimality of abstract witnesses and counterwitnesses is defined as expected.

The definitions above distinguish between overapproximations and underapproximations: Checking whether an element is a counterwitness can be more easily performed in an underapproximation whereas checking a witness is easier in an overapproximation. This is unsurprising. Remember that our intuition is that counterwitnesses are algebraic counterexamples to some semantic property and witnesses are a kind of invariant. Underapproximate reasoning is an appropriate for finding counterexamples while overapproximate reasoning is appropriate for finding invariant properties. Finding abstract witnesses or counterwitnesses is sufficient to show that r boundedness holds in the concrete, but their existence is not necessary. If the abstraction is too weak, it may neither contain witnesses nor counterwitnesses.

Theorem 3.2.22 (Abstract Witnesses)**.** *Let B, O and U be defined as in Definition 3.2.9. If there exists an abstract r-boundedness counterwitness in O or U then f is not r bounded. If there exists an abstract r-boundedness witness in O or U then f is r bounded.*

Proof. We show that abstract witnesses and counterwitnesses imply the existence of concrete witnesses, respectively, counterwitnesses. The result follows from Theorem 3.2.11.

Assume that $o \in O$ is an abstract r-boundedness counterwitness. Then $\gamma_O(o) \not\subseteq r$, $of(o)$ is γ-complete at o and $of(o) = o$. From γ-completeness, we have that $\gamma_O \circ of(o) = f \circ \gamma(o)$. Therefore $\gamma_O(o)$ is a fixed point of f since $f \circ \gamma_O(o) = \gamma_O \circ of(o) = \gamma_O(o)$. Furthermore, it holds that $\gamma_O(o) \not\subseteq r$ and since $f \circ \gamma_O(o) = \gamma_O(o) \circ f$ it follows that $f \circ \gamma_O(o)$ is a concrete counterwitness.

Now assume that $u \in U$ is an abstract r-boundedness counterwitness. Then $\gamma_U(u) \not\subseteq r$ and u is a fixed point of uf. We get from basic soundness and from the fact that f is deflationary that $\gamma_U(u)$ is a fixed point of f. Also, since $\gamma_U(u) \not\subseteq r$ holds, we have that $\gamma_U(u)$ is a concrete counterwitness.

The proof that the existence of abstract witnesses implies the existence of concrete witnesses is similar. □

3.3 Related Work

Abstract satisfaction is a framework for applying abstract interpretation to a generalization of the logical satisfiability problem. We are not aware of any previous work with closely related aims. In the following, we discuss efforts that are similar to isolated aspects of the work presented in this chapter.

Inference as Closure Formalizing deduction as application of closure operators is not new. An early example is Tarski's consequence operator *Cl* [163]. The operator *Cl* is an upper closure operator on the powerset lattice $(\mathcal{P}(\mathcal{F}), \subseteq)$, which maps a set of formulas to the set of its logical consequences. The fixed points of *Cl* are deductively closed sets, that is, they are the theories of the logic. In terms of abstract interpretation, *Cl* may be viewed as a reduction operator over the dual powerset lattice $(\mathcal{P}(\mathcal{F}), \supseteq)$, which overapproximates the concrete semantic lattice $(\mathcal{P}(\mathcal{S}), \subseteq)$. A set of formulas Φ abstractly represents the set of structures that satisfies all formulas in Φ. Since larger sets of formulas represent smaller sets of structures, the superset ordering $\supseteq$ is used. We assume that *Cl* only adds to a set of formulas F those formulas that are semantically entailed by formulas in F. Then *Cl* is a function that maps elements to more precise elements (i.e., larger sets), and for any set $\Phi \subseteq \mathcal{F}$, $Cl(\Phi)$ represents the same set of structures as Φ. Therefore, Cl is a semantic reduction operator.

In constraint satisfaction, it is known that deduction algorithms (which are called *filtering algorithms*) compute closures [149, 14]. The work in [14, 13] provides a semantic framework for constraint solving algorithms based on computing greatest fixed points over approximate lattices. The framework presented there is a limited instantiation of abstract interpretation to the special case of constraint solving, and is somewhat similar in spirit to the work in this chapter. The framework of [14, 13] focuses on formalizing a notion of abstract deduction.

The line of work in [22, 23, 24] presents an algebraic framework for search algorithms that characterizes search spaces as lattices and deduction operators as lower closures. The framework contains some equivalent developments and can be viewed as a limited instantiation of abstract interpretation that considers only abstract lattices. Concrete semantic information is represented implicitly via a lower closure operator. Similar to the work we will present in Chapter 4, a number of algorithms are characterized in the framework, including DPLL. The work also provides algebraic difficulty measures and general complexity results.

In contrast to these approaches, we give a full abstract interpretation account in a generalized algebraic setting and we give concrete and abstract semantics of various inference tasks, including abductive reasoning.

Abstract Interpretation of Logic Programs Abstract interpretation has been applied to the analysis of logic programs. A detailed overview of these efforts is given in [44]. An operational semantics for logic programs can be given in terms of a transition system [44]. A state of the transition system consists primarily of a sequence of goals. The transition relation models possible applications of some goal-search procedures such as SLD-resolution [111], which involve replacing a goal with a set of preconditions which are sufficient to derive that goal. From the transition relation, a collecting semantics can be defined.

In contrast to the work presented in this chapter, work on abstract interpretation of logic programs is focused on the procedural interpretation of logic programs as applications of goal-reduction procedures and on obtaining information about their behavior. This perspective is compatible with, yet orthogonal to the work presented in this section.

Logical Abstract Interpretation Our work is concerned with giving an abstract interpretation account of logical reasoning. *Logical abstract interpretation* [81, 82, 87, 86, 85, 84, 88, 83], on the other hand, is concerned with building abstract domains whose elements are built from logical formulas. This requires the construction of standard abstract domain operations for formulas. These operations include joins and widenings [88, 85] and abstract postcondition operators [85]. Use of logical abstract domains also enables new forms of domain combinations [85] in which the underlying logical languages are merged, rather then the abstract domains themselves.

Relating Logic and Abstractions It is well known that elements of abstract domains can be viewed as logical assertions [38, 39]. This idea is developed in detail by the line of work in [160, 158, 159, 157, 156] which elucidates the connections between logics and abstract domains. This work shows that logics can be extracted from abstract domains by giving the abstract partial ordering $\sqsubseteq$ a proof-theoretic interpretation and the concretization function γ a model-theoretic interpretation. In the other direction, constructions are given to obtain abstract domains in the standard Galois connection setting from a partially ordered set of logical assertions.

Generalizing Counterexamples In this chapter, we introduced concrete and abstract counterwitnesses, which generalize the notions of models in logic and counterexamples in program correctness. We now discuss existing work on abstract counterexamples in the context of model checking. In the dominant model checking approach to abstraction, a concrete transition system $\mathcal{T}$ is approximated by computing an abstract transition system $\mathcal{T}_A$ [7], where the set of states of $\mathcal{T}_A$ is a partition of the set of states of $\mathcal{T}$. Traces in the abstract system approximate sets of traces in the concrete. An abstract counterexample is a well-formed trace π over $\mathcal{T}_A$ whose concretization is not contained in the specification. The work in [116] lifts the notion of an abstract counterexample to powerset domains. The approaches in [121, 78] generalize counterexamples to abstract domains. Counterexamples are generalized to a sequence of abstract elements called a minimal sufficient explanation for failure [121]. Algebraic generalizations of these concepts have been studied in a different form in example-guided abstraction simplification [69] in an attempt to formalize and dualise CEGAR.

The notions of abstract counterexamples discussed above are related to our notion of an abstract counterwitness, but there is an essential difference. The existence of an abstract counterwitness in our framework implies that there is a concrete counterwitness (this is achieved by the completeness criteria on transformers). The existence of an abstract counterexample in model checking, does not imply the existence of a concrete counterexample. This difference in definition reflects a difference in focus. The purpose of abstract satisfaction is to allow generalization of satisfiability procedures. As we will see in Section 4.4, these procedures can be viewed as conflict-driven transformer refinement procedures, which trigger refinement when a "spurious" contradiction is derived (that is, one resulting from excessive underapproximation) and terminate when a counterwitness is found. Completeness arguments require that abstract counterwitnesses corresponds to concrete ones. The notion of an abstract counterexample is used within the context of counterexample-driven abstraction refinement procedures [35]. These procedures trigger refinement when spurious abstract counterexamples are found and terminate when a proof is found.

More concisely, in our case, an abstract counterwitness is a witness to the failure of the property under analysis, whereas an abstract counterexample is a witness to the failure to prove the property within the abstraction.

Chapter 4

The Algebraic Essence of Satisfiability Solvers

This chapter uses the framework of abstract satisfaction to show that popular satisfiability procedures can be understood in terms of lattices and fixed point computations. To put it more succinctly: We show that satisfiability solvers *are* abstract interpreters. The contributions of this chapter are as follows.

- We show that Boolean Constraint Propagation (BCP), the workhorse of modern satisfiability solvers, is a greatest fixed point computation, and that the lattices and transformers involved are well-known in program analysis.
- We show that the DPLL algorithm is an instance of a technique that uses partitions to refine abstractions. We identify refinement operators that generalize DPLL-style refinement to arbitrary abstract domains and relate these generalizations to existing work in program analysis.
- We show that Stålmarck's algorithm is a generic recipe for refining transformer approximations.
- We provide a high level perspective on the CDCL algorithm and show that it corresponds to a transformer refinement procedure which interleaves accelerated upward iteration in an abstract domain and accelerated downwards iteration in the downset completion of that domain.
- We show that propositional and theory solvers used within the DPLL(T) algorithm are abstract interpreters and that their combination constitutes an approximation of the reduced product between a domain for propositional reasoning and a domain for theory reasoning.

Motivation In the area of program verification, abstract interpretation and satisfiability-based software model checking are often seen as competing, orthogonal approaches. A common perception is that abstract interpretation is a means to obtain rough approximations of the program under analysis quickly, whereas satisfiability-based techniques provide accurate, albeit potentially expensive answers to semantic queries (this point of view is expressed, e.g., in [61, 101]).

While this perceived division reflects the focus of both researchers and practitioners in abstract interpretation and satisfiability, it obscures deep connections between the two fields. This chapter illuminates some of these connections by showing that a number of well-known procedures for propositional and first-order satisfiability have natural characterizations in the language of abstract interpretations.

The work described in this chapter provides a conceptual bridge between abstract interpretation and satisfiability research. This is not a matter of purely theoretical interest; this work has immediate practical implications, some of which we will explore in later chapters:

- It allows technology transfer from abstract interpretation to satisfiability research, for example the use of complex, semantic abstract domains as part of logical satisfiability algorithms.
- It allows technology transfer from satisfiability to abstract interpretation, by showing that decision procedure architectures are refinement procedures for abstract analyses.
- By their algebraic nature, abstract interpretation accounts of satisfiability solvers lead to simple generalizations of these procedures to richer logics or to entirely new problem domains.

Outline Section 4.1 shows that the BCP procedure is a fixed point computation in an abstract domain, that the unit rule is a straightforward abstraction of the deduction transformer and that BCP corresponds to a special case of constant propagation. In Section 4.2, we give an abstract interpretation account of DPLL as partition-based analysis refinement and illuminate the connection to partition-based program analysis techniques. Section 4.3 describes Stålmarck's proof procedure as an instantiation of a generic recipe for abstract transformer refinement. Section 4.4 shows that CDCL interleaves accelerated upwards iteration in a domain with accelerated downwards iteration in the downset completion of that domain. Section 4.5 shows that the theory and propositional components of DPLL(T) are abstract interpreters and that their combination is a reduced product. Section 4.6 discusses related work.

4.1 Boolean Constraint Propagation

Boolean Constraint Propagation (BCP) is a central component of the DPLL [53] and CDCL [118, 11, 137] algorithms. In this section, we show that BCP is a fixed point computation over an lattice that uses best abstract transformers and we show that BCP has precise counterparts in the program analysis literature. Section 4.1.1 provides a brief overview of BCP and shows that partial assignments form an abstract lattice. Section 4.1.2 shows that the unit rule is the best abstract transformer over partial assignments for a clause, and Section 4.1.3 shows that BCP computes a fixed point of this transformer. Section 4.1.4 informally discusses the similarities of BCP to existing techniques in program analysis.

4.1.1 An Overview of BCP

BCP is a process that deduces facts about a CNF formula φ by exhaustively applying a reasoning rule called the *unit rule* to update a data structure called a *partial assignment*. We illustrate the process with an example.

Example 4.1.1. *Consider the formula below.*

$$\varphi \mathrel{\hat{=}} p \wedge (\neg p \vee \neg q) \wedge (q \vee r \vee \neg s) \wedge (q \vee r \vee s)$$

Initially, nothing is known about the formula. From the clause p of φ, BCP *concludes that p must be true in every satisfying assignment. Since p must be true,* BCP *concludes that q must be false to satisfy the clause $\neg p \vee \neg q$. This sequence of deductions is given below.*

$$\rho_0 \mathrel{\hat{=}} \top \qquad \rho_1 \mathrel{\hat{=}} \langle p{:}\mathsf{t}\rangle \qquad \rho_2 \mathrel{\hat{=}} \langle p{:}\mathsf{t}, q{:}\mathsf{f}\rangle$$

All the remaining clauses contain more than one literal over an unassigned variable, so BCP *terminates.* BCP *is a sound but incomplete deduction procedure.* BCP *need not begin*

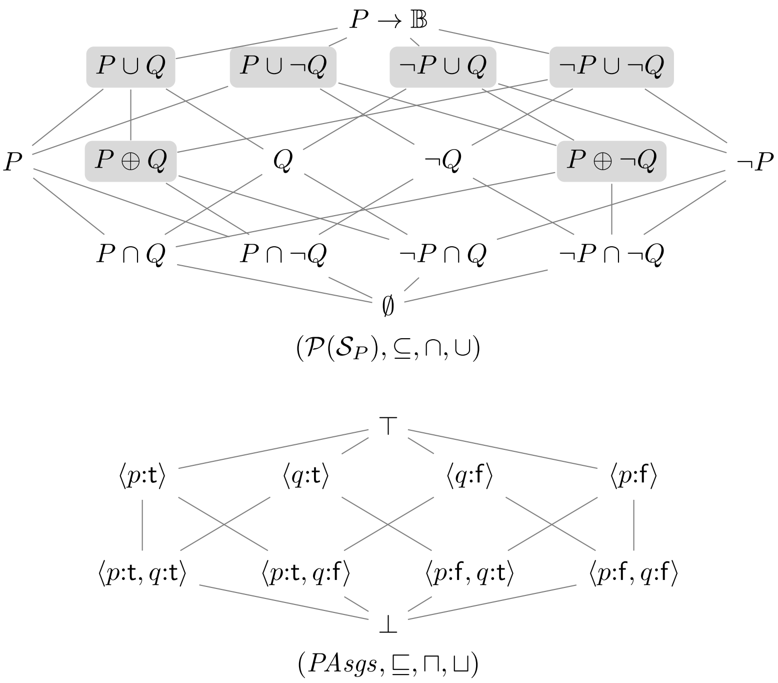

$P \to \mathbb{B}$
$P \cup Q$
$P \cup \neg Q$
$\neg P \cup Q$
$\neg P \cup \neg Q$
P
$P \oplus Q$
Q
$\neg Q$
$P \oplus \neg Q$
$\neg P$
$P \cap Q$
$P \cap \neg Q$
$\neg P \cap Q$
$\neg P \cap \neg Q$
$\emptyset$
$(\mathcal{P}(\mathcal{S}_P), \subseteq, \cap, \cup)$
$\top$
$\langle p{:}\mathsf{t}\rangle$
$\langle q{:}\mathsf{t}\rangle$
$\langle q{:}\mathsf{f}\rangle$
$\langle p{:}\mathsf{f}\rangle$
$\langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle$
$\langle p{:}\mathsf{t}, q{:}\mathsf{f}\rangle$
$\langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle$
$\langle p{:}\mathsf{f}, q{:}\mathsf{f}\rangle$
$\bot$
$(PAsgs, \sqsubseteq, \sqcap, \sqcup)$

Figure 4.1: Domains for assignments over p and q. The concrete domain $\mathcal{P}(\mathcal{S}_P)$ is above. The set P contains assignments that map p to true. The set Q is defined likewise. $P \oplus Q$ denotes symmetric difference. Elements $S \in \mathcal{P}(\mathcal{S}_P)$ that have no precise representation as partial assignments, i.e., where $\alpha \circ \gamma(S) \neq S$, are grayed out.

with ρ_0 as above. We can begin by assuming p is true, q is false and r is false, written $\rho \mathrel{\hat{=}} \langle p{:}\mathsf{t}, q{:}\mathsf{f}, r{:}\mathsf{f}\rangle$. Given ρ, BCP *concludes, from $(q \vee r \vee \neg s)$ that s must be false and from $(q \vee r \vee s)$ that s must be true. No assignment extending ρ satisfies φ. This situation, denoted $\bot$, is a* conflict.

Partial assignments are partial functions from the variables to truth values. Together with the conflict element $\bot$, partial assignments form a lattice.

Definition 4.1.1 (Partial Assignment)**.** The set of *partial assignments* is given below.

$$PAsgs \mathrel{\hat{=}} (P \to \{\mathsf{t}, \mathsf{f}, \top\}) \cup \{\bot\}$$

We denote by $\langle p_1{:}v_1, \ldots, p_k{:}v_k\rangle$ the partial assignment ρ where for each $1 \leq i \leq k$, $\rho(p_i) = v_i$ holds and where for all p that are not in $p_1, \ldots, p_k$, it holds that $\rho(p) = \top$. The set $\{\mathsf{t}, \mathsf{f}, \top\}$ has a natural partial order $\sqsubseteq$ where $\mathsf{t} \sqsubseteq \top$ and $\mathsf{f} \sqsubseteq \top$. The ordering can be extended to partial assignments as follows.

$$\rho_1 \sqsubseteq \rho_2 \text{ exactly if } \begin{cases} \rho_1 = \bot, \text{ or} \\ \rho_1 \text{ and } \rho_2 \text{ are not } \bot \text{ and } \forall p \in P.\ \rho_1(p) \sqsubseteq \rho_2(p) \end{cases}$$

Proposition 4.1.1. *The poset $(PAsgs, \sqsubseteq)$ is a complete lattice.*

In abstract interpretation parlance, partial assignments correspond to the Cartesian abstraction of Boolean variables [41]. Figure 4.1 depicts the lattice of partial assignments over two propositions.

A variant of the partial assignments domain is used for constant propagation [107]. The abstraction and concretization functions $\alpha : \mathcal{P}(\mathcal{S}_P) \to \mathit{PAsgs}$ and $\gamma : \mathit{PAsgs} \to \mathcal{P}(\mathcal{S}_P)$ below are standard and are known to form a Galois connection [45, 41].

$$\begin{array}{ll} \alpha(\emptyset) \mathrel{\hat{=}} \bot & \alpha(S) \mathrel{\hat{=}} \{p \mapsto \bigsqcup\{\sigma(p) \mid \rho \in S\} \mid p \in P\}, \text{ for } S \neq \emptyset \\ \gamma(\bot) \mathrel{\hat{=}} \emptyset & \gamma(\rho) \mathrel{\hat{=}} \{\sigma \in \mathcal{S}_P \mid \text{ for all } p \text{ in } P, \sigma(x) \sqsubseteq \rho(x)\}, \text{ for } \rho \neq \bot \end{array}$$

It is easy to see that every element of the abstraction is mapped to a distinct set of partial assignments. That is, the pair (α, γ) as defined above forms a Galois insertion.

The BCP algorithm iteratively refines a partial assignment ρ by applying the unit rule to derive new information. The unit rule compares a clause with the current partial assignment. If all literals of the clause are contradicted, the partial assignment is replaced by the conflict element $\bot$. If all but one literal are contradicted, the partial assignment is updated to reflect the fact that the remaining literal must be true. One iteration of BCP repeats this process for each clause in the formula. Subsequent iterations use the newly deduced information to refine the formula further. The basic algorithm is shown in Algorithm 1. Modern implementations use various techniques to improve efficiency, including the use of watched-literal data structures [137] which avoid the need to visit all clauses in every iteration of the main loop, or restarts [72].

Algorithm 1: Boolean constraint propagation.

```
in    : $\varphi$ – propositional CNF formula
        $\rho$ – partial assignment
out   : partial assignment

bcp($\varphi$,$\rho$)
    repeat
        $\rho' \leftarrow \rho$;
        foreach Clause $C \in \varphi$ do
            $\rho \leftarrow$ unit($C, \rho$);
        end
    until $\rho' = \rho$;
    return $\rho'$;
```

The unit rule is a sound deduction rule. From a partial assignment and a clause, it infers a new partial assignment. An algorithmic presentation of the unit rule is given in Algorithm 2. The algorithm checks the literals in clause C individually against a partial assignment ρ, to see whether each literal in C is contradicted by ρ. We extend ρ to literals by defining $\rho(\neg p) \mathrel{\hat{=}} \neg\rho(p)$ if $\rho(p)$ is $\mathtt{t}$ or $\mathtt{f}$, and $\rho(\neg p) \mathrel{\hat{=}} \top$ otherwise. If each literal is contradicted, then the procedure returns $\bot$, indicating a conflict. If all literals but one are contradicted, then the clause C is said to be *unit* under ρ, and the remaining literal u is designated as the *unit literal.*

The clause C has to evaluate to true in any model for the formula φ in order to be satisfiable. If all literals but u are contradicted, it can be deduced that u must evaluate to true and the partial assignment is modified accordingly. If the proposition $\mathit{var}(u)$ of the literal u is in positive phase, i.e., if $\mathit{phase}(\mathrm{u}) = \mathtt{t}$, then $\mathit{var}(\mathrm{u})$ is assigned to $\mathtt{t}$, otherwise, it is assigned to false. If two or more literals are not contradicted by ρ, then the procedure returns without changing ρ.

4.1.2 The Unit Rule as Best Abstract Transformer

In abstract interpretation terms, the unit rule is a decreasing transformer on the lattice of partial assignments. We define the *unit rule transformer* $\mathit{unit}_\varphi : \mathit{PAsgs} \to \mathit{PAsgs}$ for a

Algorithm 2: The unit rule.

```
in   : $C$ – propositional clause $C$
       $\rho$ – partial assignment
out  : partial assignment

unit($C$,$\rho$)
    $u \leftarrow$ null;
    foreach $Literal\ l \in C$ do
        if $\rho(l) = \mathtt{t}$ $or$ $\rho(l) = \top$ then
            if $u \neq$ null then
                return $\rho$;                    // more than one unit literal
            else
                $u = l$;                          // unit literal candidate found
            end
        end
    end
    if $u =$ null then return $\bot$;
    if $phase(var(u)) = \mathtt{t}$ then
        return $\rho[var(u) \leftarrow \mathsf{f}]$;
    else
        return $\rho[var(u) \leftarrow \mathsf{t}]$;
    end
```

formula φ inductively. First, for a proposition $p \in P$ we define the following.

$$unit_p(\rho) \mathrel{\hat{=}} \rho \sqcap \langle p : \mathsf{t}\rangle$$
$$unit_{\neg p}(\rho) \mathrel{\hat{=}} \rho \sqcap \langle p : \mathsf{f}\rangle$$

For a clause C, the unit rule checks whether each literal l is contradicted, i.e., whether $\gamma \circ unit_l(\rho) = \emptyset$. This condition is equivalent to $unit_l(\rho) = \bot$, since $\bot$ is the only representation of the empty set. We may then see that the unit rule corresponds to a straightforward abstraction of the concrete construction of Proposition 3.1.4, where disjunctions are replaced by joins.

$$unit_C(\rho) \mathrel{\hat{=}} \bigsqcup_{l \in C} unit_l(\rho)$$

Example 4.1.2. *Consider the clause $C = p \vee \neg q \vee r$ and the partial assignment $\rho = \langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle$. The clause C is unit under ρ, since its first two literals contradict the partial assignment and applying the unit rule yields the partial assignment* unit$(p \vee \neg q \vee r,\ \rho) = \langle p{:}\mathsf{f}, q{:}\mathsf{t}, r{:}\mathsf{t}\rangle$. *Observe that we may abstract the construction of Proposition 3.1.4 by replacing disjunctions with the abstract join to obtain exactly this result.*

$$\rho = \langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle \qquad \begin{aligned} unit_{p \vee q \neg q \vee r}(\rho) &= unit_p(\rho) \sqcup unit_{\neg q}(\rho) \sqcup unit_r(\rho) \\ &= (\rho \sqcap \langle p{:}\mathsf{t}\rangle) \sqcup (\rho \sqcap \langle q{:}\mathsf{f}\rangle) \sqcup (\rho \sqcap \langle r{:}\mathsf{t}\rangle) \\ &= \bot \sqcup \bot \sqcup (\rho \sqcap \langle r{:}\mathsf{t}\rangle) \\ &= \langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle \sqcap \langle r{:}\mathsf{t}\rangle = \langle p{:}\mathsf{f}, q{:}\mathsf{t}, r{:}\mathsf{t}\rangle \end{aligned}$$

Similarly, applying the unit rule to every clause in a formula corresponds to taking the meet of the unit transformers for all clauses.

$$unit_\varphi(\rho) \mathrel{\hat{=}} \bigsqcap_{C \in \varphi} unit_C(\rho)$$

Proposition 4.1.2. *For a propositional* CNF *formula φ, the function $unit_\varphi$ is a deflationary transformer on the lattice PAsgs.*

Proof. The $unit_l$ is clearly deflationary since it is defined by taking a meet between the argument and the element that represents l. The function $unit_C$ for a clause C is the join of deflationary transformers, and therefore itself a deflationary transformer. The function $unit_\varphi$ is then a deflationary transformer since it is the meet of a set of deflationary transformers. □

The unit rule soundly abstracts the concrete deduction transformer.

Proposition 4.1.3. *The unit rule $unit_\varphi : PAsgs \to PAsgs$ for a propositional* CNF *formula φ is a sound overapproximation of the concrete deduction transformer $ded_\varphi : \mathcal{P}(\mathcal{S}_P) \to \mathcal{P}(\mathcal{S}_P)$.*

Proof. Recall that $unit_\varphi$ is sound if $ded_\varphi \circ \gamma \subseteq \gamma \circ unit_\varphi$. We prove by induction on the structure of a CNF formula that $unit_\varphi$ is sound. First, consider a literal l. The transformer $unit_l$ is a sound approximation of ded_l if it holds that $ded_l \circ \gamma \subseteq \gamma \circ unit_l$.

$$\begin{aligned} ded_l \circ \gamma(\rho) &= \gamma(\rho) \cap \{\sigma \mid \sigma(var(l)) = phase(l)\} \\ &= \gamma(\rho) \cap \gamma(\langle var(l){:}phase(l)\rangle) \\ &= \gamma(\rho \sqcap \langle var(l){:}phase(l)\rangle) \\ &= \gamma \circ unit_l(\rho) \end{aligned}$$

From Proposition 3.1.4, we have for a clause C that $ded_C = \bigcup_{l \in C} ded_l$ and for a CNF formula φ that $ded_\varphi = \bigcap_{C \in \varphi} ded_C$. In a Galois connection, abstract joins and meets overapproximate their concrete counterparts. Since $unit_l$ soundly overapproximates ded_l and $unit_C$ and $unit_\varphi$ are given as $unit_C = \bigsqcup_{l \in C} unit_l$ and $unit_\varphi = \bigsqcup_{C \in \varphi} unit_C$, respectively. The result follows. □

Moreover, the unit rule is the most precise approximation of the deduction transformer ded_C for a clause C.

Theorem 4.1.4 (Unit Rule as a Best Abstraction)**.** *For a clause C, the unit rule $unit_C$ is the best abstract transformer of ded_C.*

Proof. We show that $\alpha \circ ded_C \circ \gamma = unit_C$.

$$\alpha \circ ded_C \circ \gamma(\rho) = \alpha \circ \bigcup_{l \in L} (ded_l \circ \gamma(\rho))$$

In a Galois connection, α distributes over joins.

$$= \bigsqcup_{l \in L} \alpha(ded_l \circ \gamma(\rho))$$

From the proof of Proposition 4.1.3, $ded_l \circ \gamma(\rho) = \gamma \circ unit_l(\rho)$.

$$= \bigsqcup_{l \in L} (\alpha \circ \gamma \circ unit_l(\rho))$$

Since (α, γ) is a Galois insertion, $\alpha \circ \gamma$ is the identity function.

$$= \bigsqcup_{l \in L} unit_l(\rho) = unit_C(\rho)$$

□

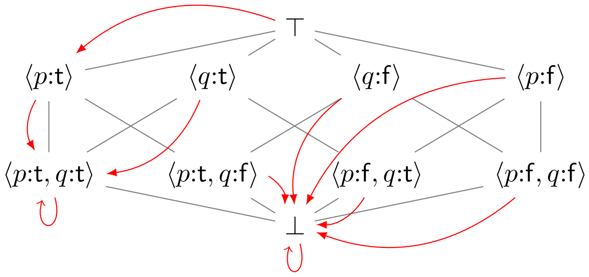


Figure 4.2: The partial assignments lattice for $P = \{p, q\}$. The red arrows represent the result of applying the unit rule transformer $unit_\varphi$ for the formula $\varphi = p \wedge (\neg p \vee q)$.

4.1.3 BCP as Fixed Point Computation

Applying the concrete deduction transformer ded_φ repeatedly yields the same result as applying it once, due to idempotence. In the abstract, iteration can provide strictly more precision.

Example 4.1.3. *We apply the unit rule for the formula $\varphi = p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$ to the empty partial assignment $\top$.*

$$\begin{aligned} unit_\varphi(\top) &= unit_p(\top) \sqcap unit_{\neg p \vee q}(\top) \sqcap unit_{\neg q \vee r}(\top) = \langle p{:}\mathsf{t}\rangle \sqcap \top \sqcap \top \\ &= \langle p{:}\mathsf{t}\rangle \end{aligned}$$

Reapplying $unit_\varphi$ to the result $\langle p{:}\mathsf{t}\rangle$ above yields strictly more precision

$$\begin{aligned} unit_\varphi(\langle p{:}\mathsf{t}\rangle) &= \langle p{:}\mathsf{t}\rangle \sqcap \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle \sqcap \langle p{:}\mathsf{t}\rangle \\ &= \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle \\ unit_\varphi(\langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle) &= \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle \sqcap \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle \sqcap \langle p{:}\mathsf{t}, q{:}\mathsf{t}, r{:}\mathsf{t}\rangle \\ &= \langle p{:}\mathsf{t}, q{:}\mathsf{t}, r{:}\mathsf{t}\rangle \end{aligned}$$

Another example is given in Figure 4.2, where the result of applying the transformer generated by a CNF formula is shown.

In Algorithm 3, we give another characterization of BCP, this time from an abstract interpretation point of view. The algorithm computes the greatest fixed point below an element ρ via Kleene iteration.

Algorithm 3: BCP as fixed point iteration.

```
in   : $\varphi$ – CNF formula $\varphi$
       $\rho$ – partial assignment
out  : partial assignment
bcp($\varphi$,$\rho$)
    repeat
        $\rho' \leftarrow \rho$;
        $\rho' \leftarrow \rho \sqcap unit_\varphi(\rho)$;
    until $\rho' = \rho$;
    return $\rho'$;
```

For the next theorem, we define a *parametric fixed point* function which maps an element to the greatest fixed point below that element.

$$\mathsf{pgfp}(f) \mathrel{\hat{=}} x \mapsto (\mathsf{gfp}\, Y.\ f(Y) \sqcap x)$$

Theorem 4.1.5 (BCP as a Greatest Fixed Point)**.** *Consider the function* `bcp` *defined by the procedure in Algorithm 1. The value returned by the algorithm* `bcp` *with arguments* φ *and* ρ *is equal to* $\mathsf{pgfp}(\mathit{unit}_\varphi)(\rho)$.

Proof. We denote by ρ_0 the initial value of the variable ρ. Recall that $\mathsf{pgfp}(\mathit{unit}_\varphi)(\rho_0)$ is equal to $\mathsf{gfp}\,X.\ \mathit{unit}_\varphi(X) \sqcap \rho_0$. For convenience, we define the function $f : \mathit{PAsgs} \to \mathit{PAsgs}$ as $f(X) \mathrel{\hat{=}} \mathit{unit}_\varphi(X) \sqcap \rho_0$. It remains to show that the value returned by the `bcp` algorithm is equal to $\mathsf{gfp}\,f$.

Since unit_φ is deflationary, it is easy to see that the statement $\rho \sqcap \mathit{unit}_\varphi(\rho)$ on the right-hand side of the assignment in Line 3 is equivalent to $\mathit{unit}_\varphi(\rho)$. Since the lattice is finite, it is clear that every run of the algorithm terminates after k iterations of the main loop. It is then easy to see that such a run computes the function $\mathit{unit}^k_\varphi(\rho_0)$. We now show that $\rho_k = \mathit{unit}^k_\varphi(\rho_0)$ is equal to $\mathsf{gfp}\,f$. The element ρ_k is a fixed point of unit_φ, otherwise, the algorithm would not have terminated after k iterations. It is also smaller than ρ_0, since unit_φ is deflationary. Therefore we have that $f(\rho_k) = \mathit{unit}_\varphi(\rho_k) \sqcap \rho_0 = \rho_k$, i.e., ρ_k is a fixed point of f.

It remains to show that ρ_k is the greatest fixed point of f. For every element in the sequence $(\mathit{unit}^i_\varphi(\rho_0))_{i\in\mathbb{N}}$ whose limit is ρ_k, there is a smaller element in the Kleene iteration sequence $(f^i(\top))_{i\in\mathbb{N}}$. We show this statement inductively. By monotonicity, the first element $\mathit{unit}^1_\varphi(\rho_0)$ is greater than $f^2(\top) = \rho_0 \sqcap \mathit{unit}_\varphi(\rho_0 \sqcap \mathit{unit}_\varphi(\top))$. Now assume that for every $1 \leq i \leq k$, it holds that $\mathit{unit}^i_\varphi(\rho_0)$ is greater than $f^{i+1}(\top)$. Then by monotonicity, $\mathit{unit}_\varphi(\mathit{unit}^i_\varphi(\rho_0))$ is greater than $\mathit{unit}_\varphi(f^{i+1}(\rho_0))$ which is equivalent to $\mathit{unit}_\varphi(f^{i+1}(\rho_0)) \sqcap \rho_0 = f^{i+2}$. This shows that $\mathit{unit}^{i+1}_\varphi$ is greater than f^{i+2}, which completes the inductive proof.

Since the greatest fixed point of f can equally be expressed as $\bigsqcap_{i\geq 0} f^i(\top)$ via Kleene's fixed point theorem (Theorem 2.3.10), and since $\rho_k = \bigsqcap_{i\geq 0} \mathit{unit}^i_\varphi(\rho_0)$, this implies that $\mathsf{gfp}\,f$ is smaller than ρ_k. Since both elements are fixed points, it follows that $\mathsf{gfp}\,f = \rho_k$. □

Modern implementation of SAT solvers do not use a generic fixed-point iteration engine but use efficient implementations based on watched-literal data structures [137]. Watched-literals are an essentially an effective, domain-specific way of implementing a working list: Instead of applying the global deduction transformer for the whole formula (which is the meet of the deduction transformer of all clauses), only clause transformers are applied that are likely to refine the current partial assignment. In the language of abstract interpretation, the watched-literals scheme is a semantics-aware form of chaotic iteration [44]: Program analyzers also do not apply the global state transformer for a whole program at once, but apply edge transformers in a sequence chosen to heuristically minimize required work.

We name the abstract transformer introduced above.

Definition 4.1.2 (BCP Transformer)**.** The BCP *transformer* for a propositional formula φ is the transformer $\mathit{bcp}_\varphi : \mathit{PAsgs} \to \mathit{PAsgs}$ defined as follows.

$$\mathit{bcp}_\varphi(\rho) \mathrel{\hat{=}} \mathsf{pgfp}(\mathit{unit}_\varphi)(\rho)$$

If $\mathit{bcp}_\varphi(\top)$ returns the $\bot$ element, then it follows from abstract satisfaction (Theorem 3.1.7) that the formula φ is unsatisfiable. If it returns an element different from $\bot$, then no conclusions can be drawn about the satisfiability of φ. As a satisfiability procedure, `bcp` is *refutationally sound*, but *incomplete*. That is, if `bcp` proves unsatisfiability, the formula is unsatisfiable, but if it fails to do so the formula may be unsatisfiable or satisfiable.

Example 4.1.4. *Consider the formula* $\varphi = p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg q \vee \neg r)$. *A run of* `bcp(`$\varphi$`, `$\rho$`)` *produces the following descending iteration sequence.*

$$\top \to \langle p{:}\mathsf{t}\rangle \to \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle \to \langle p{:}\mathsf{t}, q{:}\mathsf{t}, r{:}\mathsf{t}\rangle \sqcap \langle p{:}\mathsf{t}, q{:}\mathsf{t}, r{:}\mathsf{f}\rangle = \bot$$

The returned value is $\bot$, *indicating that the formula is unsatisfiable.*

Now consider the formula $\psi = (p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$. The formula ψ is unsatisfiable, since every truth assignment to p and q contradicts one of the clauses. A run of $\texttt{bcp(}\psi, \rho\texttt{)}$ *returns the initial element $\top$ since the unit rule does not provide any further refinement, even though ψ is unsatisfiable.*

The lack of completeness of BCP can be viewed in terms of transformer completeness.

Example 4.1.5. *Consider the unsatisfiable formula from the previous example $\psi = (p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg q)$.*

The transformer $bcp_\psi \mathrel{\hat{=}} \lambda\rho.\ \texttt{bcp(}\psi, \rho\texttt{)}$ is in general neither γ- nor α- complete, as shown below. Since ψ is unsatisfiable, ded_ψ is equal to $\bot$. Then $\alpha \circ ded_\psi(\mathcal{S}_P) = \bot$, but $bcp_\psi \circ \alpha(\mathcal{S}_P) = bcp_\psi(\top) = \top$. Therefore, bcp_ψ is not α-complete.

Now consider a formula $\varphi = p \vee q$. It holds that bcp_φ is α-complete at $\mathcal{S}_P$, since $\alpha \circ ded_\varphi(\mathcal{S}_P) = \top$ is the best representation of the set of models of φ as a partial assignment. However, it is not γ-complete though since $ded_\varphi \circ \gamma(\mathcal{S}_P) = \mathcal{S}_P \setminus \{\{p \mapsto \mathsf{f}, q \mapsto \mathsf{f}\}\}$, whereas $\gamma \circ bcp_\varphi(\top) = \gamma(\top) = \mathcal{S}_P$.

4.1.4 BCP and Static Program Analysis

We now compare BCP to existing research in static analysis of programs. The main result is that BCP corresponds to a well-known analysis in abstract interpretation.

Consider a program with a set of Boolean-valued variables. The set of memory states is then given by $\Omega \mathrel{\hat{=}} \mathit{Vars} \to \mathbb{B}$. Now consider a statement $\texttt{assume}(\varphi)$, where φ is a propositional formula which blocks executions if the memory state does not satisfy the formula φ. We formally define the semantics of a guard by associating it with the following transition relation $T_{\texttt{assume}(\varphi)} \in \Omega \times \Omega$.

$$(\omega, \omega') \in T_{\texttt{assume}(\varphi)} \iff \omega = \omega' \wedge \omega \models \varphi$$

Checking unsatisfiability of a formula φ is then equivalent to checking safety of an imperative program of the form below.

$$\texttt{assume}(\varphi); \texttt{assert}(\mathsf{f})$$

The symbol ; above denotes sequential composition, which is defined as usual and $\texttt{assert}(\mathsf{f})$ expresses a safety specification that is violated if some execution can reach the statement.

To determine safety of this program, we can compute the state semantics of the statement $\texttt{assume}(\varphi)$ using a strongest postcondition operator (see Section 2.3.8). The strongest postcondition of the assume statement is equivalent to the deduction transformer. This view will allow us to connect BCP to existing techniques in program analysis.

Proposition 4.1.6. *The following three transformers are equal.*

$$\text{(i) } post_{\texttt{assume}(\varphi)} \quad \text{(ii) } pre_{\texttt{assume}(\varphi)} \quad \text{(iii) } ded_\varphi$$

Proof. The equality of (i) and (ii) follows from the fact that $T_{\texttt{assume}(\varphi)}$ is symmetric. For the equality of (i) and (iii), consider the following.

$$\begin{aligned} post_{\texttt{assume}(\varphi)}(M) &= \{\omega \mid \exists\omega \in M.\ \omega = \omega' \wedge \omega \models \varphi\} \\ &= \{\omega \mid \omega \in M \wedge \omega \models \varphi\} \\ &= ded_\varphi(M) \end{aligned}$$

□

We now argue that BCP corresponds closely to a well-known program analysis technique. The lattice *PAsgs* is a well-known abstract lattice in static analysis. It is the lattice used in *constant propagation*, instantiated for the special case where all variables are Boolean.

Applying the unit rule to derive information over a guard is a special case of abstract interpretation with *backwards interpretation of Boolean expressions* [41]. In the context of constant propagation, it is known under the name *conditional constant propagation* [170]. Iteration of the strongest postcondition transformer to increase the precision of analyzing a guard is well-known in abstract interpretation as analysis with *locally decreasing iterations* [76].

4.2 DPLL

We now provide a characterization of the *Davis-Putnam-Logeman-Loveland algorithm* (DPLL) [53] in terms of lattices and transformers. DPLL is a propositional satisfiability algorithm that operates on partial assignments. Our characterization exposes the lattice-theoretic structure of the algorithm, and shows that it corresponds to a partition-based domain refinement recipe that has existing counterparts in the program analysis literature.

We first provide an overview of propositional DPLL in Section 4.2.1, with a brief discussion of its historical development and relevance. Section 4.2.2 informally shows parallels between the DPLL algorithm and a well-known program-analysis called trace partitioning. Section 4.2.3 gives a formalization of the procedure in terms of abstract interpretation and lattices which can be instantiated over a wide range of abstract domains and problems.

4.2.1 An Overview of DPLL

DPLL was originally conceived as an effective way of implementing the resolution-based DP procedure [54], which suffered from exponential worst-case space complexity. The algorithm introduced two refinements. First, DPLL uses the unit rule, which can be viewed as a form of restricted resolution in place of the unrestricted resolution rule used in DP. Second, DPLL uses a backtracking search on partial assignments to avoid the memory blowup of DP, rather than collecting sets of clauses like DP. This marked a shift in focus from viewing the satisfiability problem in terms of logical deduction to viewing it as a search problem.

Research interest in DPLL intensified in the late 1990s, when the introduction of several algorithmic, heuristic and engineering improvements led to the development of a sophisticated and elegant algorithmic framework known today as Conflict-Driven Clause Learning (CDCL) [11, 118, 137]. CDCL is the basis of today's propositional satisfiability procedures, which can solve problems that are many orders of magnitude larger than those solvable by DPLL, and are important components of many theorem proves for richer logics, such as SMT solvers. Historically, CDCL can be viewed as an extension of DPLL, which sometimes leads to CDCL solvers being referred to as DPLL procedures or DPLL-based procedures. The two algorithms differ significantly though, not only in speed and efficiency, but also in terms of how the search is conducted. While DPLL is, in modern terms, a straightforward branch-and-bound search algorithm, CDCL combines search and a property-directed form of resolution in a way that elegantly interleaves search for models with search for proofs. In this section, we will focus on "classic" DPLL and we defer all discussions of learning to Section 4.4, where we discuss CDCL in terms of abstract interpretation, and Chapter 5, where we generalize CDCL to abstract domains.

A basic, recursive formulation of DPLL is shown in Algorithm 4. The algorithm first calls BCP to refine the current partial assignment. If the result is a conflict, the algorithm returns the conflict element $\bot$ and if the result is a total assignment, the algorithm returns this assignment. Otherwise, a decision is made wherein a variable p that is assigned to $\top$ is heuristically selected. The algorithm then issues two recursive calls where the chosen variable is assigned, first to true, then to false. The calls determine whether the elements $\rho \sqcap \langle p{:}\mathsf{t}\rangle$ and $\rho \sqcap \langle p{:}\mathsf{f}\rangle$ abstractly represent some model of φ. If either of the two calls returns with a result that is not equal to the conflict element $\bot$, the algorithm returns this result, otherwise, it returns $\bot$, indicating that no solution could be found that refines ρ.

Algorithm 4: Classic DPLL.

```
in   : $\varphi$ – propositional CNF formula
       $\rho$ – partial assignment
out  : partial assignment

dpll($\varphi$,$\rho$)
    $\rho \leftarrow$ bcp($\varphi$, $\rho$);
    if $\rho = \bot$ then return $\bot$;
    if for all $p$ in $P$ $\rho(p) \neq \top$ then return $\rho$;
    $p \leftarrow$ decide($\rho$);
    $\rho' \leftarrow$ dpll($\varphi$, $\rho \sqcap \langle p$:t$\rangle$);
    if $\rho' \neq \bot$ then return $\rho'$;
    $\rho' \leftarrow$ dpll($\varphi$, $\rho \sqcap \langle p$:f$\rangle$);
    if $\rho' \neq \bot$ then return $\rho'$;
    return $\bot$;

decide($\rho$)
    Choose $p \in P$ s.t. $\rho(p) = \top$;
    return $p$;
```

$$\varphi = p \wedge (\neg p \vee q \vee \neg r) \wedge (q \vee r) \wedge (s \vee r) \wedge (\neg q \vee s \vee \neg r) \wedge (\neg s \vee r)$$

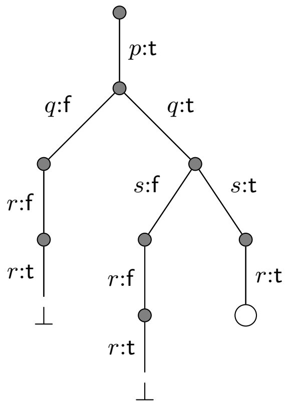


Figure 4.3: Decision tree corresponding to a complete run of DPLL on φ.

Runs of DPLL can be visualized using a decision tree, as shown in Figure 4.3, which captures a full execution of DPLL. A preorder traversal of the decision tree provides a history of partial assignments considered during search. At the root node, nothing is known, which represents the state of the algorithm before execution begins. In the first BCP phase, the proposition p is assigned to true. Then as a decision, q is chosen and assigned to $\mathtt{f}$. This assignment results in a conflict, since r is assigned to both true and false. The algorithm backtracks and flips the value of the decision variable q to $\mathtt{t}$. After another decision and a second backtrack, the algorithm finds the element $\langle p{:}\mathtt{t}, q{:}\mathtt{t}, s{:}\mathtt{t}, r{:}\mathtt{t}\rangle$, which represents a model of the formula.

4.2.2 Trace Partitioning and DPLL

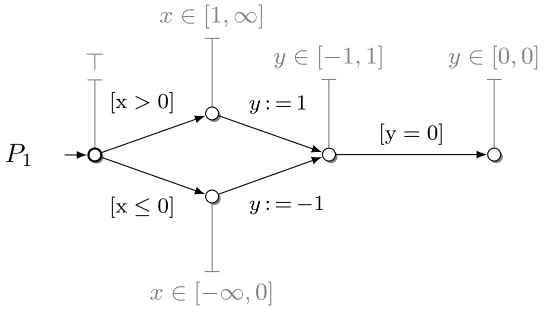
We now relate DPLL to existing techniques in program analysis. Recall that we can view checking if a formula φ is satisfiable as checking if the program $\mathtt{if}(\varphi)$ $\mathtt{assert(f)}$ is safe. If BCP is viewed as a static analysis in this way, DPLL can be understood analogous to analyzing the sequence of programs below.

$$P_0 \mathrel{\hat{=}} \begin{array}{l}\mathtt{if}(\varphi)\\ \quad\mathtt{assert(f)}\end{array} \qquad P_1 \mathrel{\hat{=}} \begin{array}{l}\mathtt{if}(p)\ P_0\\ \mathtt{else}\ P_0\end{array} \qquad P_2 \mathrel{\hat{=}} \begin{array}{l}\mathtt{if}(q)\ P_1\\ \mathtt{else}\ P_1\end{array}$$

DPLL uses decisions to dynamically restrict the range of values, in order to improve the precision of the analysis. In abstract interpretation, this corresponds to a well-known technique called *trace partitioning* [119, 151]. Trace partitioning technique for refining abstract analyses that is parametrized by a base analysis and a partition of the set of traces. The base analysis is then extended to obtain information separately about each of the partitions, which is more precise than simply running the base analysis.

Trace partitioning is a means to increase the precision of the analysis without changing the underlying abstraction. It is essentially a form of case analysis.

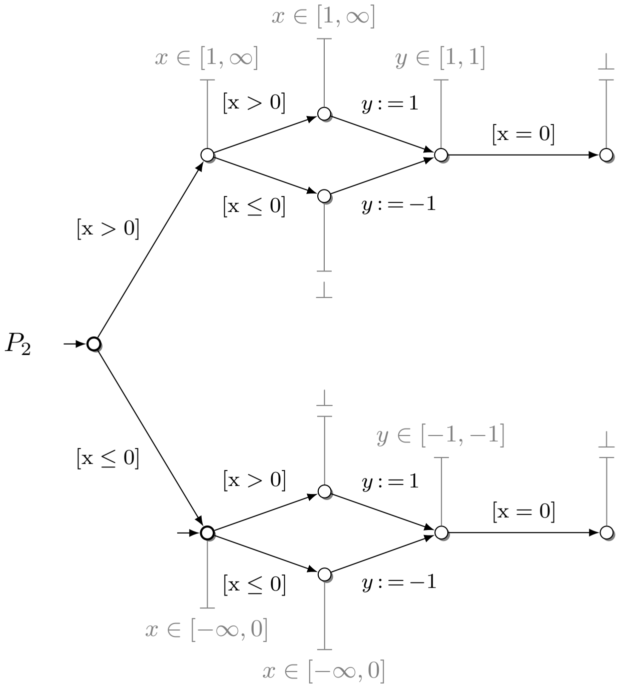
Example 4.2.1 (Trace Partitioning). *Consider the program below, whose nodes are annotated with the result of forward abstract interpretation over the domain of intervals.*



The variable y is assigned to either 1 or -1, depending on the value of x. Interval analysis imprecisely overapproximates this information and finds that y is in the range $[-1, 1]$ after the join point. Due to the precision loss, the analysis is unable to determine that y is distinct from 0 at the last control-flow node.

Analysis precision can be increased by using trace partitioning where we distinguish between (i) the set of traces where x is initially positive and (ii) the set of traces where x is initially zero or negative.

We simulate a trace partitioning by running the original analysis over the following modified CFG*, in which the case split is represented by a fresh control-flow branch.*



The CFG P_2 *duplicates the original* CFG P_1 *and we introduce a new initial node from which a case distinction originates. The upper branch assumes that* x *is strictly positive, the lower branch assumes the opposite. Analyzing* P_2 *corresponds to analyzing* P_1 *twice under different assumptions. This form of* CFG *transformation is sound, since no trace is lost: The final location in* P_1 *is reachable exactly if one of the two final locations in* P_2 *is reachable.*

Abstract analysis of P_2 *is more precise than abstract analysis of* P_1 *though: Standard interval analysis can establish that both final locations in* P_2 *are unreachable (indicated by the annotation with the abstract value* $\bot$*), but is unable to establish the same in* P_1*.*

Trace partitioning is a framework with different instantiations. Creating partitions that restrict the range of a variable at one or more program locations (as is implicitly done in the example above) is referred to as *value-based partitioning.* In terms of classic abstract interpretation, trace partitioning may be viewed as a variant of the cardinal power construction [43].

In DPLL, the set of cases is not known a priori, but is constructed on the fly. This corresponds, in terms of abstract interpretation, to a form of *dynamic, value-based trace partitioning.*

4.2.3 Abstract Partitioning

We now formalize domain refinement based on partitioning. Since we are not only interested in traces, but also partitions of logical concrete domains, we simply use the term *abstract partitioning* instead of trace partitioning.

Definition 4.2.1 (Abstract Coverings and Partitions)**.** Let C be a complete Boolean algebra, let A be a complete lattice that abstracts C with Galois connection (α, γ) and let $\chi \subseteq A$. The set χ is an *abstract covering* of an element $c \in C$ if it is non-empty and $\bigcup_{a \in \chi} \gamma(a) \supseteq c$. The set χ is an *abstract partition* of c if it is an abstract covering and if for all $a \neq a'$ in χ, $\gamma(a) \cap \gamma(a')$ is $\bot$. A set χ is an *abstract covering*, if it is an abstract covering of $\top$, and it is an *abstract partition* if it is an abstract partition of $\top$.

A covering can be used to refine an abstract analysis using cases. Each element of the covering constitutes one semantically specified case that may be analyzed separately.

Example 4.2.2. *Consider the formula $\varphi = (p \vee q) \sqcap (p \vee \neg q)$. We consider static analysis of this formula using the partial assignments abstract domain which is equivalent to deduction using the unit rule. Deduction on φ using the unit rule doesn't yield any information.*

$$unit_\varphi(\top) = (\langle p{:}\mathsf{t}\rangle \sqcup \langle q{:}\mathsf{t}\rangle) \sqcap (\langle p{:}\mathsf{t}\rangle \sqcup \langle q{:}\mathsf{f}\rangle) = \top \sqcap \top = \top$$

On the other hand, instead of applying the transformer once to $\top$, we can apply it twice to $m = \langle q{:}\mathsf{t}\rangle$ and $m' = \langle q{:}\mathsf{f}\rangle$. Since it holds that $\gamma(m) \cup \gamma(m') = \gamma(\top)$, it is sound to use the combined results of $unit_\varphi(m)$ and $unit_\varphi(m')$ instead of $unit_\varphi(\top)$ when performing entailment checks. We obtain the following.

$$\begin{tikzcd}[column sep=tiny] & {} \arrow[dl, "unit_\varphi(\langle q{:}\mathsf{t}\rangle)"'] \arrow[dr, "unit_\varphi(\langle q{:}\mathsf{f}\rangle)"] & \\ \begin{array}{l} (\langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle \sqcup \langle q{:}\mathsf{t}\rangle) \sqcap (\langle p{:}\mathsf{t}\, q{:}\mathsf{t}\rangle \sqcup \bot) \\ = \langle q{:}\mathsf{t}\rangle \sqcap \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle = \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle \end{array} & & \begin{array}{l} (\langle p{:}\mathsf{t}, q{:}\mathsf{f}\rangle \sqcup \bot) \sqcap (\langle p{:}\mathsf{t}, q{:}\mathsf{f}\rangle \sqcup \langle q{:}\mathsf{f}\rangle) \\ = \langle p{:}\mathsf{t}, q{:}\mathsf{f}\rangle \sqcap \langle q{:}\mathsf{f}\rangle = \langle p{:}\mathsf{t}, q{:}\mathsf{f}\rangle \end{array} \end{tikzcd}$$

We now formalize the above example by introducing an abstract partitioning domain.

Definition 4.2.2 (Abstract Partition Lattice)**.** Let C and A be as above and let $\chi \subseteq A$ be an abstract covering. The *abstract partition lattice* of A and χ is the lattice $(A^\chi, \sqsubseteq)$ defined below.

$$A^\chi \mathrel{\hat{=}} \chi \to A \qquad\qquad f \sqsubseteq g \iff \forall a \in \chi.\ f(a) \sqsubseteq g(a)$$

$$\bigsqcap F = \lambda a.\ \bigsqcap_{f \in F} f(a) \qquad\qquad \bigsqcup F = \lambda a.\ \bigsqcup_{f \in F} f(a)$$

Proposition 4.2.1. *The abstract partitioning lattice A^χ overapproximates C w.r.t. the following Galois connection.*

$$(C, \subseteq) \underset{\alpha_\chi}{\overset{\gamma_\chi}{\leftrightarrows}} (A^\chi, \sqsubseteq)$$

$$\alpha_\chi(c) \mathrel{\hat{=}} \{a \mapsto \alpha(\gamma(a) \cap c) \mid a \in \chi\} \quad \gamma_\chi(f) \mathrel{\hat{=}} \bigcup_{a \in \chi} \gamma(a) \cap \gamma(f(a))$$

Proof. Since both α_χ and γ_χ are constructed from monotone operations and functions, it is easy to see that both are monotone. We now show that $\gamma_\chi \circ \alpha_\chi$ is inflationary.

$$\begin{aligned} \gamma_\chi \circ \alpha_\chi(c) &= \bigcup_{a \in \chi} (\gamma(a) \cap \gamma(\alpha(\gamma(a) \cap c)) \\ &\supseteq \bigcup_{a \in \chi} (\gamma(a) \cap (\gamma(a) \cap c)) \\ &= \bigcup_{a \in \chi} (\gamma(a) \cap c) \\ &= c \cap \bigcup_{a \in \chi} \gamma(a) = c \cap \top = c \\ \implies \gamma_\chi \circ \alpha_\chi(c) \supseteq c \end{aligned}$$

We complete the proof that $(\alpha_\chi, \gamma_\chi)$ is a Galois connection by showing that $\alpha_\chi \circ \gamma_\chi$ is deflationary.

$$\alpha_\chi \circ \gamma_\chi(f)(a) = \alpha(\gamma(a) \cap (\bigcup_{a' \in \chi} \gamma(a') \cap \gamma(f(a')))$$

$$\begin{aligned}
&= \alpha((\bigcup_{a'\in\chi} \gamma(a) \cap \gamma(a') \cap \gamma(f(a')))\\
&= \alpha(\bigcup_{a'\in\chi} \gamma(a \sqcap a' \sqcap f(a')))\\
&= \bigsqcup_{a'\in\chi} \alpha(\gamma(a \sqcap a' \sqcap f(a')))\\
&\sqsubseteq \bigsqcup_{a'\in\chi} a \sqcap a' \sqcap f(a') \sqsubseteq a\\
\Longrightarrow\ & \alpha_\chi \circ \gamma_\chi(f) \sqsubseteq f
\end{aligned}$$

□

Dynamic Partitioning

The partitioning domain described above is an abstract interpretation variant of case analysis. The abstract partition (or covering) provides the set of cases and each case is mapped to its corresponding analysis result. Algorithms such as DPLL do not use a fixed set of cases but construct the set of cases on the fly, based on the results of analysis itself: cases are only introduced when necessary.

One way to model this is to consider it algorithmically as abstraction refinement: If the result of analysis is insufficiently precise the abstract partition domain is refined by choosing a more precise set of cases.

We focus on another perspective in this section, namely on that of *dynamic partitioning* [21], where all possible refinements are viewed as elements of the same lattice. In this view, case refinements are semantics-preserving steps down this lattice. We may partially order abstract partitions.

Definition 4.2.3 (Partition Ordering)**.** Let $(A, \sqsubseteq)$ be an abstract domain with $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$. For two abstract partitions χ_1 and χ_2, we define $\chi_1 \preceq \chi_2$ exactly if for all $a \in \chi_2$ there exists a subset $\chi_1' \subseteq \chi_1$ satisfying both conditions below.

(i) For all $a' \in \chi_1'$, it holds that $a' \sqsubseteq a$.

(ii) χ_1' is an abstract partition of $\gamma(a)$.

The set of abstract partitions that are definable by partial assignments forms a complete lattice, as shown in Figure 4.4.

We now define an abstraction in which the set of partitions may be changed dynamically.

Definition 4.2.4 (Dynamic Abstract Partition Lattice)**.** Let $(A, \sqsubseteq)$ be an abstract domain over a complete Boolean algebra $(C, \subseteq)$ with $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$. Let Λ be a set of abstract partitions over A such that $(\Lambda, \preceq)$ is a complete lattice with meet $\curlywedge$ and join $\curlyvee$. The *dynamic abstract partition lattice* $(\mathbb{D}(A, \Lambda), \dot{\sqsubseteq})$ is given below.

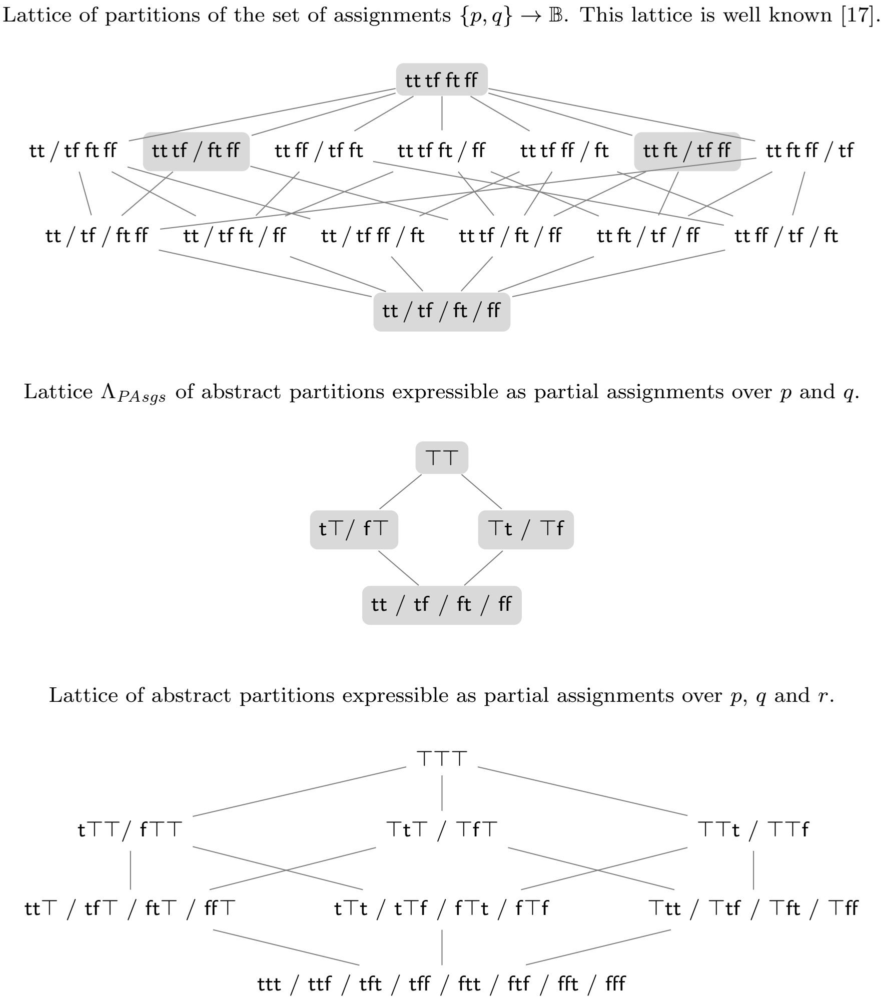
$$\mathbb{D}(A, \Lambda) \mathrel{\hat{=}} \bigcup_{\chi\in\Lambda} (\chi \to A)$$

$$\text{For } f : \chi_f \to A,\ g : \chi_g \to A \in \mathbb{D}(A, \Lambda),$$

$$f \mathrel{\dot{\sqsubseteq}} g \text{ exactly if } \chi_f \preceq \chi_g \text{ and } \forall a \in \chi_f, b \in \chi_g.\ a \sqsubseteq b \implies f(a) \sqsubseteq g(b)$$

$$\dot{\bigsqcap} F \mathrel{\hat{=}} \{a \mapsto \bigsqcap Q \mid a \in \bigcurlywedge_{f:\chi\to A\in F} \chi \text{ and } Q = \{f(a') \mid f \in F \text{ and } a' \in \chi_f \text{ and } a' \sqsupseteq a\}\}$$

$$\dot{\bigsqcup} F \mathrel{\hat{=}} \{a \mapsto \bigsqcup Q \mid a \in \bigcurlyvee_{f:\chi\to A\in F} \chi \text{ and } Q = \{f(a') \mid f \in F \text{ and } a' \in \chi_f \text{ and } a' \sqsubseteq a\}\}$$



Lattice of partitions of the set of assignments $\{p, q\} \rightarrow \mathbb{B}$. This lattice is well known [17].

Lattice Λ_{PAsgs} of abstract partitions expressible as partial assignments over p and q.

Lattice of abstract partitions expressible as partial assignments over p, q and r.

Figure 4.4: Lattice of partitions of the concrete domain of propositional assignments (above) and of the abstract domains of partial assignments for two and three propositions. We denote by tt the function $\{p \mapsto \mathsf{t}, q \mapsto \mathsf{t}\}$ and denote partition boundaries by "/". Note that not all concrete partitions can be expressed as abstract partitions. The shaded concrete partitions are those expressible in the abstract.

Proposition 4.2.2. $(\mathbb{D}(A,\Lambda), \dot{\sqsubseteq})$ *is a complete lattice.*

Proof. We skip the proof that $\dot{\sqsubseteq}$ is a partial ordering.

We first show that $\dot{\bigsqcap} F : \chi_{\dot{\sqcap}} \rightarrow A$ is a lower bound of F. Consider a function $f : \chi_f \rightarrow A \in F$. Then we have by definition that $\chi_{\dot{\sqcap}} = \bigcurlywedge_{g:\chi \rightarrow A \in F} \chi$, and since $f \in F$, it holds that $\chi_{\dot{\sqcap}} \preceq \chi_f$. Now consider $a \in \chi_{\dot{\sqcap}}, a' \in \chi_f$ such that $a \sqsubseteq a'$. Then we have that $(\dot{\bigsqcap} F)(a) = \bigsqcap Q$ where $Q = \{f(a') \mid f \in F \wedge a' \in \chi_f \wedge a' \sqsupseteq a\}$. Clearly, it holds that $f(a') \in Q$, therefore $(\dot{\bigsqcap} F)(a) \sqsubseteq f(a')$. We have shown that $\chi_{\dot{\sqcap}} \preceq \chi_f$ and that for any $a \in \chi_{\dot{\sqcap}}, a' \in \chi_f$ such that $a \sqsubseteq a'$, it holds that $(\dot{\bigsqcap} F)(a) \sqsubseteq f(a')$. Therefore, $\dot{\bigsqcap} F \,\dot{\sqsubseteq}\, f$ for any $f \in F$ and $\dot{\bigsqcap} F$ is a lower bound of F. It remains to show that $\dot{\bigsqcap} F$ is the greatest lower bound. Consider a lower bound $g : \chi_g \rightarrow A$ of F. Since $\chi_{\dot{\sqcap}}$ is a greatest lower bound in the lattice of partitions and since for all $f \in F$, $g \,\dot{\sqsubseteq}\, f$ implies that $\chi_g \preceq \chi_f$, it follows that $\chi_g \preceq \chi_{\dot{\sqcap}}$. Now let $a \in \chi_g, a' \in \chi_{\dot{\sqcap}}$ such that $a \sqsubseteq a'$. Since g is a lower bound of F, we have that $g(a) \sqsubseteq f(b)$ for all $f : \chi_f \rightarrow A \in F, b \in \chi_f$ where $a \sqsubseteq b$. Therefore, $g(a) \sqsubseteq \bigsqcap Q$ where $Q = \{f(b) \mid f \in F \wedge b \in \chi_f \wedge b \sqsupseteq a\}$. Since we have that $(\dot{\bigsqcap} F)(a') = \bigsqcap Q'$ with $Q' = \{f(b) \mid f \in F \wedge b \in \chi_f \wedge b \sqsupseteq a'\}$ and since $a \sqsubseteq a'$ we have that $Q' \subseteq Q$ and therefore, it holds that $\bigsqcap Q' \sqsupseteq \bigsqcap Q$. It follows that $g(a) \sqsubseteq (\dot{\bigsqcap} F)(a')$, which completes the proof that $g(a) \,\dot{\sqsubseteq} \dot{\bigsqcap} F$, therefore $\dot{\bigsqcap} F$ is a greatest lower bound.

The argument for $\dot{\bigsqcup} F$ is dual to the one above. □

In $\mathbb{D}(A,\Lambda)$, the partition χ is not fixed. Otherwise it is similar to the abstract partition lattice A^χ for a given partition χ. The order on $\mathbb{D}(A,\Lambda)$ captures the informal idea that using a finer precision leads to more precise case-based analyses. Since Λ is a lattice, this means that the best abstract representation of a concrete element must be a map from the most precise partition $\bot_\Lambda$ to elements of A. This is reflected in the definition of the abstraction function below.

Proposition 4.2.3. *The lattice* $(\mathbb{D}(A,\Lambda), \dot{\sqsubseteq})$ *overapproximates* $(C, \subseteq)$ *w.r.t. the following Galois connection.*

$$(C, \subseteq) \underset{\alpha_{\mathbb{D}}}{\overset{\gamma_{\mathbb{D}}}{\leftrightarrows}} (\mathbb{D}(A,\Lambda), \dot{\sqsubseteq})$$

$$\alpha_{\mathbb{D}}(c) \mathrel{\hat{=}} \{a \mapsto \alpha(\gamma(a) \cap c) \mid a \in \bot_\Lambda\}$$

$$\gamma_{\mathbb{D}}(f) \mathrel{\hat{=}} \bigcup_{a \in \chi_f} \gamma(f(a) \sqcap a)$$

Proof. We first show monotonicity of the two functions. Consider $c, c' \in C$ such that $c \subseteq c'$, then the functions $\alpha_{\mathbb{D}}(c)$ and $\alpha_{\mathbb{D}}(c')$ both have the same domain $\bot_\Lambda$. In order to show that $\alpha_{\mathbb{D}}(c) \,\dot{\sqsubseteq}\, \alpha_{\mathbb{D}}(c')$ it is therefore sufficient to show that for all $a, a' \in A$ with $a \sqsubseteq a'$ it holds that $\alpha_{\mathbb{D}}(c)(a) \sqsubseteq \alpha_{\mathbb{D}}(c')(a')$. We apply the definition and rewrite this inequality as $\alpha(\gamma(a) \cap c) \sqsubseteq \alpha(\gamma(a') \cap c')$, which follows since α, γ and $\cap$ are all monotone and since $a \sqsubseteq a'$ and $c \subseteq c'$.

Now consider $f : \chi_f \rightarrow A, g : \chi_g \rightarrow A \in C$ such that $f \,\dot{\sqsubseteq}\, g$. Then $\gamma_{\mathbb{D}}(f) = \bigcup Q_f$ where $Q_f = \{\gamma(f(a) \sqcap a) \mid a \in \chi_f\}$, and $\gamma_{\mathbb{D}}(g) = \bigcup Q_g$ where $Q_g = \{\gamma(g(a) \sqcap a) \mid a \in \chi_g\}$. Now consider an element $\gamma(f(a) \sqcap a) \in Q$. It is easy to see that since χ_f and χ_g are partitions with $\chi_f \preceq \chi_g$. There then exists $a' \in \chi_g$ with $a' \sqsupseteq a$. From $f \,\dot{\sqsubseteq}\, g$, we then get that $g(a') \sqsupseteq f(a)$ and therefore also $\gamma(g(a') \sqcap a') \sqsupseteq \gamma(f(a) \sqcap a)$. The element $\gamma(g(a') \sqcap a'$ is in the set Q_g. We have shown that for any element $\gamma(f(a) \sqcap a) \in Q$ there is a larger element $\gamma(g(a') \sqcap a') \in Q'$. Therefore, $\bigcup Q \subseteq \bigcup Q'$, or equivalently, $\gamma_{\mathbb{D}}(f) \subseteq \gamma_{\mathbb{D}}(g)$, which completes the proof that $\gamma_{\mathbb{D}}$ is monotone.

We now show that $\gamma_{\mathbb{D}} \circ \alpha_{\mathbb{D}}$ is inflationary.

$$\gamma_{\mathbb{D}} \circ \alpha_{\mathbb{D}}(c) = \bigcup_{a \in \bot_\Lambda} \gamma(\alpha(\gamma(a) \cap c) \sqcap a)$$

$$
\begin{aligned}
&= \bigcup_{a \in \bot_\Lambda} \gamma(\alpha(\gamma(a) \cap c)) \cap \gamma(a) \\
&\supseteq \bigcup_{a \in \bot_\Lambda} (\gamma(a) \cap c) \cap \gamma(a) \\
&= c \cap \bigcup_{a \in \bot_\Lambda} \gamma(a) = c \cap \top = c \\
&\implies \gamma_{\mathbb{D}} \circ \alpha_{\mathbb{D}}(c) \supseteq c
\end{aligned}
$$

We now show that $\alpha_{\mathbb{D}} \circ \gamma_{\mathbb{D}}$ is deflationary. Consider a function $f : \chi_f \to A$ in $\mathbb{D}(A, \Lambda)$. Then $\alpha_{\mathbb{D}} \circ \gamma_{\mathbb{D}}(f)$ has the signature $\bot_\Lambda \to A$, and trivially, $\bot_\Lambda \preceq \chi_f$. It remains to show that, whenever for $a \in \chi_\bot$ and for $b \in \chi_f$ it holds that $a \sqsubseteq b$, then $\alpha_{\mathbb{D}} \circ \gamma_{\mathbb{D}}(f)(a) \sqsubseteq f(b)$. Assume for a and b as above that $a \sqsubseteq b$.

$$
\begin{aligned}
\alpha_{\mathbb{D}} \circ \gamma_{\mathbb{D}}(f)(a) &= \alpha(\gamma(a) \cap \bigcup_{a' \in \chi_f} \gamma(f(a') \sqcap a')) \\
&= \alpha(\gamma(a) \cap (\gamma(f(b) \sqcap b) \cup \bigcup_{a' \in \chi_f \setminus \{b\}} \gamma(f(a') \sqcap a'))) \\
&\sqsubseteq \alpha(\gamma(a) \cap (\gamma(f(b) \sqcap b) \cup \bigcup_{a' \in \chi_f \setminus \{b\}} \gamma(f(a') \sqcap a'))) \\
&= \alpha(\gamma(a) \cap \gamma(f(b) \sqcap b)) \sqcup \alpha(\gamma(a) \cap \bigcup_{a' \in \chi_f \setminus \{b\}} \gamma(f(a') \sqcap a'))) \\
&= \alpha(\gamma(a \sqcap f(b) \sqcap b)) \sqcup \alpha(\bigcup_{a' \in \chi_f \setminus \{b\}} \gamma(a \sqcap f(a') \sqcap a'))
\end{aligned}
$$

Since χ_f is an abstract partition with $b \sqsupseteq a$, it holds for all $a' \in \chi_f \setminus \{b\}$ that $\gamma(a \sqcap a') = \bot$, therefore we can rewrite the right hand side as follows.

$$= \alpha(\gamma(a \sqcap f(b) \sqcap b)) \sqcup \alpha(\bot)$$

Since $\alpha(\bot)$ is smaller than $\alpha(\gamma(\ldots))$, this can be further simplified to the following.

$$
\begin{aligned}
&= \alpha(\gamma(a \sqcap f(b) \sqcap b)) \\
&= \alpha(\gamma(a \sqcap f(b))) \\
&\sqsubseteq a \sqcap f(b) \\
&\sqsubseteq f(b) \\
&\implies \alpha_{\mathbb{D}} \circ \gamma_{\mathbb{D}}(f)(a) \sqsubseteq f(b)
\end{aligned}
$$

We have shown that $\alpha_{\mathbb{D}}$ and $\gamma_{\mathbb{D}}$ are monotone functions such that $\gamma_{\mathbb{D}} \circ \alpha_{\mathbb{D}}$ is inflationary and $\alpha_{\mathbb{D}} \circ \gamma_{\mathbb{D}}$ is deflationary. Therefore, $(\alpha_{\mathbb{D}}, \gamma_{\mathbb{D}})$ is a Galois connection. □

The DPLL algorithm as depicted in Algorithm 4 can be viewed as implementing a fixed point iteration procedure that operates on $\mathbb{D}(PAsgs, \Lambda_{PAsgs})$, where Λ_{PAsgs} is the lattice of abstract partitions over partial assignments depicted in Figure 4.4.

Example 4.2.3. *Consider the run of* DPLL *that produces the decision tree depicted in Figure 4.3. Note that run of the algorithm partitioned the search space into the following set of cases using decisions.*

$$\chi = \{\langle q{:}\mathsf{f}\rangle, \langle q{:}\mathsf{t}, s{:}\mathsf{f}\rangle, \langle q{:}\mathsf{t}, s{:}\mathsf{t}\rangle\}$$

We can equally view the algorithm to have computed the element of the lattice $\mathbb{D}(\rho, \Lambda)$ depicted below.

$$\{\langle q{:}\mathsf{f}\rangle \mapsto \bot, \langle q{:}\mathsf{t}, s{:}\mathsf{f}\rangle \mapsto \bot, \langle q{:}\mathsf{t}, s{:}\mathsf{t}\rangle \mapsto \langle p{:}\mathsf{t}, q{:}\mathsf{t}, s{:}\mathsf{t}, r{:}\mathsf{t}\}$$

The unit rule and BCP transformers can be trivially lifted to elements of the dynamic partitioning domain by applying them pointwise. The more general insight is as follows.

Proposition 4.2.4. *Let A be an abstract domain and let $h : A \to A$ be a sound abstract transformer. Then the following is a sound abstract transformer.*

$$h : \mathbb{D}(A, \Lambda) \to \mathbb{D}(A, \Lambda) \qquad h(f : \chi \to A) \mathrel{\hat{=}} \{a \mapsto h(a) \mid a \in \chi\}$$

Decisions are used to refine the representation of an object without changing the semantics of the object. In abstract interpretation terminology, it is therefore an instance of semantic reduction. We model case splits as transformers on the lattice of partitions.

Definition 4.2.5 (Semantic Splitting Function)**.** Consider an abstract domain A with Galois connection $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$. A *semantic splitting function* is a function $split : A \to \mathcal{P}(A)$ such that the following holds for all $a \in A$.

$$(i)\ \forall b \in split(a).\ b \sqsubseteq a \qquad (ii)\ \bigcup_{b \in split(a)} \gamma(b) = \gamma(a)$$

Example 4.2.4. *Consider three totally ordered propositions $p < q < r$. In the following, we define a splitting function for partial assignments, which applies splits in the order $<$.*

$$split(\rho) \mathrel{\hat{=}} \begin{cases} \{\rho \sqcap \langle p{:}\mathsf{t}\rangle, \rho \sqcap \langle p{:}\mathsf{f}\rangle\} & \textit{if } \rho(p) = \top \\ \{\rho \sqcap \langle q{:}\mathsf{t}\rangle, \rho \sqcap \langle q{:}\mathsf{f}\rangle\} & \textit{if } \rho(p) \neq \top \textit{ and } \rho(q) = \top \\ \{\rho \sqcap \langle r{:}\mathsf{t}\rangle, \rho \sqcap \langle r{:}\mathsf{f}\rangle\} & \textit{if } \rho(p) \neq \top,\ \rho(q) \neq \top \textit{ and } \rho(r) = \top \\ \{\rho\} & \textit{otherwise} \end{cases}$$

From a splitting function, we may derive a decision operator which refines an element of the dynamic partitioning domain.

Definition 4.2.6 (Decision Operator)**.** Let $split : A \to A$ be a semantic splitting function over an abstract domain A with dynamic partitioning domain $\mathbb{D}(A, \Lambda)$, where Λ is the complete lattice of partitions of A.

Then a *decision function* of *split* is an order-preserving function $decide : \mathbb{D}(A, \Lambda) \to \mathbb{D}(A, \Lambda)$ such that for any $f : \chi_f \to A \in \mathbb{D}(A, \Lambda)$ there is a partition element $a \in \chi$ such that the following conditions (i) to (iii) below hold for $decide(f) : \chi_{decide(f)} \to A$.

$$\text{(i)} \qquad \chi_{decide(f)} = \chi_f \setminus \{a\} \cup split(a)$$

$$\text{(ii)} \qquad decide(f)(a') = \begin{cases} f(a') & \text{if } a' \in \chi_f \setminus \{a\} \\ f(a) \sqcap a' & \text{if } a' \in split(a) \end{cases}$$

$$\text{(iii)} \qquad split(a) \cap (\chi_f \setminus \{a\}) = \emptyset$$

Informally, a decision function takes a case partition together with analysis results, and refines a block of the partitioning by replacing it with the result of applying a splitting function to that block. Note that condition (iii) explicitly forbids the refining elements contained in $split(a)$ to be already elements of the set $\chi_f \setminus \{a\}$. This condition is not strictly necessary for what follows, but it is reasonable, and simplifies proofs.

Decision making corresponds to partition refinement. One way to view partition refinement is as a kind of abstraction refinement, which replaces one abstract partition lattice with a second, more precise one. We give a different but related perspective here. In the context of the dynamic partitioning lattice, partition refinement via decisions is an instance of semantic reduction, since it constitutes a refinement of the abstract representation of a concrete element, without changing its semantics.

Proposition 4.2.5. *A decision function $decide : \mathbb{D}(A, \Lambda)$ is a sound semantic reduction operator.*

Proof. By definition, *decide* is order preserving. It is easy to see that it is also deflationary. It remains to show that $\gamma_{\mathbb{D}} \circ \mathit{decide} = \gamma_{\mathbb{D}}$. In the following, we have that $f : \chi_f \to A$, and $\mathit{decide}(f) : \chi_{\mathit{decide}(f)} \to A$, and let $a \in \chi_f$ be the element such that the conditions in Definition 4.2.6 are satisfied.

$$\begin{aligned}
\gamma_{\mathbb{D}} \circ \mathit{decide}(f) = & \bigcup_{a' \in \chi_{\mathit{decide}(f)}} \gamma(\mathit{decide}f(a') \sqcap a') \\
= & \bigcup_{a' \in \chi_f \setminus \{a\}} \gamma(f(a') \sqcap a') \cup \bigcup_{a' \in \mathit{split}(a)} \gamma(f(a) \sqcap a') \\
= & \bigcup_{a' \in \chi_f \setminus \{a\}} \gamma(f(a') \sqcap a') \cup (\gamma(f(a)) \cap \bigcup_{a' \in \mathit{split}(a)} \gamma(a')) \\
= & \bigcup_{a' \in \chi_f \setminus \{a\}} \gamma(f(a') \sqcap a') \cup (\gamma(f(a)) \cap \gamma(a)) \\
= & \bigcup_{a' \in \chi_f} \gamma(f(a') \sqcap a') = \gamma_{\mathbb{D}}(f) \\
\Longrightarrow & \ \gamma_{\mathbb{D}} \circ \mathit{decide}(f) = \gamma_{\mathbb{D}}(f)
\end{aligned}$$

□

We may now understand a run of DPLL as a greatest fixed point iteration over the abstract domain $\mathbb{D}(\mathit{PAsgs}, \Lambda_{\mathit{PAsgs}})$, as shown in Figure 4.5. The abstract fixed point that is computed is as follows, where $\mathit{bcp}_{\mathcal{D}}$ is a transformer that replaces the "current" partition mapping $a \mapsto b$ with $a \mapsto \mathit{bcp}_{\varphi}(b)$.

$$\mathsf{gfp}\ \mathit{decide} \circ \mathit{bcp}_{\mathcal{D}}$$

When unraveled via iteration, the above fixed point is of the following form:

$$\mathit{decide}(\mathit{bcp}_{\mathcal{D}}(\ldots \mathit{decide}(\mathit{bcp}_{\mathcal{D}}(\top))))$$

The above unwinding corresponds to the intuitive understanding of classic DPLL as alternating decision and propagation steps.

4.3 Stålmarck's Saturation Procedure

Stålmarck's proof procedure for propositional logic (STALMARCK) [161] is a satisfiability procedure that is closely related to DPLL. In this section, we show that the *dilemma rule* and the *k-saturation procedure*, which are both components of STALMARCK are techniques for refining approximations of completely additive, deflationary functions. In practical terms, this means that we can apply the algorithm to solve instances of the bottom-everywhere problem.

We give an account of STALMARCK that uses the same data structures and reasoning rules as DPLL. Our account is not entirely faithful to the original algorithm, since STALMARCK uses its own unique data structures and reasoning rules. These are closely related, yet distinct from those employed in the DPLL algorithm. The variant of the procedure we present is sufficient for our purposes and allows us to work in the formal framework established in this section. An independent discussion of STALMARCK in the context of abstract interpretation that gives a more faithful account of the original algorithm can be found in [165, 166].

We first describe the dilemma rule and k-saturation and then go on to interpret them as refinement transformers.

$$\begin{aligned}
F_0 = &\quad \{\top \mapsto \top\} \\
\sqsupseteq F_1 = &\quad \mathit{bcp}_{\mathcal{D}}(F_0) = \quad \{\top \mapsto \langle p{:}\mathsf{t}\rangle\} \\
\sqsupseteq F_2 = &\quad \mathit{decide}(F_1) = \quad \{\langle q{:}\mathsf{f}\rangle \mapsto \langle p{:}\mathsf{t}, q{:}\mathsf{f}\rangle, \langle q{:}\mathsf{t}\rangle \mapsto \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle\} \\
\sqsupseteq F_3 = &\quad \mathit{bcp}_{\mathcal{D}}(F_2) = \quad \{\langle q{:}\mathsf{f}\rangle \mapsto \bot, \langle q{:}\mathsf{t}\rangle \mapsto \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle\} \\
\sqsupseteq F_4 = &\quad \mathit{decide}(F_3) = \quad \{\langle q{:}\mathsf{f}\rangle \mapsto \bot, \langle q{:}\mathsf{t}, s{:}\mathsf{f}\rangle \mapsto \langle p{:}\mathsf{t}, q{:}\mathsf{t}, s{:}\mathsf{f}\rangle, \langle q{:}\mathsf{t}, s{:}\mathsf{t}\rangle \mapsto \langle p{:}\mathsf{t}, q{:}\mathsf{t}, s{:}\mathsf{t}\rangle\} \\
\sqsupseteq F_5 = &\quad \mathit{bcp}_{\mathcal{D}}(F_4) = \quad \{\langle q{:}\mathsf{f}\rangle \mapsto \bot, \langle q{:}\mathsf{t}, s{:}\mathsf{f}\rangle \mapsto \bot, \langle q{:}\mathsf{t}, s{:}\mathsf{t}\rangle \mapsto \langle p{:}\mathsf{t}, q{:}\mathsf{t}, s{:}\mathsf{t}\rangle\} \\
\sqsupseteq F_6 = &\quad \mathit{decide}(F_5) = \quad \{\langle q{:}\mathsf{f}\rangle \mapsto \bot, \langle q{:}\mathsf{t}, s{:}\mathsf{f}\rangle \mapsto \bot, \langle q{:}\mathsf{t}, s{:}\mathsf{t}\rangle \mapsto \langle p{:}\mathsf{t}, q{:}\mathsf{t}, r{:}\mathsf{t}, s{:}\mathsf{t}\rangle\}
\end{aligned}$$

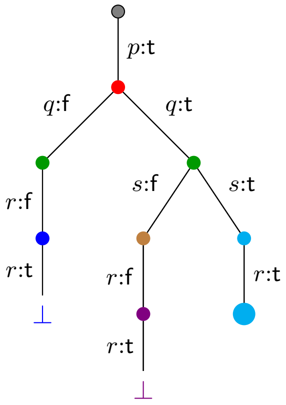


Figure 4.5: DPLL as fixed point iteration.

4.3.1 An Overview of Stålmarck's Saturation Procedure

The dilemma rule is a special "branch-and-merge" rule [161] which combines information derived on different branches of the search tree.

Example 4.3.1. *Consider the formula φ below.*

$$\varphi = (p \vee q) \wedge (\neg p \vee q)$$

Note that BCP *is insufficient to deduce any information about the above formula. The dilemma rule first introduces a branch to refine the precision of an analysis and then merges the results of each branch to yield a valid deduction result. The process is illustrated below.*

$$\begin{tikzcd}
& bcp_{\varphi}(\top) = \top \arrow[dl] \arrow[dr] & \\
bcp_{\varphi}(\langle p{:}\mathsf{t}\rangle) = \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle \arrow[dr] & & bcp_{\varphi}(\langle p{:}\mathsf{t}\rangle) = \langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle \arrow[dl] \\
& \langle q{:}\mathsf{t}\rangle &
\end{tikzcd}$$

Above, the base analysis is refined by performing a case split on the value of p. We find that no matter what value p takes, q evaluates to true. Therefore, the partial assignment $\langle q{:}\mathsf{t}\rangle$ is a valid deduction result.

In contrast to DPLL, which amounts to a classical branch-and-bound search algorithm, branches are collapsed and their results are combined. Information is merged by intersecting the partial assignments obtained on each branch, that is, keeping only those truth assignments that are common to both branches. STALMARCK may be parametrized by the branching depth that is employed before branches are collapsed and thus yields a family of proof procedures of increasing strength. A procedure that never branches is a 0-*saturation* procedure and one that branches k-times is a k-*saturation* procedure.

We give a version of the k-saturation algorithm in Algorithm 5. Note that we deviate in our presentation from the standard algorithm, since we use partial assignments and BCP for deduction. The algorithm uses a function titled `merge` to combine information. It is easy to see that `merge(`ρ_{t}`, `ρ_{f}`)` returns the join $\rho_{\mathsf{t}} \sqcup \rho_{\mathsf{f}}$ in the lattice of partial assignments.

4.3.2 Abstract Saturation

Algorithm 6 generalizes STALMARCK to lattices. The saturation procedure performs a split on each propositional variable in turn. The abstract version is parametrized by a set of splitting functions. It is possible to specialize the abstract algorithm to obtain the propositional procedure for a formula φ as follows. (i) As the lattice A, choose the lattice of partial assignments, (ii) in place of the transformer g, use the unit transformer $unit_{\varphi}$ and (iii) define the set of splitting functions $Split$ as $\{split_{<} \mid\ < \text{is a total order on } P\}$, where $split_{<}$ is defined as follows.

$$split_{<}(\rho) \mathrel{\hat{=}} \begin{cases} \{\rho\} & \text{if } \rho \text{ is an atom of } PAsgs \\ \{\rho \sqcap \langle p{:}\mathsf{t}\rangle, \rho \sqcap \langle p{:}\mathsf{f}\rangle\} & \text{else, s.t. } p \text{ is } {<}\text{-minimal s.t. } \rho(p) = \top \end{cases}$$

We now define transformers that capture the behavior of the abstract saturation procedure. We first define a transformer that models the dilemma rule. The transformer is parametrized by an abstract transformer g and a splitting function $split$. When applied to an element a, the transformer applies g to each element of the set $split(a)$ and then joins the results.

Algorithm 5: k-saturation over partial assignments.

```
in      : $k \in \mathbb{N}_0$ – constant
          $\varphi$ – propositional CNF formula $\varphi$
          $\rho$ – partial assignment
out     : partial assignment
saturate($k$, $\varphi$, $\rho$)
    if $k = 0$ then return $bcp_\varphi(\rho)$;
    repeat
        $\rho' \leftarrow \rho$;
        foreach proposition $p \in P$ do
            $\rho_\mathsf{t} \leftarrow$ saturate($k-1$,$\varphi$,$\rho \sqcap \langle p$:t$\rangle$);
            $\rho_\mathsf{f} \leftarrow$ saturate($k-1$,$\varphi$,$\rho \sqcap \langle p$:f$\rangle$);
            $\rho \leftarrow \rho \sqcap$ merge($\rho_\mathsf{t}$, $\rho_\mathsf{f}$)
        end
    until $\rho = \rho'$;
    return $\rho$;

merge($\rho_\mathsf{t}$, $\rho_\mathsf{f}$)
    if $\rho_\mathsf{t} = \bot$ then return $\rho_\mathsf{f}$;
    if $\rho_\mathsf{f} = \bot$ then return $\rho_\mathsf{t}$;
    return $\{p \mapsto v \mid p \in P, v \in \mathbb{B}.\ \rho_\mathsf{t}(p) = \rho_\mathsf{f}(p) = v\}$
```

Algorithm 6: k-saturation over lattice A.

```
in      : $k \in \mathbb{N}_0$ – constant
          $g : A \to A$ – abstract transformer
          $a \in A$ – abstract element
          $split$ – set of splitting functions
out     : abstract element of $A$
asaturate($k$, $g$, $a$, $Split$)
    if $k == 0$ then
        return $\mathsf{pgfp}(g)(a)$;
    end
    repeat
        $a' \leftarrow a$;
        foreach $split \in Split$ do
            $Q \leftarrow \{$asaturate($k-1$, $g$, $d$, $Split$) $\mid d \in split(a)\}$;
            $a \sqcap \bigsqcup Q$;
        end
    until $a = a'$;
    return $a$;
```

Definition 4.3.1 (Dilemma Transformer)**.** Let $(A, \sqsubseteq)$ be an overapproximation of a Boolean algebra $(C, \subseteq)$ w.r.t. (α, γ), let $g : A \to A$ be a transformer and let $split : A \to \mathcal{P}(A)$ be a semantic splitting function. Then the *dilemma transformer* of g and *split* is defined below.

$$dlm_{split}(g) : A \to A \qquad\qquad dlm_{split}(g)(a) \mathrel{\hat{=}} a \sqcap \bigsqcup_{d \in split(a)} g(d)$$

We now show that the dilemma transformer implements a technique that uses a splitting function to soundly refine an approximation of a completely additive, deflationary transformer. Since ded_φ is completely additive and deflationary, it follows that STALMARCK is a partition-based refinement strategy.

Proposition 4.3.1. *Let $f : C \to C$ be an additive, deflationary transformer and let $g : A \to A$ be a sound approximation of f w.r.t. (α, γ). For any splitting function split, $dlm_{split}(g)$ is more precise than g and soundly approximates f, i.e., the following two statements hold.*

$$\text{(i) } \gamma \circ dlm_{split}(g) \subseteq f \circ \gamma \qquad\qquad \text{(ii) } dlm_{split}(g) \sqsubseteq g$$

Proof. We first show (i). By the properties of a splitting function, it holds that $\bigcup_{d \in split(a)} \gamma(d) \supseteq \gamma(a)$. Therefore, by monotonicity $f(\bigcup_{d \in split(a)} \gamma(d)) \supseteq f \circ \gamma(a)$, and since f is completely additive, $\bigcup_{d \in split(a)} f \circ \gamma(d) \supseteq f \circ \gamma(a)$. Since g soundly approximates f, we then have that $\bigcup_{d \in split(a)} \gamma \circ g(d) \supseteq f \circ \gamma(a)$, and since the abstract join overapproximates the concrete, we get that $\gamma(\bigsqcup_{d \in split(a)} g(d)) \supseteq f \circ \gamma(a)$. Since f is deflationary, the left hand side can be rewritten as $\gamma(a \sqcap \bigsqcup_{d \in split(a)} g(d))$. We then have that $\gamma \circ dlm_{split}(g) \supseteq f \circ \gamma(a)$, which completes the first proof.

For (ii), recall that all d in $split(a)$ are less than a. By monotonicity, we therefore have for all such d that $g(d) \sqsubseteq g(a)$. It follows that $\bigsqcup_{d \in split(a)} g(d) \sqsubseteq g(a)$ and therefore $a \sqcap \bigsqcup_{d \in split(a)} g(d) \sqsubseteq g(a)$, which is equivalent to $dlm_{split}(g)(a) \sqsubseteq g(a)$. □

The next transformer is the saturation transformer, which is recursively defined in terms of greatest fixed points over the dilemma transformers. The saturation transformer is equivalent to the abstract saturation procedure (Algorithm 6).

Definition 4.3.2 (k-Saturation Transformer)**.** Let $(A, \sqsubseteq)$ be an overapproximation of a Boolean algebra $(C, \subseteq)$ w.r.t. (α, γ), $g : A \to A$ be a transformer and $Split \subseteq A \to \mathcal{P}(A)$ be a non-empty set of semantic splitting functions. Then for $k \in \mathbb{N}_0$, the *k-saturation transformer* of g and *Split* is recursively defined below.

$$saturate^k_{split}(g) : A \to A$$

$$saturate^k_{split}(g)(a) \mathrel{\hat{=}} \begin{cases} \mathsf{pgfp}(g)(a) & \text{if } k = 0 \\ \mathsf{pgfp}(\bigsqcap_{split \in Split} dlm_{split}(saturate^{k-1}_{Split}(g)))(a) & \text{if } k \neq 0 \end{cases}$$

The saturation transformer soundly approximates the same function as g. Furthermore, choosing higher values for k increases precision.

Proposition 4.3.2. *Let $f : C \to C$ be a completely additive, deflationary function and $g : A \to A$ be an overapproximation of f w.r.t. (α, γ). For any set of semantic splitting functions Split over A, the following two statements hold.*

(i) For all $k \in \mathbb{N}_0$, $saturate^k_{Split}(g)$ soundly approximates f.

(ii) The sequence of transformers $(saturate^i_{splitfun}(g))_{i \in \mathbb{N}_0}$ is ordered pointwise as follows.

$$\ldots \sqsubseteq saturate^2_{Split}(g) \sqsubseteq saturate^1_{Split}(g) \sqsubseteq saturate^0_{Split}(g) \sqsubseteq g$$

Proof. We show (i) by induction on k. For $k = 0$, we have that $\mathit{saturate}^k_{\mathit{Split}}(g) = \mathsf{pgfp}(g)$, which proves the base case. Since f, by Proposition 3.2.2, is a lower closure operator, and g soundly approximates f, $\mathsf{pgfp}(g)$ soundly approximates f. Now consider that, for all $1 \leq i \leq k - 1$, $\mathit{saturate}^i_{\mathit{Split}}(g)$ soundly approximates f. Then, by Proposition 4.3.1, for all *split* in *Split*, $d_{\mathit{split}} \mathrel{\hat{=}} \mathit{dlm}_{\mathit{split}}(g)\mathit{saturate}^{k-1}_{\mathit{Split}}(g)$ is a sound approximation of f. It follows that $\bigsqcap_{\mathit{split} \in \mathit{Split}} d_{\mathit{split}}$ soundly approximates f. Since f is a lower closure, this implies that $\mathsf{pgfp}(\bigsqcap_{\mathit{split} \in \mathit{Split}} d_{\mathit{split}}) = \mathit{saturate}^k_{\mathit{Split}}(g)$ soundly approximates f.

For (ii), the statement $\mathit{saturate}^0_{\mathit{Split}}(g) \sqsubseteq g$ follows from $\mathit{saturate}^0_{\mathit{Split}}(g) = \mathsf{pgfp}(g)$, since $\mathsf{pgfp}(g)(a) = \mathsf{gfp}\, X.\ a \sqcap g(X)$. For the other inequalities, we show that for any transformer $h : A \to A$, it holds that $\mathsf{pgfp}(\bigsqcap_{\mathit{split} \in \mathit{Split}} \mathit{dlm}_{\mathit{split}}(h))$ is less than h. All inequalities of the form $\mathit{saturate}^{i+1}_{\mathit{Split}}(g) \sqsubseteq \mathit{saturate}^{i+1}_{\mathit{Split}}(g)$ then follow immediately from the definition of $\mathit{saturate}^{i+1}_{\mathit{Split}}(g)$.

We now show that $\mathsf{pgfp}(\bigsqcap_{\mathit{split} \in \mathit{Split}} \mathit{dlm}_{\mathit{split}}(h))$ is less than h. From Proposition 4.3.1, we have that $\mathit{dlm}_{\mathit{split}}(h)$ is less than h, for all $\mathit{split} \in \mathit{Split}$. Then from basic lattice theory, $\bigsqcap_{\mathit{split} \in \mathit{Split}} \mathit{dlm}_{\mathit{split}}(h)$ is less than h. From monotonicity, it is then easy to see that $\mathsf{pgfp}(\bigsqcap_{\mathit{split} \in \mathit{Split}} \mathit{dlm}_{\mathit{split}}(h))$ is less than h.

Since for all $k \geq 0$, it holds that $\mathit{saturate}^{k+1}_{\mathit{Split}}(g) = \mathsf{pgfp}(\bigsqcap_{\mathit{split} \in \mathit{Split}} \mathit{dlm}_{\mathit{split}}(h))$ where $h = \mathit{saturate}^k_{\mathit{Split}}(g)$, it follows that $\mathit{saturate}^{k+1}_{\mathit{Split}}(g) \sqsubseteq \mathit{saturate}^k_{\mathit{Split}}(g)$. □

The saturation transformer $\mathit{saturate}^k_{\mathit{Split}}(g)$ models the abstract k-saturation procedure and therefore also k-saturation. Elements returned by a call to `asaturate(`*k, g, a, Split*`)` coincide with the result of applying the transformer $\mathit{saturate}^k_{\mathit{Split}}(g)$ to the element a. Applying abstract saturation is similar to the *focus* operator that is employed in shape analysis [155].

The results of this section show that Stålmarck's k-saturation procedure is a refinement technique for approximations of additive, deflationary transformers f. STALMARCK can therefore be used to decide other instance of the bottom-everywhere problem such as program safety.

4.4 CDCL

This section relates the Conflict-Driven Clause Learning (CDCL) algorithm for propositional satisfiability to abstract interpretation. CDCL can be considered the most important algorithm of modern satisfiability research and will be a focus of the chapters to come. Chapter 5 presents ACDCL, a novel, lattice-theoretic generalization of CDCL. Chapters 6 and 7 discuss instantiations of ACDCL to solve the satisfiability problem for the first-order theory of floating-point arithmetic and to decide program safety, respectively. This section relates CDCL to abstract interpretation using an abstract, high-level perspective. Section 4.4.1 gives an overview of the algorithm. CDCL can be conceptualized as the integration of two distinct subprocedures: *model search* and *conflict analysis*. Section 4.4.2 and Section 4.4.3 respectively relate these substeps to computing fixed points using acceleration operators.

4.4.1 An Overview of CDCL

Historically, CDCL was developed based on the DPLL algorithm. Because of this, CDCL solvers are sometimes referred to as modern implementations or extensions of DPLL. While they share some similarities, the two algorithms explore the space of propositional assignments in different ways.

DPLL and CDCL both explore the search space by refining a partial assignment. If DPLL encounters a region of the search space that contains no models, the algorithm systematically backtracks and explores a different region. CDCL, also explores the search space by refining a partial assignment. When it encountered a region of the search space that contains no models, it attempts to find a larger region with the same property and uses the result to modify the formula in a step called *learning*. CDCL can be understood as the interplay of two sub-procedures: *model search*, which searches for a model by refining a partial assignment, and *conflict analysis*, which searches for a conflict element by generalizing a partial assignment.

We illustrate CDCL with an example using the following formula φ.

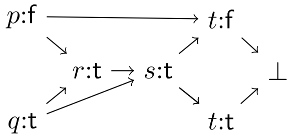
$$\varphi = (p \vee \neg q \vee r) \wedge (\neg r \vee \neg q \vee s) \wedge (\neg s \vee t) \wedge (p \vee \neg s \vee \neg t)$$

CDCL first executes the model search phase, which behaves identically to classic DPLL. Starting from the empty partial assignment $\top$, BCP is applied next.

$$bcp_{\varphi}(\top) = \top$$

Since BCP is unable to obtain any information about the instance, a decision is made, and f is assigned to p. This leads to no new information, so another decision assigns t to q. BCP now finds a conflict. The sequence of deductions is depicted below in a structure called the *implication graph*.

$$bcp_{\varphi}(\langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle) = \bot$$



Since p is false and q is true it follows via the unit rule that r is true. From q and r the value of s can be derived. Finally, a conflict is obtained since the variable t is assigned to both true and false. At this point, DPLL would backtrack and systematically explore the case where q is false. CDCL instead enters conflict analysis, which aims to generalize the partial assignment. From the graph above, we can see that when p is false and s is true, a conflict can be derived. We call elements from which a conflict can be derived *conflict reasons*. An example is the element $\langle p{:}\mathsf{f}, s{:}\mathsf{t}\rangle$, but it is not the only choice; the element $\langle q{:}\mathsf{f}, t{:}\mathsf{f}\rangle$ is also sufficient to produce a conflict. CDCL uses a heuristic which extracts one such reason from the graph.

Finding conflict reasons is an instance of abductive reasoning since it aims to find elements that are sufficient to derive a conflict.

We assume that CDCL picks the element $\langle p{:}\mathsf{f}, s{:}\mathsf{t}\rangle$ as a reason. This reason is negated to obtain a lemma in the form of the clause $C = (p \vee \neg s)$. A *learning* step adds the learned clause to the formula and we obtain a new formula $\varphi' = \varphi \wedge C$, which has the same models as φ. After this refinement, the algorithm backtracks and all assignments are undone except for the element $\langle p{:}\mathsf{f}\rangle$. Since we have a new clause $p \vee \neg s$ we can use the unit rule to derive the element $\langle s{:}\mathsf{f}\rangle$, which drives model search into a new area of the search space.

A schematic of the CDCL architecture is depicted in Figure 4.6. Model search proceeds by alternating BCP and decisions. If a model is found, the algorithm returns SAT. Otherwise, if a conflict occurs, that conflict is communicated to the conflict analysis engine. Conflict analysis attempts to generalize the conflict by identifying a more general element that is also sufficient for a conflict. Since a single conflict may have multiple, distinct reasons, one candidate is heuristically picked (in typical solvers, reasons are picked via a static hard-coded heuristic, as opposed to decisions, which are typically computed via dynamic heuristics).

4.4.2 Model Search

Model search iterates BCP and decisions until either a conflict is identified or a satisfying assignment is found. We now discuss two aspects of model search in the context of abstract interpretation. We have already shown in Section 4.1 that BCP finds a greatest fixed point

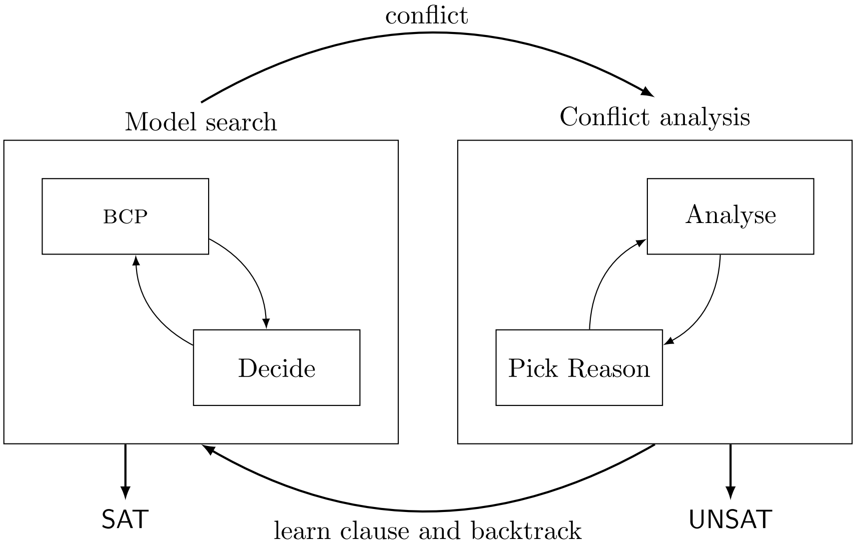


Figure 4.6: CDCL architecture.

by computing a downwards iteration sequence using the transformer $unit_\varphi$. We now demonstrate that BCP with decisions generates a downwards iteration sequence with extrapolation. We then show that the satisfiability condition in CDCL is a γ-completeness check.

Decisions as Downward Extrapolation

A decision in CDCL refines a partial assignment by choosing an unassigned variable and setting it to some value. In contrast to decisions in DPLL, decisions in CDCL are not case splits. In the example of the previous section, a decision $\langle q{:}\mathsf{t}\rangle$ is made, but after backtracking, the alternative case $\langle q{:}\mathsf{f}\rangle$ is not explored. Instead, the search is driven to a new area of the search space by using the learned clause $p \vee \neg s$ to derive $\langle s{:}\mathsf{f}\rangle$. Model search heuristically probes the search space in order to find satisfying assignments or conflicts. This process generates a downwards iteration sequence in the lattice of partial assignments. An illustration is given in Figure 4.7.

Decisions may be viewed as heuristic acceleration of the fixed point computation and therefore correspond to downwards extrapolation. Recall that a binary downwards extrapolation is a function $\nabla_\downarrow : A \times A \to A$ such that for all $a, b \in A$, $a \nabla_\downarrow b \sqsubseteq a$ and $a \nabla_\downarrow b \sqsubseteq b$ hold. Further, recall that for a transformer $f : A \to A$, the downwards iteration sequence with extrapolation is given as follows.

$$F_0^{\nabla_\downarrow} = \top \qquad F_{i+1}^{\nabla_\downarrow} \mathrel{\hat{=}} F_i^{\nabla_\downarrow} \nabla_\downarrow f(F_i^{\nabla_\downarrow})$$

Decisions refine an iteration sequence if the deduction transformer fails to make a refinement. We formalize this behavior in terms of a binary, downwards extrapolation operator $\nabla_\downarrow^{\mathsf{d}} : PAsgs \times PAsgs \to PAsgs$, which maps a pair of elements (ρ, ρ') in the lattice to a lower bound. If the two elements are different, then $\rho \nabla_\downarrow^{\mathsf{d}} \rho'$ is the greatest lower bound, otherwise, it is some element that is strictly smaller than the greatest lower bound.

$$\rho \nabla_\downarrow^{\mathsf{d}} \rho' = \begin{cases} \rho \sqcap \rho' & \text{if } \rho \neq \rho' \text{ or } \rho \sqcap \rho' \text{ is a total assignment} \\ \rho \sqcap \langle p : v\rangle & \text{if } \rho = \rho', \text{ where } \rho(p) = \top \text{ and } v \in \{\mathsf{t}, \mathsf{f}\} \end{cases}$$

We may now see that the sequence in Figure 4.7 is a descending iteration sequence with downwards extrapolation. Recall that the unit rule transformer $unit_\varphi$ for a CNF formula φ

$$\varphi = p \wedge (\neg p \vee q \vee r) \wedge (\neg r \vee s \vee \neg t)$$

$$\begin{tikzcd}[row sep=large]
\top \arrow[d, "\text{BCP}"] \\
\langle p : \mathsf{t} \rangle \arrow[d, "\text{decision}"] \\
\langle p : \mathsf{t}, q : \mathsf{f} \rangle \arrow[d, "\text{BCP}"] \\
\langle p : \mathsf{t}, q : \mathsf{f}, r : \mathsf{t} \rangle \arrow[d, "\text{decision}"] \\
\langle p : \mathsf{t}, q : \mathsf{f}, r : \mathsf{t}, s : \mathsf{f} \rangle \arrow[d, "\text{BCP}"] \\
\langle p : \mathsf{t}, q : \mathsf{f}, r : \mathsf{t}, s : \mathsf{f}, t : \mathsf{f} \rangle
\end{tikzcd}$$

Figure 4.7: Descending iteration sequence with decisions.

is defined as $\bigsqcap_{C \in \varphi} \mathit{unit}_C$. We denote by F_i the ith element of the downwards extrapolation sequence generated using unit_φ and $\nabla^{\mathsf{d}}_{\downarrow}$. Note that this sequence is identical to the sequence in Figure 4.7.

$$\begin{array}{lclcl}
F_0 & = & \top & & \\
F_1 & = & \top \, \nabla^{\mathsf{d}}_{\downarrow} \langle p{:}\mathsf{t} \rangle & = & \langle p{:}\mathsf{t} \rangle \\
F_2 & = & F_1 \, \nabla^{\mathsf{d}}_{\downarrow} \, F_1 & = & \langle p{:}\mathsf{t}, q{:}\mathsf{f} \rangle \\
F_3 & = & F_2 \, \nabla^{\mathsf{d}}_{\downarrow} \langle p{:}\mathsf{t}, q{:}\mathsf{f}, r{:}\mathsf{t} \rangle & = & \langle p{:}\mathsf{t}, q{:}\mathsf{f}, r{:}\mathsf{t} \rangle \\
F_4 & = & F_3 \, \nabla^{\mathsf{d}}_{\downarrow} \, F_3 & = & \langle p{:}\mathsf{t}, q{:}\mathsf{f}, r{:}\mathsf{t}, s{:}\mathsf{f} \rangle \\
F_5 & = & F_4 \, \nabla^{\mathsf{d}}_{\downarrow} \langle p{:}\mathsf{t}, q{:}\mathsf{f}, r{:}\mathsf{t}, s{:}\mathsf{f}, t{:}\mathsf{f} \rangle & = & \langle p{:}\mathsf{t}, q{:}\mathsf{f}, r{:}\mathsf{t}, s{:}\mathsf{f}, t{:}\mathsf{f} \rangle
\end{array}$$

The above sequence demonstrates that model search computes a downwards iteration sequence with extrapolation. The use of extrapolation here is non-traditional since it does not yield a sound approximation of the greatest fixed point. Typically, upwards extrapolation is used to compute an overapproximation of a least fixed point. Dually, downwards extrapolation can be used to compute an underapproximation of a greatest fixed point. In contrast, we use downwards extrapolation unsoundly to accelerate an overapproximate greatest fixed point computation.

We now argue why, despite this non-traditional use, conceptualizing decisions as downwards extrapolation is warranted. When instantiating model search over richer lattices and logics, convergence of the downwards iteration may be an issue, therefore downwards widening may need to be applied to converge on a conflict or solution more quickly. Just as in traditional uses of widening, it may be useful in such cases to consider the history of the iteration sequence. For example, downwards extrapolation may heuristically only be applied if the iteration sequence makes slow progress.

Satisfiability and γ-Completeness

We now discuss the criterion SAT solvers use to determine when to stop model search. Informally, this is the case when a partial assignment has sufficient precision to allow a determination that the formula is satisfiable. We will now make this notion precise using abstract interpretation terminology.

It is common to end model search when the partial assignment has become total, that is, if no proposition is mapped to $\top$. A more general criterion for determining satisfiability is to check whether each clause in the formula has one literal that is conclusively satisfied by the partial assignment.

Example 4.4.1. *Consider the formula φ and the partial assignment ρ below.*

$$\varphi = (p \vee \neg q \vee r) \wedge (q \vee \neg p \vee \neg s) \qquad\qquad \rho = \langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle$$

No matter which values we assign to the remaining propositions r and s, the resulting partial assignment is a model, since the assignment $p{:}\mathsf{f}$ ensures that the second clause is satisfied, while $q{:}\mathsf{t}$ ensures the same for the first.

This satisfiability criterion is a check for γ-completeness of the bcp_φ transformer at a certain point in the partial assignments lattice. Recall that a transformer $of : A \rightarrow A$, which is a sound approximation of a transformer $f : A \rightarrow A$, is γ-complete at an element a if the equation $\gamma \circ of(a) = of \circ \gamma(a)$ is satisfied. If bcp_φ is γ-complete, the concrete deduction transformer would not remove any countermodels from $\gamma \circ bcp_\varphi(a)$, that is, $bcp_\varphi(a)$ precisely represents a set of models.

Proposition 4.4.1. *Let $\varphi = C_1 \wedge \ldots \wedge C_k$ be a* CNF *formula and let ρ be a partial assignment. If for every clause C_i, there is a literal $l \in C_i$ such that for some proposition p in the set of propositions P (i) $l = p$ and $\rho(p) = \mathsf{t}$ or (ii) $l = \neg p$ and $\rho(p) = \mathsf{f}$ holds, then bcp_φ is γ-complete at ρ.*

Proof. Consider an element $\rho \in \gamma \circ bcp_\varphi(\rho)$. Then ρ is a model of φ, since it satisfies at least one literal in every clause. Since bcp_φ is deflationary (and since γ is monotone), it also holds that $\rho \in \gamma(\rho)$, and because ρ is a model of φ, it follows that $\rho \in ded_\varphi \circ \gamma(\rho)$. Therefore, $ded_\varphi \circ \gamma(\rho) \supseteq \gamma \circ bcp_\varphi(\rho)$. The other direction (for $\subseteq$) follows from soundness. Therefore, bcp_φ is γ-complete at ρ. □

4.4.3 Conflict Analysis

The conflict analysis phase in a SAT solver identifies conflict reasons, i.e., elements from which a conflict can be derived. Formally, a *reason* for a partial assignment ρ is an element ρ' such that $ded_\varphi \circ \gamma(\rho') \subseteq \gamma(\rho)$, i.e., ρ can be derived from ρ' via sound overapproximation of ded_φ. By the defining property of the Galois connection, this is equivalent to $\gamma(\rho') \subseteq abd_\varphi \circ \gamma(\rho)$. Therefore, finding reasons corresponds to sound underapproximation of abd_φ. A *conflict reason* is a reason for the conflict element $\bot$.

The following example illustrates that conflict reasons underapproximate abd_φ.

Example 4.4.2. *Consider a formula $\varphi = p \vee q$ and a partial assignment $\rho = \langle q{:}\mathsf{t}\rangle$. In the concrete, the most general reason for ρ can be obtained as follows. We abbreviate truth assignments $\{p \mapsto \mathsf{f}, q \mapsto \mathsf{t}\}$ to* ft.

$$abd_\varphi \circ \gamma(\rho) = \gamma(\rho) \cup \{\sigma \mid \sigma \not\models \varphi\} = \{\mathsf{tt}, \mathsf{ft}, \mathsf{ff}\}$$

The following set R is the set of all reasons for ρ.

$$R \mathrel{\hat{=}} \{\langle p{:}\mathsf{f}\rangle, \langle q{:}\mathsf{t}\rangle, \langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle, \langle p{:}\mathsf{f}, q{:}\mathsf{f}\rangle, \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle, \bot\}$$

Note that every single element of R soundly underapproximates $abd_\varphi \circ \gamma(\rho) = \{\mathsf{tt}, \mathsf{ft}, \mathsf{ff}\}$.

As shown in the above example, there may not be a single, best reason for a given element. This is because finding reasons is underapproximative, whereas partial assignments are an overapproximating abstraction. In general, overapproximating abstractions do not allow best underapproximating representations.

Underapproximate Abduction in the Downset Completion

There are multiple, incomparable reasons that can be picked in the conflict analysis phase. Modern CDCL-based SAT solvers use heuristics that pick one of these reasons [172], but the simultaneous use of multiple conflict reasons has also been explored [172, 16]. It is therefore appropriate to consider conflict analysis as an operation on sets of partial assignments, even though in practice, only one assignment may be considered for reasons of efficiency.

Downset completion is an operation that enriches an abstraction with disjunction [45, 160]. Consider an abstraction $(A, \sqsubseteq, \sqcup, \sqcap)$ of a powerset lattice $\mathcal{P}(S)$ w.r.t. a Galois connection (α, γ). The abstraction A *is disjunctive* if $\gamma(a \sqcup b) = \gamma(a) \cup \gamma(b)$.

Recall that a subset Q of A is *downwards closed*, or simply a *downset*, if for every x in Q and y in A, $y \sqsubseteq x$ implies that y is in Q. The *downset completion* of A is the complete lattice $\mathcal{D}(A)$. The downset completion of A is an underapproximation of the concrete domain, as evidenced by the following Galois connection.

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

Consult [44] for proofs that the pairs of functions above form Galois connections and that the domains are disjunctive.

Sets are not an efficient representation for practical applications, yet in many cases, downsets can be represented effectively as antichains. This is the case when the underlying lattice satisfies ACC (see Section 2.2.3 for details). Otherwise, antichain representations may still be possible depending on the structure of the lattice and the Galois connection. A detailed discussion of these conditions is beyond the scope of this document.

Choosing Reasons as Upwards Interpolation

The downset completion of a domain defines an underapproximation of the concrete domain and hence supports best conflict reasons.

Example 4.4.3. *Consider the formula $\varphi = (\neg p \vee q) \wedge (p \vee \neg q)$ used in the previous example and the partial assignment $\rho = \langle p{:}\mathsf{t}, q{:}\mathsf{t}\rangle$. The best underapproximative abduction transformer on the lattice $\mathcal{D}(PAsgs)$ of partial assignments is $uabd_\varphi \mathrel{\hat{=}} \alpha_{\mathcal{D}(PAsgs)} \circ abd_\varphi \circ \gamma_{\mathcal{D}(PAsgs)}$.*

We may now find an optimal reason for ρ using the $uabd_\varphi$ transformer in the form of a set of reasons.

$$uabd_\varphi(\mathcal{D}(\{\rho\})) = \downarrow\{\langle p{:}\mathsf{t}\rangle, \langle q{:}\mathsf{t}\rangle\}$$

Downsets and antichains are potentially expensive representations, since they are set-based. For efficiency reasons, CDCL computes just one conflict reason. In the example above, we may decide by some heuristic measure that $\langle p{:}\mathsf{t}\rangle$ is the more important of the two reasons. In order for the CDCL algorithm to avoid encountering the same conflicts repeatedly, it is important that the chosen reason generalizes the original conflict. We define a choice function that has the required properties.

Definition 4.4.1 (Heuristic Choice Function). A partial function $choose : PAsgs \times \downarrow PAsgs \to PAsgs$ is a *heuristic choice function* if for all $\rho \in PAsgs$ and $R \subseteq PAsgs$ such that $\rho \in R$, the function *choose* is defined at (ρ, R) and the following holds.

$$choose(\rho, R) \in R \wedge choose(\rho, R) \sqsupseteq \rho$$

For any set of candidate reasons R of ρ, the element $choose(\rho, R)$ is a choice for a generalization of ρ from R. From a choice function *choose* we can derive a partial upwards interpolation operator $\Delta_\uparrow : \downarrow PAsgs \times \downarrow PAsgs \to \downarrow PAsgs$ defined as follows.

$$\downarrow\{\rho\} \,\Delta_\uparrow\, R \mathrel{\hat{=}} \downarrow\{choose(\rho, R)\}$$

Proposition 4.4.2. *The partial function $\Delta_\uparrow$ derived from choose is an upwards interpolation operator.*

Proof. We show that whenever $Q \, \Delta_\uparrow \, R$ is defined for sets $Q, R \in \downarrow PAsgs$ then $Q \subseteq R \implies Q \subseteq Q \, \Delta_\uparrow \, R \subseteq R$ holds. Assume that $Q \, \Delta_\uparrow \, R$ is defined. Then Q is of the form $\downarrow\{\rho\}$.

Now assume that $Q \subseteq R$. Then it holds that $\rho \in R$ and therefore also $choose(\rho, R) \in R$. It follows that $\downarrow\{\rho\} \subseteq \downarrow\{choose(\rho, R)\} \subseteq R$, since R is downwards closed, which may be rewritten as follows.

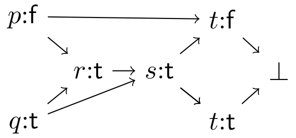
$$Q \subseteq Q \, \Delta_\uparrow \, R \subseteq R$$

□

Conflict analysis in a SAT solver first applies an underapproximate abduction transformer $uabd_\varphi : \mathcal{D}(PAsgs) \to \mathcal{D}(PAsgs)$ and then applies upwards interpolation to select one of them. These two steps may be iterated (for example, if different methods of abduction are available). Conflict analysis corresponds to computing an upwards iteration sequence with interpolation using an underapproximate abduction transformer.

To illustrate, consider again the implication graph from earlier.

$$\varphi = (p \vee \neg q \vee r) \wedge (\neg r \vee \neg q \vee s) \wedge (\neg s \vee t) \wedge (p \vee \neg s \vee \neg t)$$



To illustrate, we define an abduction transformer which applies the unit rule for a clause C in reverse. Given a set of partial assignments Q, it returns all partial assignments ρ, such that from ρ, some element of Q can be derived using the unit rule.

Definition 4.4.2 (Unit Abduction Transformer)**.** The *unit abduction transformers* $uabd_{unit_C} : \mathcal{D}(PAsgs) \to \mathcal{D}(PAsgs)$ for a clause C and a formula φ is defined as follows.

$$\begin{aligned} uabd_{\mathrm{C}}(R) &\mathrel{\hat{=}} R \cup \{\rho \mid \exists \rho' \in R.\ unit_C(\rho) \sqsubseteq \rho'\} \\ uabd_\varphi(R) &\mathrel{\hat{=}} \bigcup_{C \in \varphi} uabd_{\mathrm{C}} \end{aligned}$$

For example, for the clause $C = \neg s \vee t$ in the example above, we have that $uabd_{\mathrm{C}}(\downarrow\{\langle t{:}\mathsf{t}\rangle\})$ is equal to $\downarrow\{\langle s{:}\mathsf{t}\rangle, \langle t{:}\mathsf{t}\rangle\}$ since the value of t can be derived if s is true.

We may now see that conflict analysis computes an upwards iteration sequence with interpolation using an underapproximate abductive transformer.

Example 4.4.4. *We compute an upwards iteration sequence with interpolation using $uabd_\varphi$. We use an iteration scheme similar to chaotic iteration [44], which applies the individual clause transformers $unit_C$ sequentially instead of in parallel and we use upwards interpolation to select reasons. Both the choice of clause transformer as well as the interpolation operator are informed by the sequence of deductions recorded during model search.*

$$\begin{aligned} F_0 &\mathrel{\hat{=}} \downarrow\{\bot\} & \\ F_1' &\mathrel{\hat{=}} uabd_{\mathrm{p\vee\neg s\vee\neg t}}(\downarrow\{\bot\}) &&= \downarrow\{\langle p{:}\mathsf{f}, s{:}\mathsf{t}, t{:}\mathsf{t}\rangle\} \\ F_1 &\mathrel{\hat{=}} \bot \, \Delta_\uparrow \, F_1' &&= \downarrow\{\langle p{:}\mathsf{f}, s{:}\mathsf{t}, t{:}\mathsf{t}\rangle\} \\ F_2' &\mathrel{\hat{=}} uabd_{\mathrm{\neg s\vee t}}(F_1) &&= \downarrow\{\langle s{:}\mathsf{t}, t{:}\mathsf{f}\rangle, \langle p{:}\mathsf{f}, s{:}\mathsf{f}\rangle\} \\ F_2 &\mathrel{\hat{=}} F_1 \, \Delta_\uparrow \, F_2' &&= \downarrow\{\langle p{:}\mathsf{f}, s{:}\mathsf{f}\rangle\} \end{aligned}$$

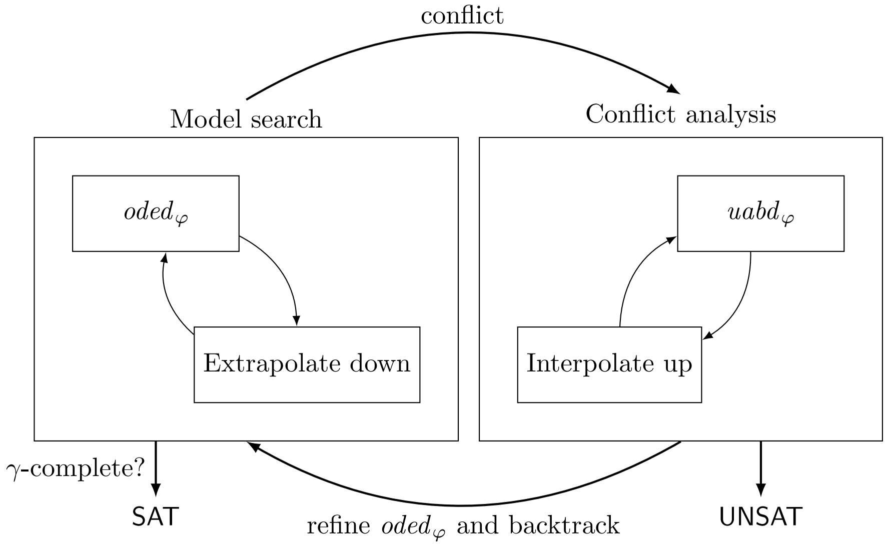


Figure 4.8: Abstract interpretation perspective of CDCL.

In practice, the interpolation and abduction step are often combined into a single computation. By modeling choice of reasons as interpolation, we make explicit that one parameter of conflict analysis is the heuristic which chooses these candidates, just as model search uses heuristic extrapolation to pick decision elements. A more thorough, algorithmic discussion of conflict analysis and the role of abduction and interpolation is given in Chapter 5.

The results of this section are summarized in Figure 4.8: Model search computes a downwards iteration sequence with extrapolation using an overapproximate deduction transformer. Conflict analysis computes an upwards iteration sequence with interpolation using an underapproximate abduction transformer. In the classic view of the algorithm, the formula φ is modified via an assignment $\varphi \leftarrow \varphi \wedge C$. In abstract interpretation terms, this corresponds to refining the overapproximate deduction transformer by assigning the transformer $oded_\varphi$ to $oded_\varphi \sqcap oded_C$. Thus, learning in a SAT solver is transformer refinement. A precise abstract formulation of learning is given in Chapter 5.

4.5 DPLL(T)

DPLL(T) [65, 143], a central algorithm in SMT research, aims to lift the power of the CDCL algorithm to SMT logics. DPLL(T) splits the deduction task of reasoning about a first-order formula between two separate reasoning engines: a propositional SAT solver and a first-order theory solver. The propositional solver is typically based on CDCL and enumerates assignments from atomic formulas to truth values. The theory solver checks whether the conjunction of theory atoms represented by the assignment has a model.

We illustrate DPLL(T) with the following example involving Boolean combinations of equality constraints.

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

A DPLL(T) solver first constructs a *Boolean skeleton* of φ, denoted 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 then be used to determine if the conjunction is satisfiable. If the conjunction defined by a satisfying assignment ρ to $\mathsf{BoolSkel}(\varphi)$ is not satisfiable, the solver can *learn* $\neg\rho$ 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.

This section shows that a DPLL(T) solver can be understood as an abstract interpreter. We will focus our discussion on the data structures and components of DPLL(T) rather than the overall algorithm. An exhaustive discussion of clause learning will be given in Chapter 5.

This section is structured as follows. Section 4.5.1 shows that reasoning about the Boolean skeleton of a formula is an abstract interpretation of its first-order semantics. Section 4.5.2 and Section 4.5.3 demonstrate that SAT solvers used within DPLL(T) are abstract interpreters. Section 4.5.4 demonstrates that the combination of propositional and theory solvers is an instance of constructing the Cartesian product, and that communication between the two solvers is used to approximate the reduced product.

4.5.1 Boolean Abstractions of First-Order Logic

This section shows that computing propositional solutions that satisfy the Boolean skeleton of a formula is an abstract interpretation of the formula's theory semantics.

In the following, fix an SMT logic $(\mathcal{F}, \models, \mathcal{S})$ where the set of formulas $\mathcal{F}$ is defined over a signature Σ. Fix φ to be a formula in $\mathcal{F}$ and P to be a fresh set of propositions disjoint from Σ. Recall that $\mathsf{A}(\varphi)$ is the set of atomic formulas that are subformulas of φ. We assume a bijective function $\mathsf{pmap} : \mathsf{A}(\varphi) \to P$ that relates the atoms in φ to propositions in P.

Definition 4.5.1 (Boolean Skeleton)**.** The *Boolean skeleton* is the propositional formula $\mathsf{BoolSkel}(\varphi)$ obtained by replacing each atomic formula $\psi_{\mathcal{A}}$ occurring in φ with $\mathsf{pmap}(\psi_{\mathcal{A}})$.

The propositions of the Boolean skeleton stand for atomic formulas. This is a form of abstraction. Below, we define the resulting abstraction directly in terms of atomic formulas (we consider the fact that they are represented as propositions an implementation detail).

Definition 4.5.2 (Boolean Abstraction)**.** The *Boolean abstraction* Bool_F for a set of formulas $F \subseteq \mathcal{F}$ is the powerset lattice $(\mathcal{P}(F \to \mathbb{B}), \subseteq)$.

Proposition 4.5.1. *The Boolean abstraction overapproximates the concrete domain.*

$$(\mathcal{P}(\mathcal{S}), \subseteq) \underset{\alpha_{\mathsf{B}}}{\overset{\gamma_{\mathsf{B}}}{\leftrightarrows}} (\mathsf{Bool}_F, \subseteq)$$

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

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

Since we are operating in two logics, we have deduction and abduction transformers for each. The first-order deduction and abduction transformers are as follows.

$$\mathit{ded}_{\varphi,\mathcal{S}} : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S}) \qquad \mathit{abd}_{\varphi,\mathcal{S}} : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$$

Recall that $\mathcal{S}_P \mathrel{\hat{=}} P \to \mathbb{B}$ is the set of propositional structures. The propositional deduction and abduction transformers are as follows.

$$\mathit{ded}_{\varphi,\mathcal{S}_P} : \mathcal{P}(\mathcal{S}_P) \to \mathcal{P}(\mathcal{S}_P) \qquad \mathit{abd}_{\varphi,\mathcal{S}_P} : \mathcal{P}(\mathcal{S}_P) \to \mathcal{P}(\mathcal{S}_P)$$

We lift the function pmap to map sets of first-order structures to sets of corresponding propositional structures $\mathsf{pmap}(S) \mathrel{\hat{=}} \{\lambda a.\beta(\mathsf{pmap}(a)) \mid \beta \in S\}$.

Relating Boolean Abstractions and the Skeleton

We now show how Boolean abstractions can be used to determine satisfiability of a formula.

Example 4.5.1. *Consider the first-order formula below, where the equality predicate has its standard interpretation.*

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

The set of atoms of the formula is $\mathsf{A}(\varphi) = \{x = y, y = z, x = z\}$*. An element of the Boolean abstraction* $\mathsf{Bool}_{\mathsf{A}(\varphi)}$ *is a set of mappings from these atoms to truth values. We denote by* $v_1v_2v_3$ *the truth assignment* $\{(x = y) \mapsto v_1, (y = z) \mapsto v_2, (x = z) \mapsto v_3\}$ *in* $\mathsf{A}(\varphi) \to \mathbb{B}$*. We define a mapping function* $\mathsf{pmap} \mathrel{\hat{=}} \{(x = y) \mapsto p, (y = z) \mapsto q, (x = z) \mapsto r\}$ *from which we obtain the Boolean skeleton below.*

$$\mathsf{BoolSkel}(\varphi) \mathrel{\hat{=}} p \wedge (\neg q \vee \neg r)$$

We use a transformer $\mathsf{BSkelModels}$ *over* $\mathsf{Bool}_{\mathsf{A}(\varphi)}$*, which finds all models of the skeleton.*

$$\mathsf{BSkelModels}(\top) = \{\mathsf{ttf}, \mathsf{tft}, \mathsf{tff}\}$$

Note that $\mathsf{BSkelModels}(\top)$ *contains an assignment* ttf*, which represents the empty set since no structure in the theory satisfies* $x = y$ *and* $y = z$*, but not* $x = z$*. The same holds for* tft*. Since both of these assignments concretize to the empty set, this does not affect the precision of the transformer* $\mathsf{BSkelModels}$*. The transformer* $\mathsf{BSkelModels}$ *is* γ*-complete at* $\top$ *since* $\gamma_{\mathsf{B}}(\mathsf{BSkelModels}(\top))$ *is equal to* $ded_\varphi(\top)$*.*

In order to decide satisfiability, we eliminate such spurious assignments by using a reduction operator $\mathsf{BoolCheck}$*. This transformer individually checks the truth assignments. If they do not represent any first-order models, they are removed from the set.*

Calling $\mathsf{BoolCheck}(\{\mathsf{ttf}, \mathsf{tft}, \mathsf{tff}\})$ *refines the representation to* $\{\mathsf{tff}\}$*. Since the set is non-empty, we can conclude that the formula is satisfiable.*

We define deduction transformers for propositional and theory reasoning. Applying the concrete propositional transformer to the Boolean skeleton of a first-order formula yields an abstract transformer for first-order reasoning in the domain $\mathsf{Bool}_{\mathsf{A}(\varphi)}$. This shows that DPLL(T) computes an abstract transformer when it performs propositional reasoning.

Proposition 4.5.2. *Let* $\psi = \mathsf{BoolSkel}(\varphi)$*, then the* skeleton transformer

$$\mathsf{BSkelModels} \mathrel{\hat{=}} \mathsf{pmap}^{-1} \circ ded_{\psi,\mathcal{S}_P} \circ \mathsf{pmap}$$

is a sound overapproximation of the deduction transformer $ded_{\varphi,\mathcal{S}}$*.*

Proof. We prove that every set of assignments $B \in \mathsf{Bool}_{\mathsf{A}(\varphi)}$ satisfies $ded_{\varphi,\mathcal{S}} \circ \gamma_{\mathsf{B}}(B) \subseteq \gamma_{\mathsf{B}} \circ \mathsf{BSkelModels}(B)$.

Consider a structure σ in the set $ded_{\varphi,\mathcal{S}} \circ \gamma_{\mathsf{B}}(B)$. Then σ is a model of φ and there is an assignment β in B such that $\sigma \in \gamma_{\mathsf{B}}(\{\beta\})$ and $\forall \psi \in \mathsf{A}(\varphi).\ \sigma \models \psi \iff \beta(\psi) = \mathsf{t}$. We now argue that this β corresponds to a model of $\mathsf{BoolSkel}(\varphi)$, given by $\rho = \lambda p.\ \beta(\mathsf{pmap}^{-1}(\psi_{\mathcal{A}}))$. Let C be some clause in $\mathsf{BoolSkel}(\varphi)$.

Then there is some first-order clause $C' \in \varphi$ with $\mathsf{BoolSkel}(C') = C$. Since $\sigma \models \varphi$, it must also hold that $\sigma \models C'$ and there is some literal $l' \in C'$ such that $\sigma \models l'$. We assume l' is an atomic formula $\psi_{\mathcal{A}}$; the case for a negative literal is similar. Since σ satisfies $\psi_{\mathcal{A}}$, we know that $\beta(\psi_{\mathcal{A}}) = \mathsf{t}$. Then the clause C contains a literal $l = \mathsf{BoolSkel}(l')$ such that the assignment ρ satisfies l.

It follows that every clause $C \in \mathsf{BoolSkel}\varphi$ has a literal l such that ρ satisfies l. Therefore, ρ is a model of $\mathsf{BoolSkel}(\varphi)$. It follows that β is in $\mathsf{BSkelModels}(B)$. Therefore, σ is in $\gamma_{\mathsf{B}} \circ \mathsf{BSkelModels}(B)$, which concludes the proof. □

The transformer $\mathsf{BSkelModels}$ defined above is not the best overapproximation of the first order-deduction transformer, since it only captures Boolean reasoning, but not theory reasoning. While it doesn't yield optimal results in the abstraction, it yields precise results in the concrete. This is evidenced by the following completeness property.

Proposition 4.5.3. $\mathsf{BSkelModels}$ *is γ-complete w.r.t. $ded_{\varphi,\mathcal{S}}$, i.e.,*

$$\gamma_{\mathsf{B}} \circ \mathsf{BSkelModels} = ded_{\varphi,\mathcal{S}} \circ \gamma_{\mathsf{B}}.$$

Proof. The direction $\supseteq$ follows from Proposition 4.5.2. To show $\subseteq$, pick some element $\sigma \in \gamma_{\mathsf{B}} \circ \mathsf{BSkelModels}$. Then there is a β in B such that $\sigma \in \gamma_{\mathsf{B}}(\{\beta\})$, hence $\forall \psi \in \mathsf{A}(\varphi).\ \sigma \models \psi \iff \beta(\psi) = \mathsf{t}$. From the proof of Proposition 4.5.2, we have that $\rho = \lambda p.\ \beta(\mathsf{pmap}^{-1}(p))$ is a model of $\mathsf{BoolSkel}(\varphi)$. We also know that $\sigma \models \psi_{\mathcal{A}}$ for $\psi_{\mathcal{A}} \in \mathsf{A}(\varphi)$ exactly if $\beta(\psi_{\mathcal{A}}) = \mathsf{t}$. Since every clause in $\mathsf{BoolSkel}(\varphi)$ has a literal that is assigned to true by ρ, it then follows that every clause of φ has a literal l such that $\sigma \models \varphi$. Therefore, σ is a model of φ. □

In other words, even though the result of applying $\mathsf{BSkelModels}$ is not the best abstract representation of the set of models of φ, its concretization precisely describes that set. In order to decide satisfiability of φ, one essential question is whether the set of models computed by $\mathsf{BSkelModels}$ represents the empty set.

In DPLL(T), the question of whether an assignment to the Boolean skeleton $\mathsf{BoolSkel}(\varphi)$ represents a set of models of φ is answered using a satisfiability checker for conjunctions of atomic formulas.

Definition 4.5.3 (Boolean Candidate Checking)**.** The function $\mathsf{BoolCheck} : \mathsf{Bool}_F \to \mathsf{Bool}_F$ defined below eliminates assignments not consistent in the theory.

$$\mathsf{BoolCheck}(B) \mathrel{\hat{=}} \Big\{\beta \in B \mid \bigwedge\{\varphi \mid \beta(\varphi) = \mathsf{t}\} \cup \{\neg\varphi \mid \beta(\varphi) = \mathsf{f}\} \text{ is } \mathcal{S}\text{-}\mathsf{SAT}\Big\}$$

Theory solvers are used to eliminate candidate solutions. Since candidate solutions have empty concretizations, this refines the representation of the abstract element without changing its semantics. In an abstract lattice, this is precisely the defining criterion of a semantic reduction operator.

Proposition 4.5.4. $\mathsf{BoolCheck}$ *is a semantic reduction operator.*

Theory solvers are required to be refutationally complete for conjunctions of theory atoms, that is, whenever a conjunction of atoms is unsatisfiable, the theory solver has to be able to conclusively determine that fact. In abstract interpretation terms, this means that if the formula is unsatisfiable, then $\mathsf{BoolCheck}$ must return the empty set, which is the bottom element of the lattice. To capture this requirement, we introduce a new completeness property.

Definition 4.5.4 (Complete $\bot$-Check)**.** A semantic reduction operator $\rho : A \to A$ is a *complete $\bot$-check* if the following holds.

$$\forall a \in A.\ \gamma(a) = \bot_C \implies \rho(a) = \bot_A$$

Proposition 4.5.5. *The function* $\mathsf{BoolCheck}$ *is a complete $\bot$-check.*

Proof. We show the contrapositive, i.e., $\gamma_{\mathsf{B}}(B)$ is non-empty if $\mathsf{BoolCheck}(B)$ is non-empty. Assume that $\mathsf{BoolCheck}(B)$ is non-empty. Then there is some $\beta \in B$ such that the formula $\bigwedge\{\varphi \mid \beta(\varphi) = \mathsf{t}\} \cup \{\neg\varphi \mid \beta(\varphi) = \mathsf{f}\}$ is $\mathcal{S}$ satisfiable. Let $\sigma \in \mathcal{S}$ be a model of the above formula. Then $\forall \psi \in F.\ \sigma \models \psi \iff \beta(\psi) = \mathsf{t}$. Therefore σ is in $\gamma_{\mathsf{B}}(B)$ and $\gamma_{\mathsf{B}}(B)$ is non-empty. □

4.5.2 Partial Assignments and 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) uses the CDCL algorithm to enumerate propositional models rather than computing the set of all models up-front. The CDCL solver that is integrated into DPLL(T) uses a partial assignment data structure, where the propositions represent ground first-order atoms. We now show that partial assignments in a DPLL(T) solver are an overapproximation of the first-order semantics. This shows that DPLL(T) solvers are abstract interpreters for first-order formulas.

Definition 4.5.5 (Cartesian Formula Abstraction)**.** For a set of formulas $F \subseteq \mathcal{F}$, we define the *Cartesian abstraction* Cart_F as the lattice $(\mathcal{P}(F \cup \neg F), \sqsubseteq)$ where the ordering $\sqsubseteq$ is the superset ordering, meets are given by $\bigcup$ and joins by $\bigcap$.

Proposition 4.5.6. *The lattice* Cart_F *overapproximates* Bool_F *and* $\mathcal{P}(\mathcal{S})$ *via the Galois connections below.*

$$\begin{tikzcd}[column sep=huge]
(\mathcal{P}(\mathcal{S}), \subseteq) \arrow[r, shift right, "\alpha_{\mathsf{B}}"'] \arrow[rr, bend right, shift right, "\alpha_{\mathsf{C}} \,\hat{=}\, \alpha_{\mathsf{BC}} \circ \alpha_{\mathsf{B}}"'] & (\mathsf{Bool}_F, \subseteq) \arrow[l, shift right, "\gamma_{\mathsf{B}}"'] \arrow[r, shift right, "\alpha_{\mathsf{BC}}"'] & (\mathsf{Cart}_F, \sqsubseteq) \arrow[l, shift right, "\gamma_{\mathsf{BC}}"'] \arrow[ll, bend left, shift right, "\gamma_{\mathsf{C}} \,\hat{=}\, \gamma_{\mathsf{BC}} \circ \gamma_{\mathsf{B}}"']
\end{tikzcd}$$

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

Partial assignments in existing DPLL(T) solvers represent $\mathsf{Cart}_{\mathsf{A}(\varphi)}$. Reasoning with the unit rule may be considered a transformer $\mathit{unit}_{C,\mathsf{A}(\varphi)} : \mathsf{Cart}_{\mathsf{A}(\varphi)} \to \mathsf{Cart}_{\mathsf{A}(\varphi)}$.

We now recall the unit rule and BCP as transformers on $\mathsf{Cart}_{\mathsf{A}(\varphi)}$. Let C be a clause of a ground, first-order formula φ. In the following, we denote the opposite phase of a literal l by $\mathsf{neg}(l)$.

$$\mathit{unit}_{C,\mathsf{A}(\varphi)}(\Theta) \mathrel{\hat{=}} \begin{cases} \bot & \text{if } \forall l \in C.\ \mathsf{neg}(l) \in \Theta \\ \Theta \sqcap \{l'\} & \text{else, if } \forall l \in C \setminus \{l'\}.\ \mathsf{neg}(l) \in \Theta \\ \Theta & \text{otherwise} \end{cases}$$
$$\mathit{bcp}_{\varphi,\mathsf{A}(\varphi)}(\Theta) \mathrel{\hat{=}} \mathsf{gfp}\, X.\ \bigsqcap_{C \in \varphi} \mathit{unit}_{C,\mathsf{A}(\varphi)}(X \sqcap \Theta)$$

Recall the Boolean reduction operator $\mathsf{BSkelModels}$, which helps eliminate propositional truth assignments from a set if they did not correspond to first-order models. One application of $\mathsf{BSkelModels}$ corresponds to a set of calls to the theory solver, one for each assignment in the set. We now define a similar reduction operator over Cart_F which corresponds to a single call to the theory solver.

Definition 4.5.6. We define $\mathsf{CartCheck} : \mathsf{Cart}_F \to \mathsf{Cart}_F$ as follows.

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

Proposition 4.5.7. $\mathsf{CartCheck}$ *is a complete* $\bot$*-check.*

Proof. We show the contrapositive that whenever $\mathsf{CartCheck}(\Theta) \neq \bot$ then $\gamma_{\mathsf{C}}(\Theta) \neq \emptyset$. Assume that $\mathsf{CartCheck}(\Theta) \neq \bot$. It follows that $\bigwedge \theta$ is satisfiable. Fix a model $\sigma \in \mathcal{S}$ of the above formula. It holds that that $\alpha_{\mathsf{B}}(\{\sigma\}) \subseteq \gamma_{\mathsf{BC}}(\Theta)$. By the properties of the Galois connection, we obtain that $\{\sigma\} \subseteq \gamma_{\mathsf{B}} \circ \gamma_{\mathsf{BC}}(\Theta)$. Therefore, Θ has a non-empty concretization. □

4.5.3 Theory Solvers as Abstract Domains

So far, we have modeled theory solvers as reduction operators. In this section, we show that theory solvers for equality with uninterpreted functions and for difference logic can be viewed as abstract interpreters. These serve as examples of the general approach to formalizing theory solvers as abstract interpreters. It is not feasible to give similar characterizations for all theory solvers in the literature in this dissertation.

Equality with Uninterpreted Functions

An *equality formula* contains the predicate $=$ and function symbols. We use $t \neq t'$ as a shorthand for $\neg(t = t')$. We define the theory of Equality with Uninterpreted Functions (EUF) as the set T_{EUF} containing all first-order structures $(\mathbb{Z}, \xi)$, where ξ interprets $=$ as the standard equality relation on $\mathbb{Z}$.

The congruence closure algorithm decides satisfiability of conjunctions of equality literals [140]. The algorithm constructs congruence classes from the set of terms $\mathsf{H}(\varphi)$ of φ and a set of pairs in $\mathsf{H}(\varphi)$ 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. $\Lambda(X)$ denotes the set of partitions of a set X, which forms a lattice. Congruence closure operates on the product of the partitions lattice $\Lambda(\mathsf{H}(\varphi))$ of the set of terms, and the powerset lattice $\mathcal{P}(\mathsf{H}(\varphi) \times \mathsf{H}(\varphi))$ of pairs of terms. An element (P, D) of the lattice contains a partition P where each block consists of a set of elements that are known to be equal, and a set of pairs D over terms, which are known to be different.

Definition 4.5.7. For an EUF formula φ, the EUF *lattice* EUF_φ is $(\mathsf{TS}, \sqsubseteq)$ where:

$$\mathsf{TS} \mathrel{\hat{=}} \Lambda(\mathsf{H}(\varphi)) \times \mathcal{P}(\mathsf{H}(\varphi) \times \mathsf{H}(\varphi))$$

and $(P, D) \sqsubseteq (P', D')$ exactly if $\forall p' \in P'. \exists p \in P$ s.t. $p \supseteq p'$ and $D \supseteq D'$.

Proposition 4.5.8. *The lattice* EUF_φ *overapproximates* $\mathcal{P}(\mathcal{S})$ *and is overapproximated by* $\mathsf{Cart}_{\mathsf{A}(\varphi)}$ *with respect to the following Galois connections.*

$$(\mathcal{P}(\mathcal{S}), \subseteq) \underset{\alpha_{\mathsf{TS}}}{\overset{\gamma_{\mathsf{TS}}}{\leftrightarrows}} (\mathsf{TS}, \sqsubseteq) \underset{\mathsf{T2B}}{\overset{\mathsf{B2T}}{\leftrightarrows}} (\mathsf{Cart}_{\mathsf{A}(\varphi)}, \supseteq)$$

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

Recall that $\mathsf{L}(\varphi)$ in the above is the set of literals that are subformulas of φ.

Proof. In order to show that γ_{TS} is the upper adjoint of a Galois connection, it is sufficient to show that it is completely multiplicative (Proposition 2.3.3). Since all lattices here are finite, this is equivalent to showing that γ_{TS} is multiplicative.

$$\gamma_{\mathsf{TS}}(a_1) \sqcap \gamma_{\mathsf{TS}}(a_2) = \gamma_{\mathsf{TS}}(a_1 \sqcap a_2)$$

Let $a_1 = (P_1, D_1)$ and $a_2 = (P_2, D_2)$ then:

$$\begin{aligned}
& \gamma_{\mathsf{TS}}(a_1) \sqcap \gamma_{\mathsf{TS}}(a_2) \\
= \ & \{\sigma \mid \forall (t_1, t_2) \in D_1.\ \sigma \models t_1 \neq t_2 \wedge \forall p \in P_1\ \forall t_1, t_2 \in p.\ \sigma \models t_1 = t_2\} \cap \\
& \{\sigma \mid \forall (t_1, t_2) \in D_2.\ \sigma \models t_1 \neq t_2 \wedge \forall p \in P_2\ \forall t_1, t_2 \in p.\ \sigma \models t_1 = t_2\} \\
= \ & \{\sigma \mid \forall (t_1, t_2) \in D_1 \cup D_2.\ \sigma \models t_1 \neq t_2 \wedge \forall p \in P_1 \cup P_2\ \forall t_1, t_2 \in p.\ \sigma \models t_1 = t_2\} \\
= \ & \{\sigma \mid \forall (t_1, t_2) \in D_1 \cup D_2.\ \sigma \models t_1 \neq t_2 \wedge \forall p \in P'\ \forall t_1, t_2 \in p.\ \sigma \models t_1 = t_2\} \\
= \ & \gamma_{\mathsf{TS}}(P', D_1 \cup D_2) \\
= \ & \gamma_{\mathsf{TS}}(a_1 \sqcap a_2)
\end{aligned}$$

where P' is the meet (in the lattice of partitions) of P_1 and P_2.

Similarly, to show that $\mathsf{T2B}$ is a Galois connection we show that it preserves joins.

Let $a_1 = (P_1, D_1)$ and $a_2 = (P_2, D_2)$ then:

$$\begin{aligned}
& \mathsf{T2B}(a_1) \sqcup \mathsf{T2B}(a_2) \\
= \ & \mathsf{L}(\varphi) \cap (\{t_1 = t_2 \mid \exists p \in P_1.\ t_1, t_2 \in p\} \cup \{t_1 \neq t_2 \mid (t_1, t_2) \in D_1\}) \cap
\end{aligned}$$

$$\begin{aligned}
& \mathsf{L}(\varphi) \cap (\{t_1 = t_2 \mid \exists p \in P_1.\ t_1, t_2 \in p\} \cup \{t_1 \neq t_2 \mid (t_1, t_2) \in D_2\}) \\
= \ & \mathsf{L}(\varphi) \cap (\{t_1 = t_2 \mid \exists p \in P_1.\ t_1, t_2 \in p \wedge \exists p \in P_2.\ t_1, t_2 \in p\} \cup \\
& \{t_1 \neq t_2 \mid (t_1, t_2) \in D_1 \cap D_2\}) \\
= \ & \mathsf{L}(\varphi) \cap (\{t_1 = t_2 \mid \exists p \in P'.\ t_1, t_2 \in p\} \cup \{t_1 \neq t_2 \mid (t_1, t_2) \in D_1 \cap D_2\}) \\
= \ & \mathsf{T2B}(P', D_1 \cap D_2) \\
= \ & \mathsf{T2B}(a_1 \sqcup a_2)
\end{aligned}$$

where P' is the join (in the lattice of partitions) of P_1 and P_2. □

Congruence closure starts with the top element of the lattice, where all terms are placed in different partitions and the set of known disequalities is empty. The algorithm then iteratively refines this structure using two operations. First, for every literal $t_1 = t_2$, the partitions of t_1 and t_2 are merged, and for every literal $t_1 \neq t_2$, a known disequality is added. This process represents the initial transfer of information from the Cartesian abstraction $\mathsf{Cart}_{\mathsf{A}(\varphi)}$ to the EUF abstraction. This transfer is an approximation of $\mathsf{B2T}$ and maps an element of the Cartesian abstraction to a semantically equivalent element of the EUF abstraction. Second, the resulting element is refined using the congruence rule, which may be stated as a proof rule as follows.

$$\frac{t_1 = t_1' \quad \ldots \quad t_k = t_k'}{f(t_1, \ldots, f_k) = f(t_1', \ldots, f_k')}$$

Example 4.5.2. *Consider the following element of the Cartesian abstraction* $\mathsf{Cart}_{\mathsf{A}(\varphi)}$.

$$\Theta = \{a = b, b = c, f(a) \neq f(c)\}$$

Transferring the information in Θ *to the* EUF *abstraction yields the following.*

$$(\{\{a, b, c\}, \{f(a)\}, \{f(c)\}\},\ \{(f(a), f(c))\})$$

Applying congruence, we obtain the following.

$$(\{\{a, b, c, f(a), f(c)\}\},\ \{(f(a), f(c))\})$$

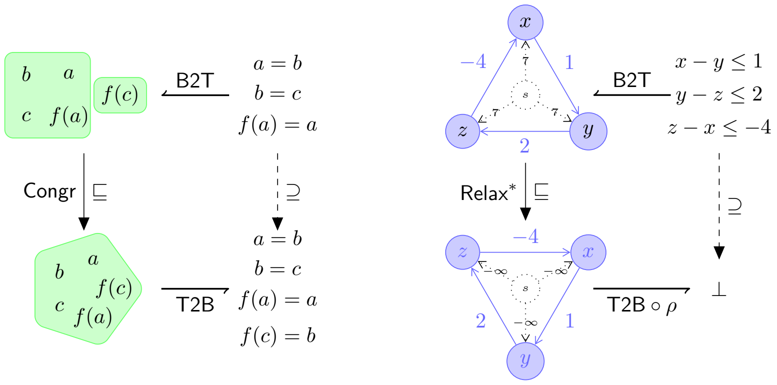
Since we know that both $f(a) = f(c)$ *and* $f(a) \neq f(c)$, *we may reduce to* $\bot$.

We define congruence as a transformer. 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 be 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 deflationary and refines the representation of a set of structures without changing the set itself. Figure 4.9(a) illustrates Congr along with the Galois connection between the EUF and the Cartesian abstractions. The set of formulas in the top right can be concretized 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 formulas in the bottom right.

Proposition 4.5.9. Congr *is a reduction operator.*



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

Figure 4.9: Examples of theory solvers as abstract domains.

Proof. To show that Congr is a transformer, it is necessary to show it is order-preserving. Let $(P, D) \sqsubseteq (P', D')$. It's easy to see that Congr is deflationary. If $\mathsf{Congr}(P', D') = (P', D')$ it therefore follows immediately that $\mathsf{Congr}(P, D) \sqsubseteq \mathsf{Congr}(P', D')$. Now consider the case where $\mathsf{Congr}(P', D')$ is strictly less than (P', D'), then:

$$\exists p, p' \in P'. f(s_1, \ldots, s_k) \in p \wedge f(t_1, \ldots, t_k) \in p' \text{ s.t. all } s_i, t_i \text{ are congr. in } P$$

By the ordering of partitions, the conditions on s_i and t_i hold for P, giving two options: if $f(s_1, \ldots, s_k)$ and $f(t_1, \ldots, t_k)$ are congruent in P then $(P, D) \sqsubseteq \mathsf{Congr}(P', D')$, otherwise (P, D) can be reduced and thus $\mathsf{Congr}(P, D) \sqsubseteq \mathsf{Congr}(P', D')$.

To show that Congr is a reduction operator it remains to show that $\gamma_{\mathsf{TS}} \circ \mathsf{Congr} = \gamma_{\mathsf{TS}}$. Again, the only case that needs to be considered is when each pair of s_i and t_i are congruent and $f(s_1, \ldots, s_k)$ and $f(t_1, \ldots, t_k)$ are not in the same congruence class. In this case, it holds that $f(s_1, \ldots, s_k) = f(t_1, \ldots, t_k)$, which implies that their equivalence classes does not remove any models, i.e. $\gamma_{\mathsf{TS}} \circ \mathsf{Congr} = \gamma_{\mathsf{TS}}$. □

The congruence closure algorithm computes the greatest fixed point $\mathsf{gfp}\ \mathsf{Congr}$. It is a refutationally complete procedure; if a conjunction of equality literals is empty, then the fixed point will be $\bot$.

Proposition 4.5.10. *The transformer* Congr^* *defined as* $\mathsf{Congr}^*(a) \mathrel{\hat{=}} \mathsf{gfp}\, X.\ \mathsf{Congr}(X \sqcap a)$ *is a complete* $\bot$*-check.*

Proof. The function Congr^* is a complete $\bot$-check if:

$$\gamma_{\mathsf{TS}}(a) = \bot \Rightarrow \mathsf{Congr}^*(a) = \bot$$

From the definition of γ_{TS}, an element of the abstraction only concretizes to $\bot$ when there is at least one pair in D that is contained in the same partition within P. This is the only situation in which Congr returns $\bot$, thus the two conditions are equivalent and Congr^* is a complete $\bot$-check. □

Difference Logic

Formulas in *difference logic* (DL) contain the binary subtraction symbol $-$ and the binary predicate $\leq$. Atoms in difference logic have the form $x - y \leq c$. The theory of *integer difference logic* (T_{IDL}) is the set of structures of the form $(\mathbb{Z}, \xi)$, where ξ maps the symbols $\leq$ and $-$ to their standard interpretations over the integers.

A conjunction of difference logic atoms can be modeled by a weighted directed graph in which a node represents a variable. 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, the source is chosen to be a fresh node s which is connected to all variable nodes. In algorithmic presentations of Bellman-Ford, the weight of this connection is often chosen to be infinite. We choose instead a weight M_φ, which is an integer constant larger than the longest possible path, e.g., the sum of the absolute values of all the integer constants in φ. The initial node weights are also M_φ. Node weights are reduced in each round if there is a neighboring 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 be viewed as special edges (s, n), (ii) BF can then be viewed as operating solely over edge weights (missing edges are given weight ∞). For a formula φ, we define the edge set E_φ as the set $(\{s\} \cup \mathsf{V}(\varphi)) \times \mathsf{V}(\varphi)$, where s is the fresh source node.

Definition 4.5.8. For a DL-formula φ, the BF *abstraction* BF_φ is the lattice $(\mathsf{TS}, \sqsubseteq)$ below.

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

Proposition 4.5.11. *The lattice* BF_φ *overapproximates the concrete domain and is overapproximated by* $\mathsf{Cart}_{\mathsf{A}(\varphi)}$ *with respect to the following Galois connection.*

$$\begin{tikzcd}
(\mathcal{P}(\mathcal{S}), \subseteq) \arrow[r, shift right, "\alpha_{\mathsf{TS}}"'] & (\mathsf{TS}, \sqsubseteq) \arrow[l, shift right, "\gamma_{\mathsf{TS}}"'] \arrow[r, shift right, "\mathsf{T2B}"'] & (\mathsf{Cart}_{\mathsf{A}(\varphi)}, \supseteq) \arrow[l, shift right, "\mathsf{B2T}"']
\end{tikzcd}$$
$$\gamma_{\mathsf{TS}}(f) \mathrel{\hat{=}} \{\sigma \mid \forall (x, y) \in \mathsf{V}(\varphi) \times \mathsf{V}(\varphi).\ \sigma \models\ x - y \leq f(x, y)\}$$
$$\mathsf{B2T}(\Theta) \mathrel{\hat{=}} \lambda(x, y).\ \min(\{k \mid x - y \leq k \in \Theta\} \cup \{\top_{\mathsf{BF}}(x, y)\})$$

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 4.5.12. $\mathsf{Relax} : \mathsf{BF}_\varphi \to \mathsf{BF}_\varphi$ *and* $\mathsf{NegC} : \mathsf{BF}_\varphi \to \mathsf{BF}_\varphi$*, defined below, are reductions.*

$$\mathsf{Relax}(f)(x, y) \mathrel{\hat{=}} \begin{cases} f(x, y) & x \neq s \\ \min(\{f(x, y)\} \cup \{f(x, z) + f(z, y) \mid z \in \mathsf{V}(\varphi)\}) & x = s \end{cases}$$
$$\mathsf{NegC}(f) \mathrel{\hat{=}} \begin{cases} \bot & \textit{if } \mathsf{Relax}^{|\mathsf{V}(\varphi)|} \neq \mathsf{Relax}^{|\mathsf{V}(\varphi)|-1} \\ \mathsf{Relax}^{|\mathsf{V}(\varphi)|} & \textit{otherwise} \end{cases}$$

In addition to the above functions, consider a reduction ρ s.t. $\rho(f) = \bot$ if f maps some edge to $-\infty$ and $\rho(f) = f$ otherwise. The functions Relax, ρ and the Galois connections to the Cartesian domain are shown in Figure 4.9(b). Similarly to Figure 4.9(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) using the relaxation function.

Proposition 4.5.13. $\mathsf{NegC}(a)$ *computes the following fixed point and is a complete* $\bot$*-check.*

$$\mathsf{gfp}\, X.\ \rho \circ \mathsf{Relax}(X \sqcap a)$$

4.5.4 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 as computing a fixed point in the product of a Cartesian and a theory domain.

Definition 4.5.9. (Theory Domain) We define a DPLL(T) *theory domain* to be an lattice $(\mathsf{TS}, \sqsubseteq)$ such that the following conditions hold.
(i) TS abstracts the concrete w.r.t. the Galois connection $(\alpha_{\mathsf{TS}}, \gamma_{\mathsf{TS}})$,
(ii) $\mathsf{Cart}_{\mathsf{A}(\varphi)}$ abstracts TS w.r.t. the Galois connection $(\mathsf{T2B}, \mathsf{B2T})$,
(iii) $\gamma_{\mathsf{C}} = \gamma_{\mathsf{TS}} \circ \mathsf{B2T}$ and $\alpha_{\mathsf{C}} = \alpha_{\mathsf{TS}} \circ \mathsf{T2B}$.

$$\begin{tikzcd}
(\mathcal{P}(\mathcal{S}), \subseteq) \arrow[r, shift right, "\alpha_{\mathsf{TS}}"'] \arrow[rr, bend right, "\alpha_{\mathsf{C}} \mathrel{\hat{=}} \mathsf{T2B}\circ\alpha_{\mathsf{TS}}"'] & (\mathsf{TS}, \sqsubseteq) \arrow[l, shift right, "\gamma_{\mathsf{TS}}"'] \arrow[r, shift right, "\mathsf{T2B}"'] & (\mathsf{Cart}_{\mathsf{A}(\varphi)}, \sqsubseteq) \arrow[l, shift right, "\mathsf{B2T}"'] \arrow[ll, bend left=50, "\gamma_{\mathsf{C}} \mathrel{\hat{=}} \gamma_{\mathsf{TS}}\circ\mathsf{B2T}"']
\end{tikzcd}$$

The first condition ensures that the data structure of the theory solver represent sets of $\mathcal{S}$ structures. The second condition ensures that conjunctions of literals in $\mathsf{A}(\varphi)$ can be expressed in TS without losing precision. This corresponds to the requirement that the fragment of the logic handled by the theory solver includes conjunctions of literals in $\mathsf{A}(\varphi) \cup \neg\mathsf{A}(\varphi)$, i.e., that satisfiability queries generated by $\mathsf{CartCheck}$ can be expressed in the theory solver. 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 4.5.10 (DPLL(T) Abstract Domain)**.** For a formula φ and a DPLL(T) theory domain TS, the DPLL(T) *abstract domain* $\mathsf{DPLL}(\mathsf{TS})$ is the product domain $\mathsf{Cart}_{\mathsf{A}(\varphi)} \times \mathsf{TS}$.

Both of the theory solvers presented earlier formed DPLL(T) abstract domains. We illustrate operations described in the following section in $\mathsf{DPLL}(\mathsf{EUF}_\varphi)$. For convenience, we denote for three terms x, $f(x)$ and z the partition $\{\{x\}, \{f(x), z\}\}$ either by $[x][f(x), z]$ or 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 architecture. There, an element $\Theta \in \mathsf{Cart}_{\mathsf{A}(\varphi)}$ 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 4.5.11 (Theory Instantiation and Theory Propagation)**.** We define the *theory instantiation* and *theory propagation* transformers in DPLL(T) below.

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

Example 4.5.3. *We assume that* $\mathsf{A}(\varphi) = \{x{=}y,\ y{=}z\}$*. Consider the following element of the abstract* DPLL(T) *domain* $\mathsf{DPLL}(\mathsf{EUF}_\varphi)$*.*

$$(\Theta, \mathsf{te}) \mathrel{\hat{=}} (\{x = y\}, ([x][y][z], \{y, z\}))$$

The application of tinst *to* (Θ, te) *yields* $(\Theta, ([x, y][z], \{y, z\}))$*. The application of* tprop *to* (Θ, te) *yields* $(\{x = y, \neg(y = z)\}, \mathsf{te})$*. Neither operator changes the semantics of the tuple.*

Proposition 4.5.14. *The transformers* tinst *and* tprop *are reductions.*

Proof. It is easy to see that both operators are deflationary, that is, they return a pair that is pointwise smaller than the argument. It remains to show that they are sound, i.e., that their result has the same concrete meaning as the argument. Consider a pair $(\Theta, \mathsf{te}) \in \mathsf{DPLL(TS)}$, with concretization $S = \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te})$. We consider separately the case of tinst and tprop.

(i) We have that $\mathsf{tinst}(\Theta, \mathsf{te}) = (\Theta, \mathsf{te} \sqcap \mathsf{B2T}(\Theta))$, with concrete meaning $S' = \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te} \sqcap \mathsf{T2B}(\Theta))$, which, due to the fact that γ_{TS} distributes over meets, can be expressed as $S' = \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te}) \cap \gamma_{\mathsf{TS}}(\mathsf{B2T}(\Theta))$. By definition, we can rewrite $\gamma_{\mathsf{TS}}(\mathsf{B2T}(\Theta))$ as the equivalent statement $\gamma_{\mathsf{C}}(\Theta)$. Replacing the above in S' yields $S' = \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te}) \cap \gamma_{\mathsf{C}}(\Theta) = \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te}) = S$. Therefore tinst preserves the semantics of the input element.

(ii) We have that $\mathsf{tprop}(\Theta, \mathsf{te}) = (\Theta \sqcap \mathsf{T2B}(\mathsf{te}), \mathsf{te})$, with concrete meaning $S' = \gamma_{\mathsf{C}}(\Theta \sqcap \mathsf{T2B}(\mathsf{te})) \cap \gamma_{\mathsf{TS}}(\mathsf{te})$. We distribute the meet over the concretization and obtain $S' = \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{C}}(\mathsf{T2B}(\mathsf{te})) \cap \gamma_{\mathsf{TS}}(\mathsf{te})$.
We now show that $\gamma_{\mathsf{C}}(\mathsf{T2B}(\mathsf{te}))$ is a superset of $\gamma_{\mathsf{TS}}(\mathsf{te})$, and therefore $S' = \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te}) = S$. We can rewrite the condition $\gamma_{\mathsf{C}} \circ \mathsf{T2B}(\mathsf{te}) \supseteq \gamma_{\mathsf{TS}}(\mathsf{te})$ as $\gamma_{\mathsf{TS}}(\mathsf{B2T} \circ \mathsf{T2B}(\mathsf{te})) \supseteq \gamma_{\mathsf{TS}}(\mathsf{te})$. Since γ_{TS} is monotone and $\mathsf{B2T} \circ \mathsf{T2B}(\mathsf{te}) \supseteq \mathsf{te}$ by the properties of the Galois connection, this condition holds.
Therefore $S' = S$, which proves that applying tprop returns an element that is semantically identical to the argument.

□

We note that *early pruning* [9] is just a special case of theory propagation in the lattice theoretic setting, i.e., the case where theory propagation finds $\bot$.

Deduction in the Cartesian domain and theory is modeled by transformers.

Definition 4.5.12 (Boolean Deduction Transformer)**.** The *Boolean deduction transformer* bded_φ is a sound overapproximation of $ded_{\varphi,\mathcal{S}}$.

In practice, bded_φ is bcp_φ, i.e., Boolean deduction is accomplished using Boolean constraint propagation, but in principle other sound abstract transformers could be used.

Definition 4.5.13 (Theory Deduction Transformer)**.** A *theory deduction transformer* tded is a sound overapproximation of $ded_{\varphi,\mathcal{S}}$.

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

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

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

Definition 4.5.14. We define $\mathsf{dplltded} : \mathsf{DPLL(TS)} \to \mathsf{DPLL(TS)}$ as follows.

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

Proposition 4.5.15. $\mathsf{dplltded}$ *is a sound overapproximation of* $ded_{\varphi,\mathcal{S}}$.

Proof. Both $\mathsf{bded}^{\times}_{\varphi}$ and $\mathsf{tded}^{\times}_{\varphi}$ are sound overapproximations of $ded_{\varpi,\mathcal{S}}$, the other operators are sound reduction operators. Composing sound approximations with reduction operators yields a sound approximation. □

Example 4.5.4. *Consider the formula φ given as $f(x) = y \land x = z \land (f(z) \neq y \lor y = z)$. We compute* $\mathsf{dplltded}$ *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.*

As in CDCL, model search corresponds to a downwards iteration sequence with extrapolation over the transformer $\mathsf{dplltded}$.

Conflict Analysis with Theory Explanations

DPLL(T) solvers are based on CDCL. The power of CDCL rests significantly in the conflict analysis step, which extracts sufficient conditions for unsatisfiability from specific contradictory cases. We describe conflict analysis abstractly (see Chapter 5 for a lifting of conflict analysis algorithms to abstract domains). Conflict analysis computes a least fixed point in the downset completion of the underlying domain (see Section 4.4). 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 [172].

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 $ded_{\varphi,\mathcal{S}}$. Conversely, finding explanations for deductions corresponds to underapproximation of the $abd_{\varphi,\mathcal{S}}$ transformer. As in the case of CDCL, we use the downset completion for the underapproximating domain, which capture the fact that multiple, different reasons may exist. The downset domain abstracts the product domain as follows.

$$(\mathcal{P}(\mathcal{S}), \subseteq) \mathrel{\substack{\xleftarrow{\gamma_{\mathcal{D}}} \\ \xrightarrow[\alpha_{\mathcal{D}}]{}}} \mathcal{D}(\mathsf{DPLL(TS)})$$

Definition 4.5.15 (Boolean Abduction Transformer)**.** A *Boolean abduction transformer* $\mathsf{babd}_\varphi : \mathcal{D}(\mathsf{Cart}_{\mathsf{A}(\varphi)}) \to \mathcal{D}(\mathsf{Cart}_{\mathsf{A}(\varphi)})$ is an underapproximation of $abd_{\varphi,\mathcal{S}}$ over the downset completion.

Example 4.5.5. *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, only to their internal state. Essentially, they correspond to abduction with respect to the formula t.

Definition 4.5.16 (Theory Abduction Transformer)**.** A *theory abduction transformer* $\mathsf{tabd} : \mathcal{D}(\mathsf{TS}) \to \mathcal{D}(\mathsf{TS})$ is a dual reduction.

Example 4.5.6. *Consider* $\mathsf{te} = ([x, y, z], \{(x, y), (y, z)\})$*, which represents a conflict. A sound abduction* $\mathsf{tabd}(\{\mathsf{te}\})$ *may return* $\{([x, y], \{(x, y)\}), ([y, z], \{(y, z)\})\}$*, highlighting two separate reasons for the conflict.*

We extend the functions babd_φ and tabd to sets in $\mathcal{D}(\mathsf{DPLL(TS)})$ 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 formula abstraction, the issue of expressibility arises since some abstract theory facts may not have precise counterparts in the Cartesian formula domain. For an element $\mathsf{te} \in \mathsf{TS}$, we write *expressible*$(\mathsf{te})$ to denote the condition that te is precisely expressible in $\mathsf{Cart}_{\mathsf{A}(\varphi)}$, i.e. $\gamma_{\mathsf{TS}}(\mathsf{te}) = \gamma_{\mathsf{C}} \circ \mathsf{T2B}(\mathsf{te})$.

Definition 4.5.17 (Theory Justification and Theory Explanation)**.** We define the *theory justification* and *theory explanation* transformer on $\mathcal{D}(\mathsf{DPLL}(\mathsf{TS}))$ below.

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

Example 4.5.7. *Consider a set of atoms* $\mathsf{A}(\varphi) = \{x{=}y,\ y{=}z\}$*, 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* $(\{x = y,\ \neg(y = z)\}, \top)$*.*

Proposition 4.5.16. *The transformers* tjustify *and* texpl *are semantics preserving, i.e., the following two equalities hold.*

$$\textit{(i)}\ \gamma_{\mathcal{D}} \circ \mathsf{tjustify} = \gamma_{\mathcal{D}} \qquad \textit{(ii)}\ \gamma_{\mathcal{D}} \circ \mathsf{texpl} = \gamma_{\mathcal{D}}$$

Proof. Consider an element (Θ, te') of $\mathsf{tjustify}(\Gamma)$. Then te' is of the form $\mathsf{te} \sqcap \mathsf{B2T}(\Theta')$ such that $(\Theta \sqcap \Theta', \mathsf{te}) \in \Gamma$. Concretizing (Θ, te') we obtain the following.

$$\begin{aligned} \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te}') &= \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te} \sqcap \mathsf{B2T}(\Theta')) \\ &= \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te}) \cap \gamma_{\mathsf{TS}}(\mathsf{B2T}(\Theta')) \\ &= \gamma_{\mathsf{C}}(\Theta) \cap \gamma_{\mathsf{TS}}(\mathsf{te}) \cap \gamma_{\mathsf{C}}(\Theta')) \\ &= \gamma_{\mathsf{C}}(\Theta \sqcap \Theta') \cap \gamma_{\mathsf{TS}}(\mathsf{te}) \end{aligned}$$

The final line is the concretization of the pair $(\Theta \sqcap \Theta', \mathsf{te})$. Therefore, every element of the set $\mathsf{tjustify}(\Gamma)$ has a semantically equivalent element in Γ, therefore $\gamma_{\mathcal{D}}(\mathsf{tjustify}(\Gamma)) \subseteq \gamma_{\mathcal{D}}(\Gamma)$. Furthermore, it is easy to see that every element in Γ is also in $\mathsf{tjustify}(\Gamma)$, therefore $\gamma_{\mathcal{D}}(\Gamma) \subseteq \gamma_{\mathcal{D}}(\mathsf{tjustify}(\Gamma))$, which completes the proof that $\gamma_{\mathcal{D}} \circ \mathsf{tjustify} = \gamma_{\mathcal{D}}$.

The proof that $\gamma_{\mathcal{D}} \circ \mathsf{texpl} = \gamma_{\mathcal{D}}$ is similar. □

The transformer 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 $\mathsf{Cart}_{\mathsf{A}(\varphi)}$. In both cases, the formula φ is not taken into consideration.

We note that *conflict set generation* [9] is a combination of theory abduction tabd of the $\perp$ element, followed by theory explanation.

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

Definition 4.5.18. We define the transformer dplltabd on $\mathcal{D}(\mathsf{DPLL}(\mathsf{TS}))$ as:

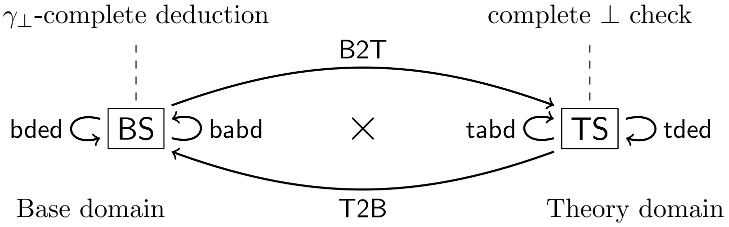
$$\mathsf{dplltabd} \mathrel{\hat{=}} \mathsf{texpl} \circ \mathsf{tabd}^{\times} \circ \mathsf{tjustify} \circ \mathsf{babd}^{\times}_{\varphi}$$

Proposition 4.5.17. dplltabd *is a sound underapproximation of* $\mathit{abd}_{\varphi,\mathcal{S}}$*.*

Proof. The transformers babd and tabd underapproximate $\mathit{abd}_{\varphi,\mathcal{S}}$ and and the transformers tjustify and texpl preserve semantics. The result follows. □

Conflict analysis can be viewed as computing a least fixed point over dplltabd, starting from a propositional conflict $\{(\perp, \mathsf{te})\}$ or theory conflict $\{(\Theta, \perp)\}$. In practice, solvers do not keep track of sets of explanations for a conflict, but will instead consider only one. As in CDCL, this may be considered an instance of upwards extrapolation.

We have demonstrated that DPLL(T) computes the Cartesian product of two abstract interpreters. Communication between the two components of the product amounts to reduction. Figure 4.10 summarizes the findings of this section.



	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 ded_{φ,T_Σ}
Theory Instantiation	$\mathsf{tinst}(\mathsf{be},\mathsf{te}) \mathrel{\hat{=}} \quad (\mathsf{be}, \mathsf{te} \sqcap \gamma_{\mathsf{TS}}(\mathsf{be}))$
Theory Propagation	$\mathsf{tprop}(\mathsf{be},\mathsf{te}) \mathrel{\hat{=}} \quad (\mathsf{be} \sqcap \alpha_{\mathsf{TS}}(\mathsf{te}), \mathsf{te})$
Deduction	$\mathsf{dplltded} \mathrel{\hat{=}} \quad \mathsf{tprop} \circ \mathsf{tded}^\times \circ \mathsf{tinst} \circ \mathsf{bded}^\times$
Model Search	gfp over $\mathsf{dplltded}$ with downwards extrapolation 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 abd_{φ,T_Σ}
Theory Justification	$\mathsf{tinst}(\Gamma) \mathrel{\hat{=}} \quad \{(\Theta, \gamma_{\mathsf{TS}}(\Theta') \sqcap \mathsf{te}) \mid (\Theta \sqcap \Theta', \mathsf{te}) \in \Gamma\}$
Theory Explanation	$\mathsf{texpl}(\Gamma) \mathrel{\hat{=}} \quad \{(\Theta \sqcap \mathsf{T2B}(\mathsf{te}), \mathsf{te}') \mid (\Theta, \mathsf{te} \sqcap \mathsf{te}') \in \Gamma$ s.t. $\mathit{expressible}(\mathsf{te})\}$
Abduction	$\mathsf{dplltabd}(\Gamma) \mathrel{\hat{=}} \quad \mathsf{texpl} \circ \mathsf{tabd}^\times \circ \mathsf{tjustify} \circ \mathsf{babd}^\times$
Conflict Analysis	lfp over $\mathsf{dplltabd}$ with upwards interpolation

Figure 4.10: DPLL(T) as a product construction.

4.6 Related Work

In this chapter, we presented abstract-interpretation accounts of popular satisfiability procedures.

Abstract Interpretation Accounts of Satisfiability Procedures Apart from the work this chapter is based on [58, 59, 25], there have recently been two other lines of work to explain decision procedures in terms of abstract interpretation. The first is an abstract interpretation account of Stålmarck's procedure [165], which also offers a lattice-theoretic generalization of the algorithm. An application of the resulting generalized algorithm to the problem of synthesizing abstract transformers was given in [166]. The work on Stålmarck's dilemma rule discussed in Section 4.3 was developed independently of the work in [165] since it is a simple consequence of characterizing DPLL as an abstract interpreter. The work in [165] is more faithful to the original algorithm than our presentation and contains a discussion of completeness.

In Section 4.5, we gave an abstract interpretation account of the DPLL(T) algorithm as an instantiation of a reduced product. One aspect we did not discuss is theory combination. Given two or more solvers for first-order theories, under certain conditions it is possible to derive a solver for the combination of the theories using the Nelson-Oppen procedure [139]. The work in [49] shows that Nelson-Oppen theory combination is an instance of the reduced product of theory solvers. This insight enables DPLL(T) solvers to be combined with an abstract interpreter. Note that Nelson-Oppen is distinct to the product construction that we identify within DPLL(T). We identify a product between the Boolean and the theory solver and we show that communication between the two corresponds to approximation of the reduced product, even in the absence of theory combination.

Downsets as Logics In Sections 4.2 and 4.5, we use downsets of partial assignments to formalize conflict analysis in SAT solvers. Partial assignments correspond to conjunctions over literals. The downset completion adds disjunctions acts as a kind of semantic counterpart to a DNF formula.

This idea is explored in more depth in the work by Schmidt on logic and abstract interpretation [160, 158, 159, 157, 156]. The work in [160] introduces the notions of an *internal logic* and *external logic* of an abstract domain. The internal logic of the partial assignments domain consists of conjunctions of literals. The external logic has disjunction and is obtained via a downset completion.

In this chapter, we showed that satisfiability solvers are abstract interpreters that use partition-based refinement. In Section 7.5, we will discuss abstract analyses that use partition-based refinement and abstract analyses that use satisfiability solvers as part of the analysis.

Chapter 5

Generalizations of Satisfiability Solvers

This chapter presents an approach based on the abstract satisfaction framework, for instantiating CDCL on new problems. The main contribution of this section is a strict, mathematical generalization of the CDCL algorithm to abstract lattices. We call the resulting algorithmic framework *Abstract Conflict-Driven Clause Learning* (ACDCL) [59]. The other contributions of this chapter are as follows:

- A characterization of conflict-driven learning in terms of learning transformers.
- A characterization of a class of lattices that support a generalization of clause learning.
- A set of completeness criteria for ACDCL.
- The AFIRSTUIP procedure [89], a lattice-theoretic generalization of the FIRSTUIP confict-analysis procedure used in CDCL solvers.

Motivation The performance of SAT solvers has improved at an exponential rate in the last decade. Several factors contribute to these improvements including an elegant algorithm, efficient, architecture-aware implementations of data structures, and heuristics that exploit the non-adversarial nature of practical problem instances. The result is an algorithmic framework referred to as the Conflict-Driven Clause Learning algorithm (CDCL).

SAT solvers based on CDCL have been applied to problems in a number of areas ranging from hardware and software verification to bioinformatics [117]. Problems in these areas are reduced to a formula, which is then solved by a SAT solver. This approach does not always work well because the SAT solver is not aware of the structure of the original problem domain. A focus of ongoing research, articulated in the quotation below from a survey by Malik and Zhang [115], is whether CDCL can be applied directly to new problems.

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

A well-explored strategy for extending CDCL to first-order theories is the DPLL(T) framework [143]. DPLL(T) solvers lift CDCL to first-order logics by combining a propositional, CDCL solver with a solver for conjunctions of theory literals. Learning in DPLL(T) is restricted to facts that can be encoded over a fixed propositional vocabulary. This restriction may prevent the solver from learning the most general theory reason for a conflict, which in turn reduces the efficiency of search. Program analysis frameworks inspired by DPLL(T) [91] suffer from similar issues. To ease these limitations, a number of refinements to DPLL(T) have been proposed, such as on-the-fly extensions of the propositional vocabulary [8].

An alternative to DPLL(T) is to lift CDCL to a new problem by identifying the counterparts of model search, conflict analysis and learning in the problem context. Such alternatives have been proposed for SMT logics [124, 37, 64, 169, 6, 104, 102, 56] and in a program analysis setting [123]. For example, in the lifting of CDCL to SMT in [124, 37], a decision is an assignment of a value to a first-order variable, while in the lifting to programs [123], a decision may correspond to restricting analysis to a single control-flow path. These liftings are ad-hoc and do not provide general principles for lifting CDCL to other problems.

The ACDCL framework, which we present in this section, strictly generalizes the CDCL algorithm in terms of lattices and abstract transformers. It is a framework for the systematic derivation of sound, problem-guided analysis procedures. Compared to other attempts at generalization, ACDCL has the following advantages:

- It is mathematically rigorous.
- It covers both logical satisfiability and program correctness.
- It is detailed enough to inform solver architecture and allows generic completeness proofs for classes of lattices.
- ACDCL can be implemented as a general, domain-independent algorithm that connects to an abstract domain via a well-defined abstract domain interface (see Chapter 6).

Our work also enables a new understanding of CDCL. Many existing lattices used in static analysis lack negation, have meet operations that precisely model conjunction and join operations that overapproximate disjunction. Precision loss due to joins is often eliminated by enriching a domain or analysis with disjunction. Such enrichment may suffer from *case explosion*, meaning that the number of disjunctive cases to be considered grows infeasibly large as the analysis progresses. The conceptual insight of this chapter is that learning techniques used by SAT solvers can be viewed as synthesizing an abstract transformer for negation. The combination of precise conjunction in the partial assignments domain with imprecise negation provided by learning allow a propositional solver to reason indirectly about disjunction without enumerating cases. ACDCL is a framework for lifting learning techniques in SAT solvers to analyzers that operate on non-distributive lattices. ACDCL equips such analyzers with a form of negation, which can be used to refine an analysis.

Outline Section 5.1 introduces Abstract Conflict-Driven Learning (ACDL), an abstract characterization of conflict-driven learning procedures in terms of fixed point computation over lattices and shows that learning, in the sense of CDCL, is overapproximation of negation. Section 5.2 gives a lattice-theoretic generalization of clause learning and the unit rule and characterizes a class of lattices over which these operations are possible. Section 5.3 presents the ACDCL algorithm, an instantiation of ACDL that uses a generalized unit rule as a learning transformer and operates on a lattice and its downset completion. Section 5.4 introduces AFIRSTUIP, a non-trivial generalization of the FIRSTUIP conflict analysis algorithm to lattices. Section 5.5 discusses other attempts to generalize CDCL and relates them to the ACDCL framework.

5.1 Abstract Conflict Driven Learning

This section presents *Abstract Conflict-Driven Learning* (ACDL), a framework for lattice-based learning algorithms, and gives a characterization of learning in the language of abstract interpretation. The goal of this section is to provide an abstract framework for developing conflict-driven learning algorithms and their proofs of soundness and completeness. Implementation details that are not necessary for this goal are omitted.

A conflict-driven learning algorithm for logical satisfiability alternates model search and conflict analysis. Learning improves the precision of model search based on the information generated by conflict analysis.

Recall from Section 3.2.2, that checking if a formula φ is satisfiable is an instance of the bottom-everywhere problem over the following structure.

$$(\mathcal{P}(\mathcal{S}), \subseteq, \cap, \cup, \mathit{ded}_\varphi, \mathit{abd}_\varphi, \mathit{cded}_\varphi, \mathit{cabd}_\varphi)$$

This section generalizes the two phases of CDCL, model search and conflict analysis, to the bottom-everywhere problem.

Model search corresponds to a search for an abstract counterwitness (since if the instance is satisfiable, then ded_φ is *not* bottom-everywhere) and the conflict analysis phase corresponds to a witness search. It aims to find an element (in the form of a generalized conflict) which shows that no solution exists.

Section 5.1.1 and Section 5.1.2 describe abstract procedures that perform search for a counterwitnesses and witnesses for bottom-everywhere. Section 5.1.3 gives a formal definition of learning and shows how counterwitness search, witness search and learning can be combined in a conflict-driven learning algorithm.

5.1.1 Abstract Counterwitness Search

Recall from Section 4.4 that model search uses an abstract deduction transformer to generate a downwards iteration sequence with extrapolation. In terms of the bottom-everywhere problem, a satisfying assignment is a counterwitness. We may therefore view model search as an abstract counterwitness search.

The procedure in Algorithm 7 gives sound but incomplete answers to the bottom-everywhere problem over $(B, f, \tilde{f}, cf, \tilde{cf})$. In place of partial assignments, the algorithm uses an overapproximation O of B. BCP is replaced by an overapproximation $\mathit{of} : O \to O$ of the concrete transformer f, a downwards extrapolation operator $\nabla_\downarrow : O \times O \to O$ is used in place of a decision heuristic, and the check for satisfiability is replaced by a γ-completeness check. When a fixed point is reached, downwards extrapolation may be applied to increase precision, therefore the exit condition of the loop is not the classic fixed point condition $o \sqsubseteq \mathit{of}(o)$. When instantiating the algorithm as propositional model search, the γ-completeness check can, for example, take the form of checking whether at least one literal in each clause is satisfied by the current partial assignment (see Proposition 4.4.1).

Algorithm 7: Not bottom-everywhere algorithm.

```
in     : $\mathit{of} : O \to O$ – overapprox. of $f$ in bottom-everywhere algebra
         $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, \tilde{cf})$,
in-out: $o \in O$ – abstract element
out    : $(\mathsf{not}\ \bot, o)$ or $(\mathsf{unknown}, o)$ for some $o \in O$
Abstract-non-$\bot$($\mathit{of}$, $o$)
    repeat
        $o' \leftarrow o$;
        $o \leftarrow o \nabla_\downarrow \mathit{of}(o)$;
    until $o = o' \vee \gamma_O(o) = \bot$;
    if $\gamma(o) \neq \bot$ and $\mathit{of}$ is $\gamma$-complete at $o$ then
        return $(\mathsf{not}\ \bot, o)$ ;
    else
        return $(\mathsf{unknown}, o')$;
    end
```

Proposition 5.1.1 (Soundness). *If* `Abstract-non-⊥(`of`, ⊤)` *returns* not $\bot$*, then* f *is not bottom everywhere.*

Proof. Assume that the algorithm returns (not $\bot, o$). Then o is a fixed point of the function $\lambda x.of(x) \sqcap x$, which soundly approximates f. Further, $\gamma(o)$ is not equal to $\bot$ since otherwise the algorithm would have returned $\bot$. Together, these facts imply that o is an abstract counterwitness. It follows from Theorem 3.2.22 that f is not bottom everywhere. □

Algorithm 7 provides a sound but incomplete way to determine satisfiability and returns unknown if satisfiability cannot be determined. Unknown results may come about in two ways. Either model search has failed to make an element sufficiently precise for determining satisfiability using the γ-completeness check, or the resulting element is a conflict, that is $\gamma(o) = \bot$. In addition, the main loop of the algorithm may never terminate. Termination can be ensured by choosing a downwards extrapolation that is also a widening.

The iteration sequence is generated by a run of Algorithm 7 is unusual in terms of classical abstract interpretation. The procedure works in the context of an overapproximate abstraction, but ultimately, aims to compute a sound underapproximation of f that is precise enough to show satisfiability.

5.1.2 Abstract Witness Search

We now present an algorithm that generalizes conflict analysis. We have shown in Section 4.4 that conflict analysis in a SAT solver computes an underapproximate abduction transformer using upwards interpolation. Propositional conflict analysis is an instance of the abstract witness search shown in in Algorithm 8. The algorithm computes classic upwards iteration with interpolation, using a binary interpolation operator $\Delta_\uparrow : U \times U \to U$.

Algorithm 8: Top-everywhere algorithm.

```
in     : $u\tilde{f} : U \to U$ – underapprox. of $\tilde{f}$ in bottom-everywhere algebra
         $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, c\tilde{f})$
in-out: $u \in U$ – abstract element with $f \circ \gamma(u) = \bot$.
out    : $\top$ or unknown
Abstract-⊤($u\tilde{f}$, $u$)
    repeat
        $u' \leftarrow u$;
        $u \leftarrow u \,\Delta_\uparrow\, u\tilde{f}(u)$;
    until $u = u'$;
    if $\gamma(u) = \top$ then
        return $(\top, o)$ ;
    else
        return (unknown, $o$);
    end
```

Proposition 5.1.2 (Soundness). *If* `Abstract-⊤(`$u\tilde{f}$`, `u`)` *returns* $\top$ *and* $f \circ \gamma(u) = \bot$*, then* $\tilde{f}$ *top everywhere.*

Proof. We have that $f \circ \gamma(u) = \bot$, for the argument u of the call. By the properties of the Galois connection $(f, \tilde{f})$ (Proposition 3.2.4), we then have that $\gamma(u) \subseteq \tilde{f}(\bot)$. Since $\tilde{f}$ is an upper closure, we have that $\tilde{f} \circ \gamma(u) = \tilde{f}(\bot)$. Algorithm 8 computes an upwards iteration sequence with interpolation. We have from Theorem 2.3.12 that the result u' is an underapproximation of $\tilde{f} \circ \gamma(u)$. Therefore, $\tilde{f} \circ \gamma(u) = \tilde{f}(\bot) \supseteq \gamma(u') = \top$, i.e., $\tilde{f}(\bot) = \top$. It follows that $\tilde{f}$ is top everywhere and, consequently, f is bottom everywhere. □

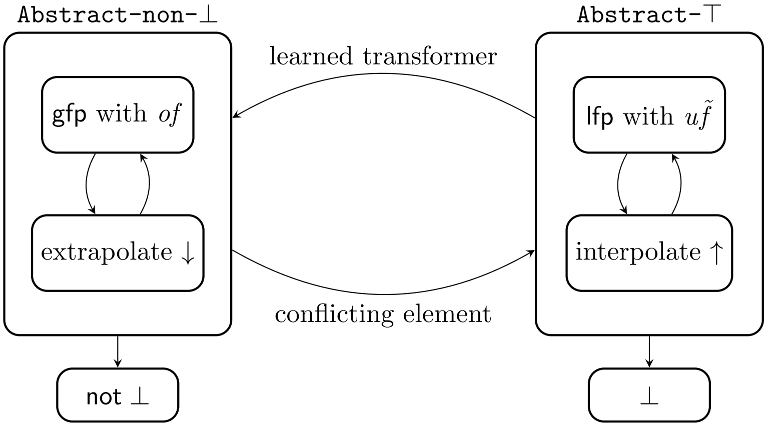


Figure 5.1: Abstract conflict driven learning.

The algorithm is sound but incomplete for determining if $\tilde{f}$ is top everywhere. It returns unknown if it cannot be established that $\tilde{f}$ is top everywhere. Additionally, it may fail to terminate. The latter can be avoided if the interpolation operator $\Delta_{\uparrow}$ is a narrowing.

5.1.3 Conflict-Driven Learning

The CDCL algorithm combines Abstract-non-⊥ and Abstract-⊤ as summarized in Figure 5.1. Rather than return unknown from Abstract-non-⊥, information from the fixed point computation drives Abstract-⊤. If the procedure Abstract-⊤ produces inconclusive results, then Abstract-non-⊥ can still "learn" information about the conflict for the greatest fixed point computation. This section makes this combination precise.

The *Abstract Conflict Driven Learning* procedure (ACDL) is shown in Algorithm 11. The procedures alternates runs of Abstract-non-⊥ and Abstract-⊤. Communication between the two procedures is achieved using two functions, $ou\tilde{f}$ and learn. Conflicting elements are transferred from Abstract-non-⊥ to Abstract-⊤ using the function $ou\tilde{f} : O \to U$. We require that this function soundly underapproximates $\tilde{f}$, i.e., that $\tilde{f} \circ \gamma_O \subseteq \gamma_U \circ ou\tilde{f}$. A natural choice for this transformer is to compute the composition $\alpha_U \circ \gamma_O$, which maps an abstract element in O to its best underappproximation in U. In the other direction, a transformer is learned, as discussed below.

Best Learning Transformer

If we learn that f maps the concretization of an element $u \in U$ to bottom, we can use this information to assist in future counterwitness search steps. Since u does not represent a counterwitnesses to f being bottom everywhere, we may subtract u from any element $o \in O$ during counterwitness search.

The *best learning transformer* below takes elements u and o and subtracts u from o:

$$\mathsf{learn}_. : U \to (O \to O) \qquad\qquad \mathsf{learn}_u(o) \mathrel{\hat{=}} \alpha_O(\gamma_O(o) \cap \neg\gamma_U(u))$$

A *sound learning transformer* is a function $g : U \to (O \to O)$ such that for all $u \in U$, $g(u)$ is a transformer and $g(u) \sqsupseteq \mathsf{learn}_u$.

The above transformer uses only information contained within the conflicting element u. More precision may be obtained by taking into account the concrete transformer f that is approximated. The *best f-learning transformer* is the following function.

$$\mathsf{flearn}_. : U \to (O \to O) \qquad\qquad \mathsf{flearn}_u(o) \mathrel{\hat{=}} \alpha_O(f(\gamma_O(o) \cap \neg\gamma_U(u)))$$

Algorithm 9: Abstract conflict-driven learning.

```
in     : $u\tilde{f} : U \to U$ – underapprox. of $\tilde{f}$
         $\mathsf{learn} : U \to (O \times O)$ – learning transformer
         $ou\tilde{f} : O \to U$ – underapprox. of $\tilde{f}$, i.e., $\tilde{f} \circ \gamma_O \subseteq \gamma_U \circ ou\tilde{f}$.
in-out: $of : O \to O$ – overapprox. of $f$,
out    : $\bot$, $\mathsf{not}\ \bot$ or $\mathsf{unknown}$

ACDL($of$, $u\tilde{f}$, $ou\tilde{f}$, $\mathsf{learn}$)
    loop
        $(s, o) \leftarrow$ Abstract-non-$\bot$($of$, $\top$);
        if $s = \mathsf{not}\ \bot$ then return $\mathsf{not}\ \bot$;
        if $\gamma_O(o) \neq \emptyset$ then return $\mathsf{unknown}$;
        $u \leftarrow ou\tilde{f}(o)$;
        $(s, u) \leftarrow$ Abstract-$\top$($u\tilde{f}$, $u$);
        if $s = \top$ then return $\bot$;
        $of \leftarrow of \sqcap \mathsf{learn}(u)$;
    end
```

A *sound f-learning transformer* is a function $g : U \to (O \to O)$ such that for all $u \in U$, $f(u)$ is a transformer and $f(u) \sqsupseteq \mathsf{flearn}_u$. Note that every learning transformer is an f-learning transformer, but the converse does not hold.

The next theorem is the core of the soundness argument of Algorithm 11. Recall the *parametric fixed point* functions that map an element x of L to the least fixed point above x and greatest fixed point below x, respectively.

$$\mathsf{plfp}(f) \mathrel{\hat{=}} \lambda x.(\mathsf{lfp}\, Y.\ f(Y) \sqcup x) \qquad \mathsf{pgfp}(f) \mathrel{\hat{=}} \lambda x.(\mathsf{gfp}\, Y.\ f(Y) \sqcap x)$$

Theorem 5.1.3. *Let of be a sound overapproximation of a completely additive, reductive transformer f, let $u\tilde{f}$ be a sound underapproximation of the De Morgan dual $\tilde{f}$ of f, and let* flearn *be a sound f-learning transformer. If* $\mathsf{pgfp}(of)(o)$ *represents concrete $\bot$, then the transformer*

$$of \sqcap \mathsf{flearn}_u, \ \textit{where } u \textit{ is } \mathsf{plfp}(u\tilde{f})(\alpha_U(\gamma_O(o)))$$

is a sound overapproximation of f.

Proof. Let $o \in O$ be an element such that $\mathsf{pgfp}(of(o))$ represents concrete $\bot$, that is, $\gamma_O \circ \mathsf{pgfp} \circ of(o) = \bot$. Then we have that $\gamma_O \circ \mathsf{pgfp} \circ of(o) \supseteq f(\gamma(o))$, i.e., $f(\gamma(o)) = \bot$. By the Galois connection, we get that $\gamma(o) \subseteq \tilde{f}(\bot)$. Therefore, $\alpha_U(\gamma_O(o))$ soundly underapproximates $\tilde{f}(\bot)$. By basic soundness in abstract interpretation, the element $u = \mathsf{plfp}(u\tilde{f})(\alpha_U(\gamma_O(o)))$ also soundly underapproximates $\tilde{f}(\bot)$, which may be expressed as $f(\gamma_U(u)) = \bot$.

Now consider an element $o \in O$. Then, by Proposition 3.2.1, $f(\gamma_O(o))$ may be expressed as $\gamma_O(o) \cap f(\top)$. We get the following.

$$\begin{aligned}
f \circ \gamma_O(o) &= f(f \circ \gamma_O(o)) \\
&= f(\gamma_O(o) \cap f(\top)) \\
&= f(\gamma_O(o) \cap (f(\gamma_U(u) \cup \neg\gamma_U(u)))) \\
&= f(\gamma_O(o) \cap (f(\gamma_U(u)) \cup f(\neg\gamma_U(u)))) \\
&= f(\gamma_O(o) \cap f(\gamma_U(u))) \cup f(\gamma_O(o) \cap f(\neg\gamma_U(u))) \\
&= f(\bot) \cup f(\gamma_O(o) \cap f(\neg\gamma_U(u))) \\
&\subseteq f(\gamma_O(o) \cap \neg\gamma_U(u))
\end{aligned}$$

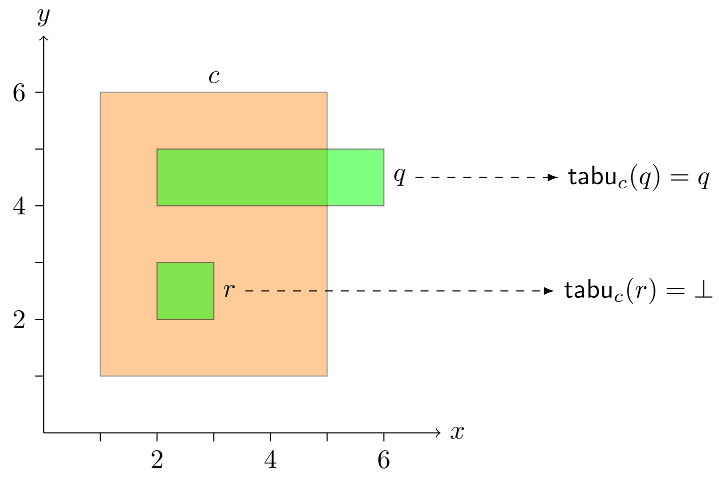


Figure 5.2: The rule tabu_c on the lattice of interval environments *Vars* → *Itv*.

$$\begin{aligned} &\subseteq \gamma_O(\alpha_O(f(\gamma_O(o) \cap \neg\gamma_U(u)))) \\ &= \gamma_O \circ \mathsf{flearn}_u(o) \\ \implies f \circ \gamma_O(o) &\subseteq \gamma_O \circ \mathsf{flearn}_u(o) \end{aligned}$$

The above proves that flearn_u soundly overapproximates f. The result follows. □

Theorem 5.1.3 strictly generalizes learning from SAT solvers to the bottom-everywhere problem.

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

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

The tabu rule checks whether search has entered a region that is known to map to $\bot$. We illustrate the tabu rule in Figure 5.2 on the lattice of interval environments, which is the pointwise lifting of *Itv* to the lattice *Vars* → *Itv*.

The `ACDL` procedure is sound by construction.

Theorem 5.1.4 (Soundness)**.** *If* `ACDL` *returns* not $\bot$*, then f is not bottom everywhere. If* `ACDL` *returns $\bot$, then f is bottom everywhere.*

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

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

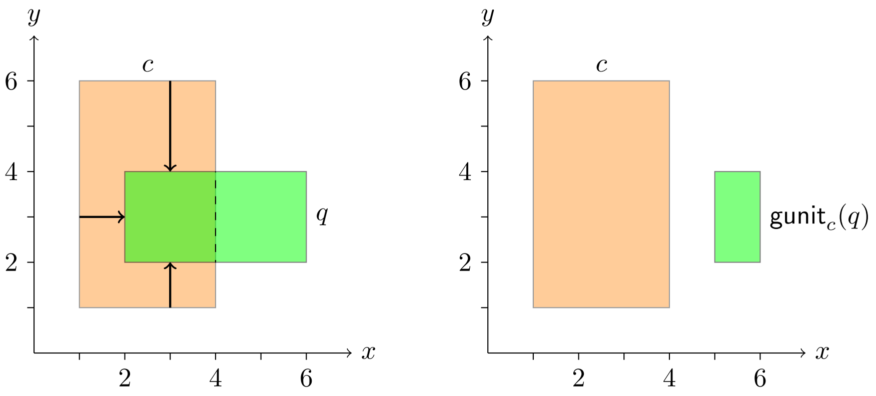


Figure 5.3: Generalized unit rule gunit_c for intervals on $\mathbb{Z}$.

5.2 Clause Learning in Lattices

The previous section showed that learning is not restricted to propositional logic. Clause learning is a form of learning in which abstract transformers are implicitly represented by clauses. Since model search in SAT solvers is driven by the unit rule, clause learning can be viewed as learning unit rule transformers. We present a generalized unit rule that lifts clause learning to richer lattices. The generalization yields Abstract Conflict Driven Clause Learning (ACDCL), a strict generalization of the CDCL algorithm in SAT solvers. We demonstrated in Section 4.4 that CDCL solvers operate on the domain *PAsgs* and its downset completion $\mathcal{D}(PAsgs)$. ACDCL operates on an overapproximation A and its downset completion $\mathcal{D}(A)$, and is a variant of the ACDL procedure from the previous section.

To understand the lattice-theoretic essence of clause learning, it is useful to compare the unit rule with tabu learning. Recall that the tabu rule for a conflict element u reports $\bot$ when the argument is contained within u and does nothing otherwise. Consider a clause $C = p \vee \neg q$ and the partial assignment $\rho = \langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle$, which contains no satisfying assignment to C. Consider ρ' strictly greater than ρ. We have that $\mathsf{tabu}_\rho(\rho') = \rho'$. However, if ρ' is the partial assignment $\langle p{:}\mathsf{f}\rangle$, by unit rule application we have $unit_C(\rho') = \langle p{:}\mathsf{f}, q{:}\mathsf{f}\rangle$. The tabu rule only drives search away from elements where f is bottom. In contrast, the unit rule, when applied to an element ρ' on which f is "almost bottom" will drive the search away from the part of ρ' that leads to bottom.

Example 5.2.1. *We illustrate a generalized unit rule for the abstract domain of interval environments. Assume an abstract transformer of is $\bot$ on the interval c in Figure 5.3. We design a generalized unit rule that maps the shaded interval q in the figure to the interval* $\mathsf{gunit}_c(q)$*, which drives the search to the portion of q not known to lead to $\bot$.*

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

We highlight similarities between the propositional unit rule and the interval unit rule in Example 5.2.1. Every interval is the intersection of one-way infinite intervals, just as a partial assignment is the conjunction of literals. One-way infinite intervals, like propositional literals, have complements. The complement of a partial assignment is a clause, and the complement of an interval can be represented as the disjunction of one-way infinite intervals. The rest of this section lifts the unit rule to new domains.

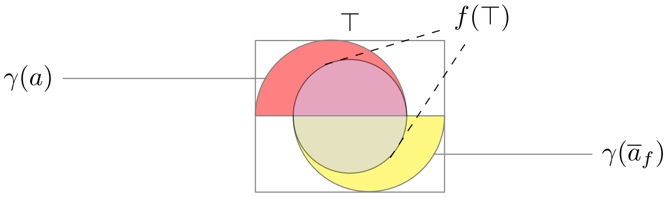


Figure 5.4: Precise f-complement $\overline{a}_f$ of element a.

5.2.1 Complementable Decompositions

We now introduce material that is necessary to generalize the unit rule. Consider a clause $C = p \vee \neg q \vee r$ and consider a partial assignment $\rho = \langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle$ that occurs during model search. An implementation of the unit rule would first check whether the partial assignment ρ "nearly" contradicts the clause C, that is, whether all but one literals are contradicted by truth assignments. In the above example, this is the case, since both the literal p and the literal $\neg q$ are contradicted. Next, the remaining literal is asserted by setting r to $\mathtt{t}$.

If C was a learned clause, it was generated from a conflicting partial assignment $c = \langle p{:}\mathsf{f}, q{:}\mathsf{t}, r{:}\mathsf{f}\rangle$. When viewed in terms of this assignment, the unit rule first checks whether all but one assignment in c are asserted in ρ, and therefore ρ is "nearly conflicting". Next, it forces the remaining element of c to be contradicted to drive ρ away from the known conflict.

In this section, we discuss how special kinds of decompositions may be used to generalize the "nearly conflicting" criterion and how restricted complementation may be used to drive away an abstract element from a region that is known to be conflicting.

The lattice of partial assignments does not admit precise complementation. Consider for example the partial assignment $\langle p{:}\mathsf{t}, q{:}\mathsf{f}\rangle$. This partial assignment represents exactly the singleton set of assignments $\{p \mapsto \mathtt{t}, q \mapsto \mathtt{f}\}$. Its complement $\gamma(\rho) = \mathcal{S}_P \setminus \{p \mapsto \mathtt{t}, q \mapsto \mathtt{f}\}$ has no precise representation as a partial assignment. The following definition fixes a precise notion of complementation in an abstraction.

Definition 5.2.1 (Precise Complement)**.** Let A be a lattice and a be an abstract element in a. A *precise complement* of a, denoted $\overline{a}$ is an element of A such that $\gamma(\overline{a}) = \neg\gamma(a)$.

A precise form of complementation as above allows us to perform case analysis in the abstraction. Note that in a case analysis, it is not necessary that all cases are covered as long as we know that the cases that are not covered do not describe anything of interest. With this in mind, we can extend the scope of our notion of complementation and give the following definition, which is illustrated in Figure 5.4.

Definition 5.2.2 (Precise Complementation Relative to f)**.** Consider the bottom-everywhere algebra $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, c\tilde{f})$ and an overapproximation A of B. A *precise complement* of a relative to f (or simply f-complement) denoted $\overline{a}_f$ is an element of A such that $f \circ \gamma(\overline{a}_f) = f(\neg\gamma(a))$.

We now define a class of lattices that have complementation properties similar to those of the lattice of partial assignments. Not all partial assignments have a precise complement. Only singleton assignments of the form $\langle x{:}\mathsf{t}\rangle$ and $\langle x{:}\mathsf{f}\rangle$ precisely complement each other. The existence of these elements allows for a simple form of case analysis, where instead of applying deduction to a partial assignment ρ, we can apply it twice, once to $\rho \sqcap \langle x{:}\mathsf{t}\rangle$ and once to $\rho \sqcap \langle x{:}\mathsf{f}\rangle$ to increase precision.

The following lattice properties generalize this notion of complementation to other lattices. Recall that a *meet irreducible* is an element that cannot be expressed as the meet of two other elements and that a *meet-factorization* is a representation of an element in terms of meet irreducibles. The meet irreducible elements of the partial assignments lattice are

	Interval Envs.	Octagons
Concrete Domain	$\mathcal{P}(V \to \mathbb{Z})$	$\mathcal{P}(V \to \mathbb{Z})$
Abstract Elements	$\langle x : [l, u], \ldots \rangle$	$\langle \pm x \pm y < c, \ldots \rangle$
Meet Irreducible	$\langle x \leq 4 \rangle$	$\langle x + y < 1 \rangle$
Complemented Irreducible	$\langle 5 \leq x \rangle$	$\langle -x - y < 0 \rangle$

	Equality	Array Abstr.
Concrete Domain	$\mathcal{P}(V \to \mathbb{Z})$	$\mathcal{P}(V \to (\mathbb{N} \to \mathbb{Z}))$
Abstract Elements	$\langle x = y, w \neq z, \ldots \rangle$	$\langle x \preceq \lambda i.\ c_x, y \not\preceq \lambda i.\ c_y, \ldots \rangle$
Meet Irreducible	$\langle x \neq y \rangle$	$\langle x \preceq \lambda i.\ 4 \rangle$
Complemented Irreducible	$\langle x = y \rangle$	$\langle x \not\preceq \lambda i.\ 4 \rangle$

	Set Abstr.	Control-Flow Abstr.
Concrete Domain	$\mathcal{P}((V \to D) \times (SV \to \mathcal{P}(D))$	$\mathcal{P}(\mathit{Traces})$
Abstract Elements	$\langle x \in S, y \notin R, \ldots \rangle$	$\langle l_i \to \texttt{else}, \ldots \rangle$
Meet Irreducible	$\langle x \in Q \rangle$	$\langle l_1 \to \texttt{else} \rangle$
Complemented Irreducible	$\langle x \notin Q \rangle$	$\langle l_1 \to \texttt{if} \rangle$

Figure 5.5: Examples of domains with complementable decompositions.

given by the singleton assignments of the form $\langle p{:}v \rangle$ where v is in $\{\mathsf{t}, \mathsf{f}\}$. These are also exactly the elements of the partial assignments lattice that admit precise complementation.

Definition 5.2.3 (Complementable Meet Irreducibles)**.** Consider the bottom-everywhere algebra $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, \tilde{cf})$ and an overapproximation A of B. A has *complementable meet irreducibles* if every element $m \in \mathbb{I}_\sqcap(A)$ has a precise complement $\overline{m} \in \mathbb{I}_\sqcap(A)$. A has *f-complementable meet irreducibles* if every element $m \in \mathbb{I}_\sqcap(A)$ has a precise f-complement $\overline{m}_f \in \mathbb{I}_\sqcap(A)$.

In a lattice with complementable meet irreducibles, every element can be viewed as representing a set of cases. Our generalization of clause learning operates on such a representation.

Definition 5.2.4 (Meet Factorization Operator)**.** A function $\mathit{mdc} : A \setminus \{\bot\} \to \mathcal{P}(\mathbb{I}_\sqcap(A))$ is a *meet factorization operator* if for all $a \in A \setminus \{\bot\}$, $\mathit{mdc}(a)$ is a meet factorization of a. The operator mdc is *finite* if for all $a \in A \setminus \{\bot\}$, the set $\mathit{mdc}(a)$ is finite.

Many abstract domains used in practice have both complementable meet irreducibles and admit finite meet factorizations. For example, partial assignments can be factored into sets of assignments to a single variable and intervals and octagons can be factored into sets of half-spaces. Examples of abstract domains with complementable decompositions are given in Figure 5.5. In the following, we focus on precise f-complementation which is a strictly weaker notion than precise complementation but is sufficient to allow liftings of CDCL. We could weaken the notion of complementation even further and work with lattices that admit overapproximate rather than precise complementation of meet irreducibles. Such lattices still admit sound instantiations of CDCL, but completeness arguments are more difficult in this setting.

5.2.2 Generalized Unit Rule

An abstraction A with (f-)complementable meet irreducibles admits a learning transformer $\mathsf{gunit} : \mathcal{D}(A) \times A \to A$ which generalizes clause learning and subsequent application of the unit rule. The unit rule is defined with respect to a clause. Clauses are derived from conflicting partial assignments via negation. We instead define the generalized unit rule with respect to elements that lead to $\bot$. This is only a difference of presentation; the negation of a clause is a partial assignment on which $unit_\varphi$ is bottom.

Definition 5.2.5 (Generalized Unit Rule)**.** Let A be a lattice with f-complementable meet irreducibles and a meet factorization function $mdc : A \to \mathcal{P}(\mathbb{I}_\sqcap(A))$. The generalized unit rule is defined as follows.

$$\mathsf{gunit}_c(a) \mathrel{\hat{=}} \begin{cases} \bot & \text{if for all } m \in mdc(c).\ a \sqsubseteq m \\ a \sqcap \overline{n}_f & \text{if } mdc(c) = M \cup \{n\}, \\ & a \not\sqsubseteq n \text{ and for all } m \in M.\ a \sqsubseteq m \\ a & \text{otherwise} \end{cases}$$

The transformer gunit can be lifted to a learning transformer $\mathsf{gunit}_. : U \times A \to A$, where $U = \mathcal{D}(A)$ by defining $\mathsf{gunit}_S \mathrel{\hat{=}} \bigsqcap_{c \in S} \mathsf{gunit}_c$.

Theorem 5.2.1 (Soundness of the Generalized Unit Rule)**.** *Let $S \in \mathcal{D}(A)$. The generalized unit rule transformer gunit_S for an element $S \in \mathcal{D}(A)$ is an f-learning transformer, i.e., $\mathsf{gunit}_S \sqsupseteq \mathsf{flearn}_S$.*

Proof.

$$\begin{aligned} \mathsf{flearn}_S(c) &= \alpha \circ f(\gamma(a) \cap \neg\gamma_U(S)) \\ &= \alpha \circ f(\gamma(a) \cap \neg \bigcup_{c \in S} \gamma(c)) \\ &= \alpha \circ f(\gamma(a) \cap \bigcap_{c \in S} \neg\gamma(c)) \\ &= \alpha \circ f(\bigcap_{c \in S} (\gamma(a) \cap \neg\gamma(c))) \\ &= \alpha \circ f(\bigcap_{c \in S} (\gamma(a) \cap \neg \bigcap_{m \in mdc(c)} \gamma(m))) \\ &= \alpha \circ f(\bigcap_{c \in S} (\gamma(a) \cap \bigcup_{m \in mdc(c)} \neg\gamma(m))) \\ &= \alpha \circ f(\bigcap_{c \in S} \bigcup_{m \in mdc(c)} \gamma(a) \cap \neg\gamma(m)) \\ &= \alpha(\bigcap_{c \in S} \bigcup_{m \in mdc(c)} f(\gamma(a) \cap \neg\gamma(m))) \end{aligned}$$

Using claim (i) $\bigcup_{m \in mdc(c)} f(\gamma(a) \cap \neg\gamma(m)) \subseteq \gamma \circ \mathsf{gunit}_c(a)$, which we will show momentarily.

$$\begin{aligned} &\sqsubseteq \alpha(\bigcap_{c \in S} \gamma \circ \mathsf{gunit}_c(a)) \\ &= \alpha \circ \gamma(\bigsqcap_{c \in S} \mathsf{gunit}_c(a)) \\ &= \alpha \circ \gamma(\mathsf{gunit}_S(a)) \\ &\sqsubseteq \mathsf{gunit}_S(a) \end{aligned}$$

We now show claim (i):

$$\bigcup_{m \in mdc(c)} f(\gamma(a) \cap \neg\gamma(m)) \subseteq \gamma \circ \mathsf{gunit}_c(a)$$

The case where $\mathsf{gunit}_c(a) = a$ and where $\mathsf{gunit}_c(a) = \bot$ are simple. We now show the remaining case, where $mdc(a) = M \cup \{n\}$ such that $a \not\sqsubseteq n$ and for all $m \in M$. $a \sqsubseteq m$. For all $m \in M$, since $a \sqsubseteq m$ we have that $a \sqcap \overline{m}_f = \bot$, and therefore also that $f(\gamma(a) \cap \neg\gamma(m)) = \gamma(\bot)$. We then have the following.

$$\begin{aligned}
\bigcup_{m \in mdc(c)} f(\gamma(a) \cap \neg\gamma(m)) &= [\bigcup_{m \in M} f(\gamma(a) \cap \neg\gamma(m))] \cup f(\gamma(a) \cap \neg\gamma(n)) \\
&= \gamma(\bot) \cup f(\gamma(a) \cap \neg\gamma(n)) \\
&= f \circ \gamma(a) \cap f(\neg\gamma(n)) \\
&= f \circ \gamma(a) \cap f \circ \gamma(\overline{n}_f) \\
&= f \circ \gamma(a \sqcap \overline{n}_f) \\
&\subseteq \gamma(a \sqcap \overline{n}_f) \\
&= \gamma \circ \mathsf{gunit}_c(a)
\end{aligned}$$

We have completed the proof that $\bigcup_{m \in mdc(c)} \gamma(a) \cap \neg\gamma(m) \subseteq \gamma \circ \mathsf{gunit}_c(a)$. □

The next section introduces ACDCL, which is an instantiation of ACDL that learns generalized unit rules.

5.3 Abstract Conflict Driven Clause Learning

This section introduces the ACDCL algorithm which generalizes CDCL to the bottom-everywhere problem and to lattices. We first recall the model search step and the overall algorithmic framework of CDCL in Section 5.3.1. Section 5.3.2 presents ACDCL as a lifting of CDCL to lattices and Section 5.3.3 provides completeness and termination arguments for a restricted class of instantiations of ACDCL.

5.3.1 The CDCL Algorithm

CDCL combines model search and conflict analysis using a *non-chronological* backtracking search. Upon encountering a conflict, the algorithm – unlike classic DPLL– does not simply flip a previously made decision, but instead may additionally undo some decisions that did not contribute to the conflict. Since the algorithm may skip several unexplored branches while doing so, this process is also called *backjumping*. Intuitively, backjumping improves efficiency by allowing the algorithm to recover from decisions that did not yield interesting results. The completeness argument relies on the so-called "asserting" property of learned clauses [118, 137], which ensures that after backjumping, the clause derived from the conflict drives the algorithm away from previously explored search regions.

In this section, we abstract away the details of conflict analysis by assuming that there exists a conflict analysis algorithm with this asserting property. This abstract view is sufficient to give completeness conditions for lattices. In the next section, we generalize the propositional conflict analysis algorithm.

Our descriptions of CDCL and ACDCL focus on clause learning. Clause learning has been empirically shown to be the component of CDCL that has the greatest impact on performance [106]. CDCL benefits from numerous further algorithmic and engineering advances [9], such as smart variable selection heuristics for decisions effective data structures like watched literals and algorithmic improvements such as restarts. Discussing all these improvements in a lattice based setting is beyond the scope of this dissertation. Some, for example restarts,

Algorithm 10: The propositional CDCL algorithm.

```
in     : set of propositional clauses $\Phi$
out    : SAT or UNSAT
CDCL($\Phi$)
    $\rho$: partial assignment;
    trail: sequence of propositional assignments;
    reasons: partial function from $P \cup \{\bot\}$ to $\Phi$;
    $\rho \leftarrow \top$; trail $\leftarrow \epsilon$; reasons $\leftarrow \emptyset$;
    loop
        if ModelSearch($\rho$, trail, reasons, $\Phi$) = SAT then
            return SAT;
        end
        $c \leftarrow$ Analyse(trail, reasons);
        $\Phi \leftarrow \Phi \cup \{c\}$;
        if $\neg$Backjump($\rho$, trail, $c$) then
            return UNSAT;
        end
    end
```

Algorithm 11: Propositional model search.

```
in     : $\Phi$ – set of propositional clauses
in-out: reasons – partial function from $P \cup \{\bot\}$ to $\Phi$
         $\rho$ – partial assignment
         trail – sequence of propositional assignments
out    : conflict or SAT
ModelSearch($\rho$, trail, reasons, $\Phi$)
    loop
        repeat
            // Boolean Constraint Propagation
            foreach $C \in \Phi$ do
                $q \leftarrow unit_C(\rho)$;
                if $q = \bot$ then
                    reasons[$\bot$] $\leftarrow C$;
                    return conflict;
                end
                if $q \sqsubset \rho \wedge q = \rho \sqcap \langle p{:}v \rangle$ then
                    $\rho \leftarrow \rho \sqcap \langle p{:}v \rangle\}$;
                    trail $\leftarrow$ trail $\cdot \langle p{:}v \rangle$;
                    reasons[$p$] $\leftarrow C$;
                end
            end
        until $\rho$ unchanged;
        if $\rho$ is a complete assignment then  return SAT;
        $\langle p{:}v \rangle \leftarrow$ decide($\rho$);
        $\rho \leftarrow \rho \sqcap \langle p{:}v \rangle$;
        trail $\leftarrow$ trail $\cdot \langle p{:}v \rangle$;
        reasons[$p$] $\leftarrow$ nil;
    end
```

lift to ACDCL in a trivial manner while others, such as variable selection heuristics, require domain-specific adaptations.

The CDCL algorithm is recalled in Algorithm 10. We use lattice notation to denote manipulations of partial assignments. The algorithm first calls model search. If a satisfying assignment is found, CDCL terminates with a SAT result. Otherwise, if a conflict is identified, conflict analysis is used to extract a general reason in the form of a learned clause. Conflict analysis is discussed in detail in the next section. The learned clause is added to the formula and the algorithm uses backjumping to reset its internal data structures to a state before the conflict was found. The model search algorithm is shown in Algorithm 11.

The CDCL algorithm uses two data structures to keep track of the state of the search. These include a *trail* and a *reasons* array. The trail is a sequence of singleton partial assignments of the form $\langle p{:}\mathsf{t}\rangle$ or $\langle p{:}\mathsf{f}\rangle$. The symbol ϵ in the algorithm denotes the empty trail. Whenever a singleton assignment of the form above is obtained via the unit rule or a decision, it is added to the back of the trail. The reasons array maps propositions to clauses. If the assignment to p was the result of the unit rule, the clause C is recorded as the reason for p. Additionally, when a conflict is derived, the reasons array maps the conflict element $\bot$ to a clause that contradicts the partial assignment. In the conflict analysis phase, the trail and reason array serve as a chronological record of search leading up to the conflict.

5.3.2 Lifting CDCL to Lattices

We now lift CDCL to lattices. The *Abstract Conflict Driven Clause Learning* (ACDCL) framework is shown in Algorithm 12.

ACDCL can be viewed as an instance of the conflict-driven learning framework from Section 5.1, although we deviate slightly from the algorithmic framework presented there. We provide specialized versions of the `Abstract-non-`$\bot$ and `Abstract-`$\top$ functions which store additional information about the search.

We now generalize aspects of CDCL to lattices. In ACDCL, the partial assignments lattice is replaced by an overapproximate lattice O. Figure 5.6 shows the abstract requirements for instantiating ACDCL. Requirement (i) states that the lattice O has $(f-)$complementable meet-irreducibles. This is a prerequisite for learning generalized unit rules (see Section 5.2). Requirement (ii) uses a downwards extrapolation for the role that a decision heuristic plays in CDCL. In place of a set of input clauses, ACDCL uses a set of sound deduction transformers F, which corresponds to requirement (iii). Requirements (iv), (v) and (vi) are specific to AFIRSTUIP conflict analysis and will be discussed in the next section.

The trail data structure translates directly from CDCL. Recall that there, the trail stores a sequence of singleton assignments, which correspond to the meet-irreducible elements of the partial assignment lattice. Consequently, the trail in ACDCL is a sequence of meet irreducibles. The reasons array is slightly different than in the propositional algorithm. In CDCL, reasons is a mapping from propositions to clauses, since every proposition may only be assigned to a truth value once during the run of the procedure. In ACDCL, reasons maps trail indices to the transformers in F that were used to derive elements at those indices.

Model search is replaced by the `ACDCL-non-`$\bot$ algorithm shown in Algorithm 13, an implementation of `Abstract-non-`$\bot$ that collects auxiliary information in the trail and reasons data structures. Recall that downwards extrapolation replaces decision making. The results of transformer application and extrapolation are decomposed into meet irreducibles using a meet factorization operator *mdc*. Meet irreducibles that refine the current element are deposited on the trail, and the transformer used to obtain them is stored as their reason. If a meet irreducible is the result of extrapolation, nil is stored as a reason.

If a conflict is encountered where the current element a represents the empty set, then `ACDCL-`$\top$ is called to generalize the conflict. A unit rule gunit_c is learned from the result of generalization and the function `Backjump` resets the algorithm to an earlier, non-contradictory state. The most recently learned unit rule is then immediately applied. The restrictions on the backjumping ensure that no conflict is encountered at this point.

(i) An overapproximate lattice with f-complementable meet irreducibles

$$(A, \sqsubseteq, \sqcap, \sqcup),$$

(ii) a downwards extrapolation

$$\nabla_{\downarrow} : A \times A \to A,$$

(iii) a set of overapproximations of f

$$F \subseteq A \to A,$$

(iv) for each $\mathit{of} \in F$ and $a \in A$, an inflationary underapproximation

$$\mathit{uf}_{\mathit{of},a} : \mathcal{D}(A) \to \mathcal{D}(A)$$

of the transformer $\tilde{f}_a \mathrel{\hat{=}} c \mapsto \tilde{f}(c \cup \gamma(a))$,

(v) an ACDCL (upwards) interpolation

$$\Delta_{\uparrow} : \mathcal{D}(A) \times \mathcal{D}(A) \to A,$$

(vi) a finite meet factorization $\mathit{mdc} : A \to \mathcal{P}(A)$ such that the following holds for all sets $M \in \mathbb{I}_{\sqcap}(A)$ and elements $a \in A$.

$$\text{If } \bigsqcap M \sqsubseteq a \text{ then } \forall q \in \mathit{mdc}(a).\ \exists r \in M.\ r \sqsubseteq q.$$

Figure 5.6: Requirements for applying ACDCL to the bottom-everywhere problem for the algebra $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, c\tilde{f})$.

Algorithm 12: The ACDCL algorithm.

```
in  : $F \subseteq A \to A$ – a set of overapprox. transformers of $f$
out : $\bot$, not $\bot$ or unknown

ACDCL($F$)
    $a$: element of $A$;
    trail: sequence of meet irreducibles;
    reasons: partial function from trail indices and $\bot$ to $F$;
    $a \leftarrow \top$; trail $\leftarrow \epsilon$; reasons $\leftarrow \emptyset$;
    loop
        $s \leftarrow$ ACDCL-non-$\bot$($a$, trail, reasons, $F$);
        if $s =$ not $\bot \vee s =$ unknown then return $s$;
        // $s =$ conflict
        $c \leftarrow$ ACDCL-$\top$(trail, reasons);
        $F \leftarrow F \cup \{\mathsf{gunit}_c\}$;
        if $\neg$Backjump($a$, trail, $\bigsqcap F$) then return $\bot$;
        putOnTrail(trail, $\mathsf{gunit}_c(a)$, $\mathsf{gunit}_c$);
    end

Backjump($a$, trail, $of$)
    if there is a prefix trail$'$ of trail s.t. $\gamma \circ of(\bigsqcap \mathsf{trail}) \neq \bot$ then
        trail $\leftarrow$ pick a trail$'$ as above;
        $a \leftarrow \bigsqcap$trail;
        return t;
    else
        return f;
    end

putOnTrail(trail, $q$, $r$)
    $Q \leftarrow mdc(q)$;
    foreach $m \in Q$ do
        if $a \not\sqsubseteq m$ then
            if $\gamma(a \sqcap m) = \bot$ then
                reasons[$\bot$] $\leftarrow of$;
                return conflict;
            else
                reasons[|trail|] $\leftarrow of$;
            end
            $a \leftarrow a \sqcap m$;
            trail $\leftarrow$ trail $\cdot\, m$;
        end
    end
```

Algorithm 13: ACDCL bottom-everywhere counterwitness search.

```
in     : $F \subseteq A \to A$ – a set of overapprox. transformers of $f$
in-out: $a$ – element of $A$
        trail – sequence of meet irreducibles
        reasons – partial function from trail indices and $\bot$ to $F$
out    : conflict, not $\bot$ or unknown
ACDCL-non-$\bot$($a$, trail, reasons, $F$)
    loop
        repeat
            // Downwards iteration sequence with extrapolation
            $a' \leftarrow a$;
            foreach $of \in F$ do
                $q \leftarrow of(a)$;
                if putOnTrail(trail, $q$, $of$) = conflict then
                    return conflict;
                end
            end
            $d \leftarrow a' \nabla_{\downarrow} a$;
            if $\gamma(d) = \bot$ then return unknown;
            if putOnTrail(trail, $d$, nil) = conflict then
                return unknown;
            end
        until $a' = a$;
        if $\bigsqcap F$ is $\gamma$-complete at $a$ then return not $\bot$;
        else return unknown;
    end
```

For now, we leave the implementation of $\texttt{ACDCL-}\top$ open, but we make the following assumption.

Assumption 1. *In each iteration of the algorithm, the call to* $\texttt{ACDCL-}\top$ *computes an element* $c \in \mathcal{D}(A)$ *such that* $\gamma_{\mathcal{D}(A)}(c) \subseteq \tilde{f} \circ \gamma(a)$.

This assumption is sufficient to allow us to prove soundness of the overall algorithm.

Lemma 5.3.1. *If Assumption 1 holds, then in each iteration of the main loop of* `ACDCL`*, the set* F *is a set of sound overapproximations of* f.

Proof. We show the property by induction. We denote by F_i the value of the set variable F at the start of the ith iteration of the main loop. From the precondition of the algorithm, we have that F_1 is a set of sound overapproximations of f. Now assume that F_i is a set of sound overapproximations, we show that the same holds for $F_{i+1} = F \cup \{\mathsf{gunit}_c\}$ (if i is not the last iteration). It suffices to show that gunit_c soundly overapproximates f. Note that c is the result of calling $\texttt{Abstract-}\top$ from a conflicting element a. From Assumption 1, we get that $\gamma_{\mathcal{D}(A)}(c)$ soundly underapproximates $\tilde{f}(\gamma_{\mathcal{D}(A)}(\{a\}))$ which is equal to $\tilde{f}(\bot)$ since a is a conflicting element. We then have from Theorem 5.1.3 and Theorem 5.2.1 that gunit_c is a sound approximation of f, which completes the proof. □

Theorem 5.3.2 (Soundness)**.** *If* `ACDCL` *returns* $\mathsf{not}\ \bot$*, then* f *is not bottom everywhere. If Assumption 1 holds and* `ACDCL` *returns* $\bot$*, then* f *is bottom everywhere.*

Proof. Assume that `ACDCL` returns $\mathsf{not}\ \bot$. Then the final element a is γ-complete with respect to $\bigsqcap F$ and it holds that $\gamma(a) \neq \bot$. From Lemma 5.3.1 we have that $\bigsqcap F$ is a sound approximation of f. Therefore a is an abstract bottom-everywhere counterwitness. It follows that f is not bottom everywhere.

If, on the other hand, `ACDCL` returns $\bot$, then we have that $\gamma \circ \bigsqcap F(\top) = \bot$. Since $\bigsqcap F$ soundly approximates f, we have that $f(\top) = \bot$. Therefore, f is bottom everywhere. □

5.3.3 Termination and Completeness

We now discuss both lattice-theoretic and algorithmic conditions that are sufficient to establish completeness of an instantiation of ACDCL. Note that ACDCL may be applied to a variety of problems, including those that are undecidable. It is therefore difficult to provide completeness conditions that cover all instantiations. We focus on lifting the propositional argument to the finite lattice case.

ACDCL may be applied to problems on which it is not complete. In this case, it is necessary to choose between designing an algorithm that always terminates, but may return an undetermined result, and a procedure that may not terminate but always returns either $\bot$ or $\mathsf{not}\ \bot$.

We identify three obstructions to completeness.

(i) Non-convergence of the iteration sequences computed by $\texttt{ACDCL-}\top$ and $\texttt{ACDCL-non-}\bot$.

(ii) Convergence of the iteration sequence computed by $\texttt{ACDCL-}\top$ to an element that is not a counterwitness, i.e., that is neither conflicting nor allows a conclusive determination that f is not bottom everywhere.

(iii) Non-convergence of the `ACDCL` algorithm, i.e., non-termination of the main learning loop.

The first issue can be addressed by requiring the use of a downwards widening operator $\nabla_{\downarrow}$ in $\texttt{ACDCL-non-}\bot$ instead of an arbitrary extrapolation, respectively, an upwards narrowing $\Delta_{\uparrow}$ in $\texttt{ACDCL-}\top$ instead of an arbitrary interpolation, which ensure convergence.

The second issue may be addressed by placing certain precision requirements on the downwards extrapolation function used by the algorithm. Informally, any abstract fixed point a of of that does not satisfy the γ-completeness criterion used to check for counterwitnesses must be refined further by a call to the downwards extrapolation operator. In CDCL, this would correspond to the requirement that a decision heuristic must keep refining partial assignments until at least one literal in each clause is satisfied. We introduce a notion of precision for extrapolation operators that formally characterizes the necessary behavior.

Definition 5.3.1 (Precise Extrapolation)**.** The binary downwards extrapolation $\nabla_\downarrow : A \times A \to A$ is *precise* with respect to of if the following two properties hold for any $a, b \in A$.

(i) Whenever $a \nabla_\downarrow a = a$ then of is γ-complete at a.

(ii) Whenever $\gamma(a \nabla_\downarrow b) = \bot$ then $\gamma(a \sqcap b) = \bot$.

Theorem 5.3.3 (Relative Completeness)**.** *If the downwards extrapolation* $\nabla_\downarrow : A \times A \to A$ *is precise with respect to* $\bigsqcap F$ *then* `ACDCL(`F`)` *never returns* `unknown`*.*

Proof. We prove by contradiction. Assume that `ACDCL` returns `unknown`, then `unknown` was returned by the call to `ACDCL-non-`$\bot$. There are two possible cases: (i) The result `unknown` was returned after applying downwards extrapolation. (ii) The result `unknown` was returned after the downwards iteration sequence converged, and $\bigsqcap F$ was not γ-complete at the resulting element.

In case (i), it held that $\gamma(a \nabla_\downarrow a') = \bot$. Since $\nabla_\downarrow$ is precise, it follows that $\gamma(a \sqcap a') = \bot$. But then the call to `putOnTrail(`$\mathsf{trail}, q, \mathit{of}$`)` must have returned `conflict` for some $\mathit{of} \in F$. Therefore, `ACDCL-non-`$\bot$ must have returned `conflict`, which contradicts the assumption that it returned `unknown`.

In case (ii) then $a \nabla_\downarrow a = a$. Since $\nabla_\downarrow$ is precise with respect to the original set of transformers F_i it must then hold that $\bigsqcap F_i$ is γ-complete at a with respect to the concrete transformer f. We denote by F the value of the set variable when the algorithm returns. We have that $F_i \supseteq F$, therefore $\bigsqcap F \sqsubseteq \bigsqcap F_i$. From Lemma 5.3.1, we have that $\bigsqcap F$ soundly overapproximates f. Therefore, since $\bigsqcap F_i$ is γ-complete at a with respect to f, the same holds for $\bigsqcap F$. But then the algorithm would have returned `not` $\bot$, which contradicts the assumption that `unknown` was returned. We have shown that the assumption that `unknown` is returned leads to a contradiction. Therefore `ACDCL(`F`)` never returns `unknown`. □

The third source incompleteness, i.e., non-convergence of the ACDCL algorithm is more complex since it depends on the precise interplay of a number of parameters, including the lattice, transformers, learning and backjumping. Even in the propositional case, care must be taken to ensure that the algorithm does not repeatedly explore the same regions of the search space. This is accomplished via the mechanism of learning *asserting clauses* [118, 137], that is, clauses that upon backjumping help drive the search towards new areas of the search space.

We give an abstract definition of this behavior which we call *asserting backjumps*, but first we require some notation to discuss executions of ACDCL at the required level of abstraction. A *state* in ACDCL is a mapping from program variables to values. An *execution* of `ACDCL` is given by a well-formed trace, that is, a sequence of states which respect the semantics of the algorithm. For some fixed execution, we use the following notation to denote the value of program variables in the ith iteration of the main loop:

v_i	value of program variable v at the start of the main loop
a_i^c	conflicting value of a immediately before the call to `Backjump`
a_i^b	value of a immediately after the call to `Backjump`

Definition 5.3.2 (Asserting Backjumps)**.** An execution of `ACDCL` has *asserting backjumps* if for all iterations $i > 1$ it holds that $a_i \not\sqsupseteq a_{i-1}^c$.

After an asserting backjump, the learned abstract transformer can be immediately used to refine the current element. Informally, the transformer application pushes the search away from the conflict that was found most recently. In this dissertation, we do not discuss how asserting backjumps can be achieved. The required techniques lift from the propositional case [118, 137].

We now show that `ACDCL` with asserting backjumps will not get stuck indefinitely in some part of the search space. This forms the core of the termination argument.

Lemma 5.3.4. *Consider an infinite* `ACDCL` *execution with asserting backjumps over a finite lattice A. Then for any iteration i, there is an iteration $j > i$ such that $a_j^b \sqsupset a_i$.*

Proof. For a lattice element $a \in A$, we define the *height* of a as the longest path from a to the $\bot$ element. Fix an infinite execution. We prove the property by induction on the height of the element a_i. The induction hypothesis I_k is that for every element a_i of height $\leq k$, it holds that there is an iteration $j > i$ such that $a_j \sqsubset a_i$.

Consider the base case I_1, where a_i is an atom of the lattice. We show that $a_{i+1} \sqsupset a_i$. Note that since a_i is an atom, it must hold that $a_i^c = a_i$, that is, the call to `ACDCL-non-`$\bot$ will not modify the value of the variable a in iteration i. Since backjumping must reset the value of a to some earlier, non-conflicting value, it follows that $a_{i+1} \sqsupset a_i$, which concludes the base case.

For the induction step, assume that I_k holds. For a contradiction, assume that I_{k+1} is wrong. There is an element a_i of height $k+1$ such that there is no $j > i$ with $a_j \sqsupset a_i$. Since the run under consideration is infinite and since I_k holds, it follows that the element a_i is the target of a backjump infinitely often. Then there are iterations $i < j_1 < j_2$ such that $a_{j_1}^b = a_{j_2}^b = a_i$ and $a_{j_1}^c = a_{j_2}^c$. Due to asserting backjumps, we have that $a_{j_1+1} \not\sqsupseteq a_{j_1}^c$, where $a_{j_1+1} = \mathsf{gunit}_c(a_{j_1}^b)$ for some $c \in \mathcal{D}(A)$. In order to reach the same conflict in iteration j_2, there needs to be an intervening backjump to the element a_i. But every such backjump is followed by a call to `ACDCL-non-`$\bot$, during which gunit_c is applied to yield an element that is not greater than $a_{j_1}^c$. Therefore, the conflicting element $a_{j_1}^c$ is never encountered again during a call to `ACDCL-non-`$\bot$, which contradicts the initial assumption which requires that $a_{j_2}^c = a_{j_1}^c$. Therefore I_{k+1} must hold, which completes the inductive step of the proof. □

Theorem 5.3.5. *If an execution of* `ACDCL` *has* asserting backjumps *and the lattice A is finite, then* `ACDCL` *terminates.*

Proof. It's easy to see that every call to `ACDCL-`$\top$ and `ACDCL-non-`$\bot$ must terminate on a finite lattice. Therefore, it is sufficient to prove that the main loop in `ACDCL` always terminates. Note that the value of a at the beginning of the first iteration is $\top$. From Lemma 5.3.4, we get that on an infinite run, eventually there must be a backjump to an element greater than $\top$. This is impossible, hence there can be no infinite run. □

Theorem 5.3.6. *If all of the following conditions hold, then* `ACDCL(`F`)` *is a sound and complete method to decide bottom everywhere.*

(i) The lattice A is finite.

(ii) $\nabla_{\downarrow}$ is a precise downwards extrapolation with respect to $\bigsqcap F$.

(iii) `ACDCL-`$\top$ *is sound, i.e., Assumption 1 holds.*

(iv) Every execution has asserting backjumps.

Proof. From Theorem 5.3.5 we get that the call to `ACDCL(`F`)` terminates. Theorem 5.3.3 implies that, upon termination, either $\bot$ or `not` $\bot$ is returned. Theorem 5.3.2 shows that the result correctly indicates whether f is bottom everywhere. □

5.4 Abstract Conflict Analysis

In our discussion of the ACDCL framework so far, we have left open the algorithmic details of implementing the conflict analysis step ACDCL-$\top$ over lattices. Instead, we have relied on two interface assumptions for proving soundness and completeness: (i) the assumption that the call to ACDCL-$\top$ computes a sound underapproximation of $\tilde{f}$ (Assumption 1), (ii) the assumption that every backjump in the algorithm is asserting. This section will present an algorithmic framework that can be used to derive conflict analysis procedures that satisfy these assumptions.

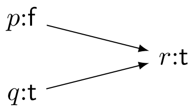
In CDCL, conflict analysis is performed via the FIRSTUIP algorithm [172]. The information collected during model search can be viewed as representing an *implication graph*, which records dependencies between deductions. The FIRSTUIP algorithm extracts conflict reasons by computing cuts on this graph. The cuts are constructed to ensure that the resulting clauses can be used for asserting backjumps.

In this section, we first review FIRSTUIP and then show how the algorithm can be lifted to lattices to yield *Abstract* FIRSTUIP *(* AFIRSTUIP*)*. Since arbitrary abstract domains have a more complex structure than the domain of partial assignments, a non-trivial lifting is required to obtain good conflict generalization. While the propositional algorithm is purely graph-based, working over other lattices creates the opportunity for semantic generalization via abductive transformers. In addition to providing a sound conflict analysis, the FIRSTUIP strategy finds asserting clauses.

The section is structured as follows. In Section 5.4.1, we first give an overview of the FIRSTUIP algorithm in propositional logic. Section 5.4.2 presents AFIRSTUIP and discusses soundness.

5.4.1 An Overview of First-UIP

We now discuss the FIRSTUIP conflict analysis strategy, which is the dominant conflict analysis method in modern SAT solvers. FIRSTUIP operates on a data structure called the *implication graph*, which represents a record of decisions and deductions made during model search. The nodes of this graph are singleton assignments of the form $\langle p{:}\mathsf{t}\rangle$ or $\langle p{:}\mathsf{f}\rangle$. The set of incoming edges to a node represents the necessary preconditions for unit rule application. Consider, for example, a clause $C = p \vee \neg q \vee r$. Applying the unit rule transformer $unit_C$ to the element $\langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle$ yields $\langle p{:}\mathsf{f}, q{:}\mathsf{t}, r{:}\mathsf{t}\rangle$. An implication graph represents this rule application as follows.

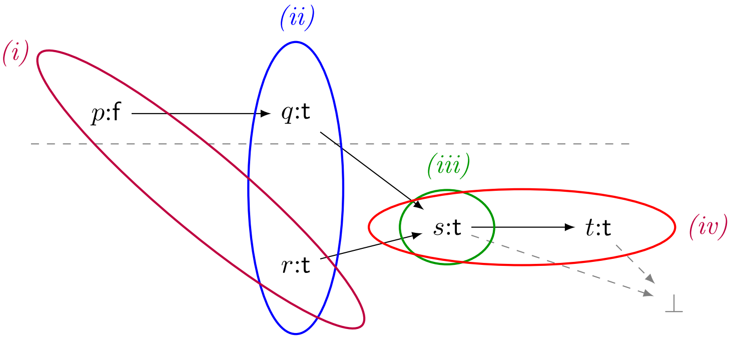


The edges $(p{:}\mathsf{f}, r{:}\mathsf{t})$ and $(q{:}\mathsf{t}, r{:}\mathsf{t})$ signify that the value of r is a necessary consequence of the values of p and q. Conversely, we can read the edges backwards as expressing that the values of p and q are a reason for the element $\langle r{:}\mathsf{t}\rangle$. The conflict analysis algorithm in propositional logic computes generalized conflict reasons by stepping backwards through an implication graph and identifying sets of nodes (called *cuts*) that are sufficient to derive a conflict.

Example 5.4.1. *Consider the following formula φ.*

$$\varphi = (p \vee q) \wedge (q \vee \neg r \vee s) \wedge (\neg s \vee t) \wedge (\neg s \vee \neg t)$$

The following implication graph represents a run of model search on φ.



Implication graphs are a useful way to explain FIRSTUIP, but are not explicitly constructed by the procedure. Rather, the implication graph is a means of interpreting the information encoded in the trail and reasons data structures: nodes represent elements of trail, edges can be extracted from the clauses in the reasons array. The FIRSTUIP algorithm steps backwards through the trail and thereby implicitly traverses the implication graph. While doing so, it keeps track of a set of nodes.

In the example, the nodes p:f and r:t are *decision elements*, that is, they were added to the graph as the result of a decision and hence have no incoming edges. All other nodes are the result of unit rule applications. The dashed gray line in the example above separates assignments made at different *decision levels.* The decision level of a node n, denoted $\mathsf{dl}(n)$ is the number of decision elements in the graph after the node n was added. The nodes above the gray line have decision level 1; nodes below the gray line have decision level 2. Since no node was added to the graph before the first decision was made, there is no node of decision level 0.

The partial assignment that results from the model search run encoded in the above graph is conflicting (denoted via a special $\perp$ node), since it contradicts the clause $\neg s \vee \neg t$. The final partial assignment ρ, shown below, is a trivial conflict reason.

$$\rho = \langle p{:}\mathsf{f}, q{:}\mathsf{t}, r{:}\mathsf{t}, s{:}\mathsf{t}, t{:}\mathsf{t}\rangle$$

FIRSTUIP determines a conflict reason by computing a *cut*, i.e., a set of nodes that separate all decision nodes from the conflict node. Four such cuts, denoted (i)-(iv), are depicted above, which describe the following conflict reasons.

$$\text{(i)}\ \langle p{:}\mathsf{f}, r{:}\mathsf{t}\rangle \quad \text{(ii)}\ \langle q{:}\mathsf{t}, r{:}\mathsf{t}\rangle \quad \text{(iii)}\ \langle s{:}\mathsf{t}\rangle \quad \text{(iv)}\ \langle s{:}\mathsf{t}, t{:}\mathsf{t}\rangle$$

Among the nodes with decision level 1, the two nodes $\langle s{:}\mathsf{t}\rangle$ and $\langle r{:}\mathsf{t}\rangle$ are special in that they are individually sufficient to produce the conflict. They are called *Unique Implication Points* (UIP). A UIP is a node n such that any path from the last decision to the error node $\perp$ must include n. The decision node of the deepest decision level is always trivially a UIP. For example, the node t:t is *not* a UIP because the path $r{:}\mathsf{t} \to s{:}\mathsf{t} \to \perp$ does not include it. The node s:t is called the first UIP since it is closest to the conflict.

Computing UIPs is helpful in order to generate asserting cuts. An *asserting cut* is a cut that includes exactly one node from the deepest decision level. Asserting cuts yield clauses that can be immediately used to derive new information after backtracking. In the above example, the cuts (i), (ii) and (iii) are asserting while (iv) is not. If cut (iii) is selected, the clause $\neg s$ is learned, backjumping will undo all assignments and the unit rule can be applied immediately to deduce the assignment s:f, which drives the search to a new region of the search space. In contrast, if cut (iv) were selected, the clause $\neg s \vee \neg t$ would be learned; since both s and t are unassigned prior to decision level 1, an asserting backjump is not possible with this clause.

The FIRSTUIP algorithm is shown in Algorithm 14. It consists of a main loop, which iteratively finds new conflict reasons by traversing the implication graph backwards. As an initial conflict reason R, those nodes are chosen that directly contradict the conflict clause stored in $\mathsf{reasons}[\bot]$. The algorithm steps backwards through the trail data structure, skipping over trail assignments that did not contribute to the conflict. When an assignment $\langle p{:}v \rangle$ is encountered that contributed to the conflict, it is removed from R and replaced by a set of assignments sufficient to infer $\langle p{:}v \rangle$. This set is obtained by analyzing the clause stored at $\mathsf{reasons}[p]$.

Algorithm 14: FIRSTUIP conflict analysis.

```
in   : trail – sequence of propositional assignments
       reasons – mapping from prop. assignments to clauses
out  : conflict reason as set of literals
Analyse(trail, reasons)
    $R \leftarrow \{\langle var(l){:}\neg phase(l) \rangle \mid l \in \mathsf{reasons}[\bot]\}$;
    for $(i \leftarrow |\mathsf{trail}|; \neg$isUIP$(R, \mathsf{trail}); i \leftarrow i - 1)$ do
        $\langle p{:}v \rangle \leftarrow \mathsf{trail}[i]$;
        if $\langle p{:}v \rangle \notin R$ then
            continue;
        end
        $R \leftarrow R \cup \{\langle var(l){:}\neg phase(l) \rangle \mid l \in \mathsf{reasons}[p]\}$;
        $R \leftarrow R \setminus \{\langle p{:}v \rangle\}$;
    end
    return $\{p \mid p \mapsto \mathsf{t} \in R\} \cup \{\neg p \mid p \mapsto \mathsf{f} \in R\}$;

isUIP($R$, trail)
    $d \leftarrow$ deepest decision level on trail;
    if $|\{n \in R \mid \mathsf{dl}(n) = d\}| = 1$ then
        return $\mathsf{t}$;
    else
        return $\mathsf{f}$;
    end
```

The algorithm steps backward through the trail until the set R contains exactly one node on the deepest decision level. Once this is the case, a UIP has been reached, which means that the current cut R is asserting.

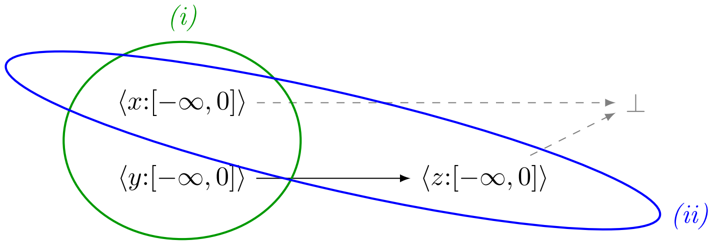
5.4.2 Lifting First-UIP to Lattices

In this section, we lift the FIRSTUIP conflict analysis algorithm to lattices. A trivial lifting of the algorithm will not produce results on structurally rich lattices, as the following example illustrates.

Example 5.4.2. *Consider the following first-order formula φ, where variables are interpreted over the integers and the symbols $+$ and $<$ have their standard meaning.*

$$\varphi \mathrel{\hat{=}} (y = z) \wedge (x + z > 10)$$

We restrict the value of x and y to be zero or negative, in analogy to propositional decision making. This causes a conflict with φ, as recorded in the interval implication graph below.


(i)
$\langle x{:}[-\infty,0]\rangle$
$\langle y{:}[-\infty,0]\rangle$
$\langle z{:}[-\infty,0]\rangle$
$\bot$
(ii)

We can compute a cut in the implication graph above, which yields the two conflict reasons below.

$$\textit{(i)}\ \langle x{:}[-\infty,0], y{:}[-\infty,0]\rangle \qquad \textit{(ii)}\ \langle x{:}[-\infty,0], z{:}[-\infty,0]\rangle$$

While these two reasons are indeed sufficient to produce a conflict, they are not as general as they could be. For example, instead of reason (i), it is preferable to use one of the reasons (iii) and (iv) since they strictly generalize (i).

$$\textit{(iii)}\ \langle x{:}[-\infty,4], y{:}[-\infty,6]\rangle \qquad \textit{(iv)}\ \langle x{:}[-\infty,8], y{:}[-\infty,2]\rangle$$

Note we may choose any combination of upper bounds that add up to 10 to obtain a maximally general reason.

The above example shows that in some lattices such as the intervals, computation and choice of conflict reasons is more complex than in propositional logic. We now introduce the AFIRSTUIP algorithm, a generalization of FIRSTUIP to lattices which is able to find reasons such as (iii) and (iv) above. Conflict analysis with FIRSTUIP implements abductive reasoning (see Section 4.4), that is, it computes an underapproximation of the concrete abduction transformer abd_φ. In terms of ACDCL, this corresponds to computing an underapproximation $u\tilde{f}$ of the dual transformer $\tilde{f}$. The key to AFIRSTUIP is to underapproximate $\tilde{f}$ not just by graph traversal, but also explicitly, using abstract transformers.

Recall that instead of clauses, ACDCL takes a set F of abstract overapproximations of the concrete transformer f. In propositional conflict analysis, the clause information stored in the $\mathsf{reasons}$ array is used to perform abduction. Considered as a whole, the FIRSTUIP algorithm computes an inflationary underapproximation of the abduction transformer abd_φ, since the result of conflict analysis generalizes the original conflict. Inside the procedure, the global abduction task is broken down into small, local abduction steps, which compute generalized reasons for individual deductions.

For example, consider a deduction result $\langle p{:}\mathsf{t}\rangle$, which was derived via a clause $q \vee p$, starting from the partial assignment $\rho = \langle q{:}\mathsf{f}, r{:}\mathsf{t}\rangle$. During the run of the procedure, conflict analysis may replace $\langle p{:}\mathsf{t}\rangle$ with the reason $\langle q{:}\mathsf{f}\rangle$, which generalizes the partial assignment ρ. We model this step as application of an abductive generalization function $\mathsf{gen}_{C,\langle p:\mathsf{t}\rangle} : PAsgs \to PAsgs$ which maps an existing reason ρ for $\langle p{:}\mathsf{t}\rangle$ to a more general reason ρ'. We give another example below to illustrate the concept.

$$\mathsf{gen}_{p\vee\neg q\vee r,\langle r:\mathsf{t}\rangle}(\langle p{:}\mathsf{f}, q{:}\mathsf{t}, s{:}\mathsf{f}\rangle) = \langle p{:}\mathsf{f}, q{:}\mathsf{t}\rangle$$

In ACDCL, this is reflected by requirement (vi) in Figure 5.6. In terms of bottom everywhere, the function gen above is (i) inflationary (since it returns a generalization of the original partial assignment) and (ii) an underapproximation of the function $c \mapsto abd_C(c \cup \gamma(\langle r{:}\mathsf{t}\rangle))$. For general lattices, we assume that each approximation $(of : A \to A) \in F$ of f and element $a \in A$ is associated with an inflationary underapproximation $u\tilde{f}_{of,a} : \mathcal{D}(A) \to \mathcal{D}(A)$ which underapproximates the following function.

$$\tilde{f}_a(c) \mathrel{\hat{=}} \tilde{f}(c \cup \gamma(a))$$

Recall that CDCL uses upwards interpolation to select a conflict reason. To increase efficiency, the FIRSTUIP algorithm only keeps track of *one* conflict reason. In general lattices, some heuristic choice is required, since there may be multiple incomparable generalizations for each application of a transformer $of \in F$. Recall that heuristic choice among conflict reasons is a form of upwards interpolation. We define a special type of upwards interpolation function that returns single elements.

Definition 5.4.1 (ACDCL Interpolation)**.** An ACDCL *interpolation function* is an upwards interpolation function $\Delta_\uparrow : \mathcal{D}(A) \times \mathcal{D}(A) \to \mathcal{D}(A)$ such that for any $a \in A$ and $c \in \mathcal{D}(A)$, $(\downarrow a) \, \Delta_\uparrow \, c$ is of the form $\downarrow a'$ for some $a' \in A$.

The AFIRSTUIP algorithm, shown in Algorithm 15 uses an ACDCL interpolation function, which is requirement (v) in Figure 5.6. In the algorithm, we write $\downarrow a' \leftarrow \downarrow a \, \Delta_\uparrow \, c$ to denote the assignment of a variable a' to the unique maximal element of the downset $\downarrow a \, \Delta_\uparrow \, c$. The main data structure of the algorithm is an array m which we refer to as the *marking*. The marking assigns trail indices to generalizations that are sufficient to produce a conflict.

The initial marking is produced by analyzing the cause of $\bot$. This is accomplished by computing $\downarrow a \, \Delta_\uparrow \, \tilde{uf}_{of,\bot}(\downarrow a)$ where $a = \bigsqcap \mathsf{trail}$ is the final element found in model search, which represents a conflict. The expression $\tilde{uf}_{of,\bot}(\downarrow a)$ computes a downset of generalized candidate reasons for $\bot$. The ACDCL interpolation $\Delta_\uparrow$ selects one , which is stored in a variable r. Next, r is decomposed into meet irreducibles which are then stored at the earliest possible trail marking $m[i]$ such that the resulting element at $m[i]$ generalizes the trail element at index i.

In the main loop, the algorithm steps backwards through the trail and replaces a marking $m[i]$ with a set of earlier trail markings that explain $m[i]$. In order to accomplish this, the `updateMarking` algorithm decomposes the explanation for $m[i]$ into a set of meet irreducibles. Each irreducible q is then placed at an index where it overapproximates the trail element at the same position. This is similar to FIRSTUIP, where a binary marking is used to indicate whether a given trail element is necessary for deriving a conflict. AFIRSTUIP marks trail elements with elements of the abstract lattice, which not only indicate whether a given trail element is necessary, but also how far it can be generalized while preserving the resulting conflict.

In general, a suitable trail position for a generalized element may not exist since the meet decomposition function *mdc* may decompose an element into irreducibles that are incomparable to those on the trail. In order to ensure that the algorithm functions properly, we require that the decomposed generalizations of the trail can be related to the trail. This is expressed by the final requirement (vi) of Figure 5.6.

Assumption 2. *Let M be a set of meet irreducibles of A and $a \in A$. If $\bigsqcap M \sqsubseteq a$ then $\forall q \in mdc(a) \; \exists r \in M. \; r \sqsubseteq q$.*

An invariant that holds after each iteration of the main loop is that the meet of all markings is a conflict reason. The main loop continues until the first UIP is encountered. This behavior is similar to the propositional case. The existence of a UIP can be ensured by choosing an appropriate downwards extrapolation during model search, which only adds a single meet irreducible to the result of transformer application. In this case, the most recent decision element is guaranteed to be a UIP. We do not go into detail, but the techniques for asserting backjumps lift straight from the propositional case [172].

We now show that AFIRSTUIP soundly underapproximates the concrete transformer $\tilde{f}$, which ensures that Assumption 1 is satisfied.

Theorem 5.4.1 (Soundness)**.** *Let a be the return value of* `ACDCL-⊤(trail, reasons)`*, then under Assumption 2 it holds that $\gamma(a) \subseteq \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$.*

We prove some auxiliary lemmas before we proceed with the proof of the theorem.

Algorithm 15: Abstract AFIRSTUIP conflict analysis.

```
in   : trail – sequence of meet irreducibles
       reasons – partial map from $\mathbb{N} \cup \{\bot\}$ to overapproximations $of$ of $f$
out  : conflict reason as abstract element
ACDCL-$\top$(trail, reasons)
    $m \leftarrow \{1 \mapsto \top, \ldots, |\mathsf{trail}| \mapsto \top\}$;
    $of \leftarrow \mathsf{reasons}[\bot]$;
    $a \leftarrow \bigsqcap \mathsf{trail}$;
    $\downarrow r \leftarrow \downarrow a \,\Delta_{\uparrow}\, \widetilde{uf}_{of,\bot}(\downarrow a)$;
    updateMarking($m$, trail, $r$);
    for ($i \leftarrow |\mathsf{trail}|$; isUIP($m$, trail); $i \leftarrow i - 1$) do
        if $m[i] = \top$ then continue;
        $of \leftarrow \mathsf{reasons}[i]$;
        $a \leftarrow \bigsqcap_{1 \leq j < i} \mathsf{trail}[j]$;
        $\downarrow r \leftarrow \downarrow a \,\Delta_{\uparrow}\, \widetilde{uf}_{of,m[i]}(\downarrow a)$;
        updateMarking($m$, trail, $r$);
        $m[i] \leftarrow \top$;
    end
    return $\bigsqcap_{1 \leq i \leq |\mathsf{trail}|} m[i]$;

updateMarking($m$, trail, $r$)
    $Q \leftarrow mdc(r)$;
    foreach $q \in Q$ do
        $j \leftarrow$ smallest index $i$ s.t.$\mathsf{trail}[i] \sqsubseteq q$;
        $m[j] \leftarrow m[j] \sqcap q$;
    end
```

Lemma 5.4.2. *If Assumption 2 holds, then at the end of each iteration of the main loop in Line 7, the following holds.*

$$\forall j.\ i \leq j \leq |\mathsf{trail}| \implies m[j] = \top$$

Proof. The assignment in Line 13 sets $m[i]$ to $\top$ at the current loop index i. All other assignments to m occur inside the call to `updateMarking`. In order to prove the above property, it is therefore sufficient to show that, whenever `updateMarking` is called at Line 12, the element j selected in Line 21 is never greater than i.

Consider the element r that is supplied as an argument to the call to `updateMarking`. The element r is the result of the assignment in Line 11. Recall that the transformer $\tilde{uf}_{of,m[i]}$ is required to be inflationary. Therefore, $r \sqsupseteq a = \bigsqcap_{1 \leq j < i} \mathsf{trail}[j]$. From Assumption 2, we then get that for every $q \in mdc(r)$, there is a j such that $1 \leq j < i$ and $\mathsf{trail}[i] \sqsubseteq q$. Therefore, every element j selected in the call to `updateMarking` is smaller than i which proves the property. □

Lemma 5.4.3. *If Assumption 2 holds, then the following property holds just before execution enters the loop in Line 7.*

$$\gamma(m[1] \sqcap \cdots \sqcap m[|\mathsf{trail}|]) \subseteq \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$$

Proof. Consider the assignment to the variable r in Line 5. Recall that $\tilde{uf}_{of,\bot}$ is an inflationary, sound underapproximation of the function $c \mapsto \tilde{f}(c \cup \gamma(\bot))$. Since $\gamma(\bot) = \bot$, it is therefore also an underapproximation of $\tilde{f}$. It therefore holds after Line 5 that $\gamma(r) \subseteq \tilde{f} \circ \gamma(a) = \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$. Since $\tilde{uf}_{of,\bot}$ is inflationary and $\Delta_\uparrow$ is an interpolation, we have that $r \sqsupseteq a = \bigsqcap \mathsf{trail}$. From Assumption 2, we then have that $\forall q \in mdc(r)\ \exists s \in \mathsf{trail}.\ s \sqsubseteq q$. It follows that, after the call to `updateMarking` in Line 6, $\bigsqcap_{1 \leq l \leq |\mathsf{trail}|} m[l]$ is equal to r. We have shown that, just before the main loop, the two following equalities hold.

$$r = m[1] \sqcap \cdots \sqcap m[|\mathsf{trail}|] \qquad\qquad \gamma(r) \subseteq \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$$

The property follows. □

Lemma 5.4.4. *Consider the two following properties.*

(i) $\gamma(m[1] \sqcap \cdots \sqcap m[i]) \subseteq \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$

(ii) $\gamma(m[1] \sqcap \cdots \sqcap m[i-1]) \subseteq \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$

In every iteration of the main loop, if (i) holds at the beginning of the loop body, then (ii) holds at the end.

Proof. Assume (i) holds at the beginning of some iteration of the main loop. Let m' denote the marking at the start of the loop body, and m denote the marking after the loop body executes. The marking m is related to m' as follow. From Lemma 5.4.2 we have that for all indices $j \geq i$, it holds that $m[j] = \top$. For all indices $j < i$, it holds that $m'[j] = m[j]$ or $m'[j] = m[j] \sqcap q_1 \sqcap \cdots \sqcap q_k$, where $q_1, \ldots, q_k \in mdc(r)$, from the call to `updateMarking` in Line 12. From Assumption 2, we get that for all $q \in mdc(r)$ there is some index l such that $m[l] \sqsubseteq q$. Therefore, we have the following.

$$m[1] \sqcap \cdots \sqcap m[i-1] = m'[1] \sqcap \cdots \sqcap m'[i-1] \sqcap r$$

From the assignment in Line 11, we get the following.

$$\gamma(r) \subseteq \tilde{f}(\gamma(\bigsqcap_{1 \leq j < i} \mathsf{trail}[j]) \cup \gamma(m'[i]))$$

Note that $\tilde{f}$ is additive, since for any concrete elements $a, b \in B$, we have that $\tilde{f}(a \cup b) = a \cup b \cup \tilde{f}(\bot)$ (by Proposition 3.2.1) which is equal to $\tilde{f}(a) \cup \tilde{f}(b)$. We may rewrite the condition above as follows.

$$\gamma(r) \subseteq \tilde{f}(\circ\gamma(\bigsqcap_{1\leq j<i} \mathsf{trail}[j]) \cup \tilde{f} \circ \gamma(m'[i]))$$

The trail marking at i is the result of applying an approximation of f to $\bigsqcap_{1\leq j<i} \mathsf{trail}[j]$, therefore it holds that $f \circ \gamma(\bigsqcap_{1\leq j<i} \mathsf{trail}[j]) \subseteq \gamma(\mathsf{trail}[i])$. It is easy to see by construction of trail markings that they generalize the trail. Therefore, we have that $\mathsf{trail}[i] \sqsubseteq m'[i]$. By transitivity and monotonicity of γ, we get that $f \circ \gamma(\bigsqcap_{1\leq j<i} \mathsf{trail}[j]) \subseteq \gamma(m'[i])$. This in turn allows us to conclude the following.

$$\gamma(r) \subseteq \tilde{f} \circ \gamma(m'[i])$$

We apply the function f to our characterization of m in terms of m'.

$$\begin{aligned} f \circ \gamma(m[1] \sqcap \cdots \sqcap m[i-1]) &= f \circ \gamma(m'[1] \sqcap \cdots \sqcap m'[i-1] \sqcap r) \\ &= f \circ \gamma(m'[1]) \cap \cdots \cap f \circ \gamma(m'[i-1]) \cap f \circ \gamma(r) \end{aligned}$$

Since $\gamma(r) \subseteq \tilde{f} \circ \gamma(m'[i])$, it also holds that $f \circ \gamma(r) \subseteq \gamma(m'[i])$, and that $f \circ \gamma(r) \subseteq f \circ \gamma(m'[i])$. We conclude the following.

$$\begin{aligned} f \circ \gamma(m[1] \sqcap \cdots \sqcap m[i-1]) &\subseteq f \circ \gamma(m'[1]) \cap \cdots \cap f \circ \gamma(m'[i-1]) \cap f \circ \gamma(m'[i]) \\ &\subseteq f \circ \gamma(m'[1] \sqcap \cdots \sqcap m'[i]) \end{aligned}$$

From (i), we have $\gamma(m'[1] \sqcap \cdots \sqcap m'[i]) \subseteq \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$, which is equivalent to $f \circ \gamma(m'[1] \sqcap \cdots \sqcap m'[i]) \subseteq \gamma(\bigsqcap \mathsf{trail})$. We have therefore established the following ordering.

$$f \circ \gamma(m[1] \sqcap \cdots \sqcap m[i-1]) \subseteq f \circ \gamma(m'[1] \sqcap \cdots \sqcap m'[i]) \subseteq \gamma(\bigsqcap \mathsf{trail})$$

From transitivity, we get that $f \circ \gamma(m[1] \sqcap \cdots \sqcap m[i-1]) \subseteq \gamma(\bigsqcap \mathsf{trail})$, which is equivalent, by the defining property of the Galois connection, to $\gamma(m[1] \sqcap \cdots \sqcap m[i-1]) \subseteq \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$, which completes the proof. □

We are now ready for the proof of the soundness theorem.

Proof of Theorem 5.4.1. We prove inductively that, at the end of every iteration of the main loop, the following holds.

$$\gamma(\bigsqcap_{1\leq j\leq \mathsf{trail}} m[j]) \subseteq \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$$

The base step is established by Lemma 5.4.3, the induction step by Lemma 5.4.4. Note that the return value a of the algorithm is given by $\bigsqcap_{1\leq j\leq \mathsf{trail}} m[j]$. Then the following statement holds, which completes the proof.

$$\gamma(a) \subseteq \tilde{f} \circ \gamma(\bigsqcap \mathsf{trail})$$

□

We have shown that conflict analysis is sound, which establishes Assumption 1 used in the soundness proof of Theorem 5.3.2. Another assumption used in the proof was that all backjumps are asserting. AFIRSTUIP ensures this requirement by constructing conflict reasons that contain exactly one element associated with the most recent decision level, namely, the first UIP. The argument is symmetric to the propositional case [172].

5.5 Related Work

This chapter introduced ACDL, a framework based on abstract interpretation for extending the scope of learning techniques in decision procedures. We also presented ACDCL, an instantiation of ACDL that uses a generalization of clause learning.

The literature contains several proposals for lifting CDCL from SAT solvers to new domains. In the remainder of this section, we compare our work to formal frameworks that enable a systematic lifting of CDCL to new domains.

5.5.1 The DPLL(T) Framework

The most popular framework for extending CDCL to first-order logics is DPLL(T) [65]. A formalization of DPLL(T) is presented in [141, 142] in the form of a non-deterministic transition system. A well-formed trace of the transition system corresponds to a run of an SMT solver. A strategy for resolving non-determinism corresponds to a concrete algorithmic instantiation.

DPLL(T) extends a CDCL solver by interfacing it with a first-order theory solver that is complete for determining if a conjunction of theory literals is satisfiable. In terms of abstract interpretation, DPLL(T) corresponds to a product construction, where one component is fixed and the other is a parameter (see Section 4.5). ACDCL has no such restriction and describes a class of solvers over abstract domains which includes product constructions but is not limited to them. An advantage of the product construction in DPLL(T) is that it allows for modular implementation. A single implementation of DPLL(T) can be interfaced with different theory solvers to produce a number of different decision procedures. As we will see in Chapter 6, ACDCL also allows for implementations that separate the algorithmic framework from the underlying abstract domain.

While ACDCL instantiates a generalized CDCL algorithm over an abstract domain, DPLL(T) extends an existing propositional solver by connecting it to a first-order reasoning module. The downside of the latter approach is that the vocabulary of the solver is inherently limited to propositional facts. This limits the effectiveness of decision making and learning, both of which operate over the propositional component of the solver, which can lead to performance problems (experiments to this effect are presented in [25]). Furthermore, vocabulary limitations of DPLL(T) place strict completeness requirements on the theory solver. Since the SAT solver inside DPLL(T) enumerates conjunctions of atoms, the theory solver needs to be complete for the conjunctive fragment of the logic. In ACDCL, it is possible to instantiate complete satisfiability solvers even when the underlying abstract domain does not support complete reasoning over conjunctions. We will present an example of this in Chapter 6.

The limitations of DPLL(T) discussed above are well-known. To avoid the interface restrictions between propositional and theory solvers, research in DPLL(T) has increasingly sought to integrate the two components more tightly at the expense of reducing modularity [37]. In terms of the DPLL(T) framework, this has led to two developments. The first is research on natural-domain SMT procedures, which aim to instantiate the CDCL algorithm over new target domains. We will discuss these procedures in the next section. The second development concerns proposed extensions to the basic DPLL(T) framework by on-the-fly introduction of new propositional vocabulary. Work in this area includes development of theory-based decision heuristics [8, 71] and dynamic instantiation of first-order axioms in propositional logic [62]. The first approach enables making decisions on theory facts by dynamically adding propositions to represent them; the second allows simulation of a kind of theory learning by dynamically adding propositional instantiations of first-order theory axioms. Since ACDCL does not distinguish between propositional and theory facts, making decisions and learning automatically happens in the theory.

ACDCL provides a systematic procedure to derive richly semantic conflict analyses from abstract domains. In contrast, DPLL(T) provides no algorithmic guidance for constructing conflict analysis procedures. The abstract framework of [141, 142] models learning as a single step of the transition system in which a true formula is added to the instance.

5.5.2 Frameworks for Natural Domain SMT

We use the term *natural domain* SMT to denote approaches to SMT solving that are similar to CDCL but do not use a propositional SAT solver. The term originates in [37]. We now discuss frameworks for natural domain SMT that have been proposed in the literature and compare them to ACDCL.

Cotton's Natural Domain SMT Framework The framework in [37], which we will denote by NDSMT, proposes a depth-first search procedure that operates on non-Boolean partial assignments, which map first-order variables to domain values. Non-Boolean partial assignments form an abstract domain. If the first-order variables in V take values in a domain D, then the partial assignments domain for V and D is given by the lattice $\{\bot\} \cup (V \to D \cup \{\top\}$. Note that this is exactly the lattice of constants in program analysis.

The core procedure repeatedly refines a partial assignment ρ, by making *selections*, where some variable in V is assigned to a value in D. Valid selection results are required to satisfy certain consistency criteria. In effect, selections combine decisions and deduction into a single black-box function call. This process repeats until the partial assignment ρ is total, or until an inconsistency is detected. Inconsistencies are resolved by a blackbox learning procedure.

Compared to the original CDCL algorithm, there are significant differences in the framework presented in [37]. In the propositional algorithm, decisions perform a binary case analysis. This is not the case in [37], since variables are assigned to one of many values. Unlike CDCL, decisions cannot be simply complemented.

In terms of abstract satisfaction, NDSMT is an instance of ACDL, but not ACDCL. The first order partial assignments lattice used in NDSMT does not have complementable meet irreducibles and does therefore not support generalized clause learning. Another difference is that the lattice in NDSMT is fixed whereas ACDCL can be instantiated on a number of different abstract lattices. Further, the ACDCL framework gives a systematic methodology to synthesize model search and conflict analysis procedures from abstract domains, whereas the framework in [37] provides very little guidance.

Generalized DPLL Another generalization of CDCL, called *Generalized* DPLL (GDPLL), is presented in [124]. The basic data structure in GDPLL for a logic $(\mathcal{F}, \models, \mathcal{S})$ is a semantic structure $\sigma \in \mathcal{S}$. Instead of assigning unassigned variables to values, GDPLL mutates semantic structures by reassigning variables to new values. Modeling CDCL as an instance of GDPLL requires the use of non-standard semantics for propositional logic.

The state of the procedure is a pair (F, σ) where $F \subseteq \mathcal{F}$ is a set of formulas and $\sigma \in \mathcal{S}$ is a semantic structure. The framework consists of alternating two steps until either a satisfying assignment is found, or a contradiction is derived. The first step mutates the structure σ by reassigning some variables to new values. The second step adds a new formula φ to F. Both steps are required to advance the pair (F, σ) along some well-founded ordering to ensure termination.

ACDL and ACDCL provide systematic frameworks for deriving model search, conflict analyses and learning procedures. GDPLL offers no solution as to how a mutation or deduction should operate. ACDCL can be used to derive decision procedures from abstractions simply by providing an extended abstract domain interface. GDPLL is restricted to operate on single assignments, although this requirement can be weakened by interpreting logics over non-standard models. We feel that abstract interpretation is a conceptually more robust and more exhaustively studied approach to approximation than the use of non-standard logical semantics.

Model Constructing Satisfiability Calculus The work in [103, 56], which extends previous work in [104], presents the *Model-Constructing Satisfiability Calculus* (MCSAT). MCSAT is more operational in nature than GDPLL and NDSMT and is presented as an abstract transition system. The core idea is to generalize CDCL by extending the trail data structure. In addition to propositional assignments, special model assignments occur on the trail which assign first-order variables to domain values.

In terms of abstract satisfaction, the main data structure in MCSAT is the Cartesian product of three domains: a Boolean partial assignments lattice, a first-order partial assignments lattice and the abstract domain of the theory solver. Since MCSAT includes a propositional solver, it suffers the same vocabulary restrictions as DPLL(T). To solve this problem, the framework allows for new propositional variables to be introduced on the fly to represent new facts about the theory. This allows the algorithm to simulate richer lattices in decision making and learning over richer lattices.

For conflict analysis and learning, MCSAT requires the implementation of a blackbox function by the theory solver which explains deductions in terms of propositional clauses. In contrast, given an abstract domain with the necessary complementation properties ACDCL automatically synthesizes a learning procedures.

An instantiation of MCSAT operates in three distinct worlds: the Boolean world of propositions, which abstractly represent theory facts, assignments of first-order variables to domain values and the data structure of the theory solver. For many applications, the redundant representation of information in three separate domains is unnecessary. In these cases, ACDCL offers a conceptually more concise solution to algorithm design. Since MCSAT requires some amount of bookkeeping to keep propositional variables and theory facts synchronized, ACDCL-based algorithms may offer more efficiency in cases where MCSAT would generate a large number of new propositions during the run of the procedure.

ACDCL is based on abstract satisfaction and can handle problems in logical satisfiability and program correctness. We are unaware of other algorithmic frameworks that are general enough to allow derivation of procedures in both of these problem domains.

Using Abstract Interpreters in Satisfiability Procedures Recent work in constraint programming has explored the systematic integration of abstract domains into constraint programming frameworks [145] and the use of constraint programming algorithms in abstract interpretation [148, 147]. In contrast to the work presented in this chapter, these approaches do not address learning.

Using Satisfiability Procedures in Abstract Analyses ACDCL instantiates a decision procedure architecture in the context of an abstract domain. We now review some approaches that aim to improve abstract analyses using decision procedures. All of the following approaches combine an instance of a logical satisfiability procedure with an abstract analysis, whereas we use the architecture of a satisfiability procedure on top of an abstract domain.

A general means to abstract-interpretation based approximation of logical semantics using decision procedures is formalized in the *from-below* approximation technique described in [150] and developed for a special case in [108]. In this technique, a semantic approximation of a logical formula is constructed by computing an upwards iteration sequence. The next element of the iteration sequence is determined by a satisfiability solver that reasons about the concrete semantics of the formula. The solver identifies a small concrete element that is not yet soundly approximated by the current iterate. The next iterate is then computed by conjoining the current iterate with an abstract representation of this concrete element. Once the satisfiability solver cannot find any more concrete elements, the abstract element represents a sound approximation. In [135], a variant of this technique is applied to approximate a least fixed point of the static analysis equation: an SMT solver is used to find well-formed traces of a program that are not yet covered by a candidate approximation. These traces are then abstractly analyzed and the result is joined together with the current candidate approximation to produce the next approximation.

Another approach for approximating logical semantics which works “from above” is presented in [165, 167]. The technique computes a downwards iteration sequence starting from $\top$. In each iteration, the procedure picks an abstract element a and determines with a decision procedure whether the set $\neg\gamma(a)$ intersects the semantics of the concrete system under consideration. If not, then the next element of the iteration sequence is computed as the meet of the previous element and a. The “from above” and “from below” techniques can be combined into an algorithm for bilateral approximation [167].

Chapter 6

Deciding Floating-Point Logic with ACDCL

The previous chapter characterizes ACDCL as an algorithm to solve the bottom-everywhere problem. A practical consequence of this characterization is that ACDCL can be instantiated to yield satisfiability checkers for richer logics. This chapter presents FP-ACDCL [89, 26], a satisfiability checker for floating-point logic based on an instantiation of the ACDCL framework over the abstract domain floating-point intervals. FP-ACDCL was implemented by Alberto Griggio inside the MATHSAT [33] SMT solver, based on an unpublished prototype by the author. We show experimentally that FP-ACDCL is a state-of-the-art solver. Traditional solvers are based on bit-vector encodings of floating-point logic combined with propositional satisfiability solving. On a large set of benchmarks, FP-ACDCL outperforms this approach in 80% of cases and is faster by one order of magnitude or more on 60% of the benchmarks.

In program analysis, abstract interpretation is both a methodological as well as an algorithmic framework, in which a fixed point iteration engine communicates with an abstract domain via an abstract domain interface. New analysis algorithms can be instantiated by providing new abstract domains leaving the iteration algorithms untouched. Similarly, we have implemented an ACDCL framework designed for logical decision procedures. FP-ACDCL is the instantiation of this framework on the domain of floating-point interval environments. Other instantiations can be obtained by plugging new abstract domains into the framework. The formal requirements for ACDCL (summarized in Figure 6.1) translate to an extended abstract domain interface.

Motivation Floating-point computations are pervasive in low-level control software and embedded applications. Such programs are frequently used in contexts where safety is critical, such as automotive and avionic applications. Scalable and accurate reasoning about floating-point numbers is therefore an important problem. An SMTLIB [10] theory of Floating-Point Arithmetic (FPA) was proposed [153] to facilitate developments of tools to solve this problem.

Previous approaches to deciding FPA are based on translating FPA problems to equisatisfiable propositional formulas by a process called bit-blasting [55, 33, 113]. Bit-blasting yields large formulas that are hard for current SAT solvers. In contrast, FP-ACDCL uses interval propagation which is extremely efficient. For insight into the operation of our solver, consider the following formula where the variables x and y have double-precision floating-point values.

$$0.0 \leq x \wedge x \leq 10.0 \wedge y = x^5 \wedge y > 10^5$$

Floating-point interval propagation [127] tracks the range of each variable and can derive the fact $x \in [0.0, 10.0]$ from the first two constraints, which implies the fact $y \in [0.0, 100000.0]$ from the third constraint. This range is not compatible with the final conjunct $y > 10^5$, so

(i) An overapproximate lattice with f-complementable meet irreducibles

$$(A, \sqsubseteq, \sqcap, \sqcup),$$

(ii) a downwards extrapolation

$$\nabla_{\downarrow} : A \times A \to A,$$

(iii) a set of overapproximations of f

$$F \subseteq A \to A,$$

(iv) for each $\mathit{of} \in F$ and $a \in A$, an inflationary underapproximation

$$\mathit{uf}_{\mathit{of},a} : \mathcal{D}(A) \to \mathcal{D}(A)$$

of the transformer $\tilde{f}_a \mathrel{\hat{=}} c \mapsto \tilde{f}(c \cup \gamma(a))$,

(v) an ACDCL (upwards) interpolation

$$\Delta_{\uparrow} : \mathcal{D}(A) \times \mathcal{D}(A) \to A,$$

(vi) a finite meet factorization $\mathit{mdc} : A \to \mathcal{P}(A)$ such that the following holds for all sets $M \in \mathbb{I}_{\sqcap}(A)$ and elements $a \in A$.

$$\text{If } \bigsqcap M \sqsubseteq a \text{ then } \forall q \in \mathit{mdc}(a).\ \exists r \in M.\ r \sqsubseteq q.$$

Figure 6.1: Requirements for applying ACDCL to the bottom-everywhere problem for the algebra $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, c\tilde{f})$.

the formula is unsatisfiable. The computation requires a fraction of a second. In contrast, translating the formula above into a bit-vector and invoking the SMT solver Z3 to prove unsatisfiability takes 16 minutes on a 2.6 GHz processor.

The efficiency of interval reasoning comes at the cost of completeness. Consider the floating point formula below.

$$z = y \land x = y \cdot z \land x < 0$$

After bit-vector encoding, the solver Z3 can decide satisfiability of this formula in a fraction of a second. The interval abstraction cannot represent relationships between the values of variables. Interval propagation will not deduce that y and z are either both positive or both negative. The interval solver cannot conclude that x must be positive and cannot show that the formula is unsatisfiable. The ACDCL framework is a means of recovering precision lost by interval analysis, while retaining efficiency.

Outline This chapter is organized as follows. Section 6.1 informally introduces floating point numbers and the SMTLIB theory FPA. Section 6.2 introduces an extended abstract domain interface for satisfiability algorithms and discusses the implementation of the requirements listed in Figure 6.1. Section 6.3 provides experimental results and analyzes the trade-off between computational effort spent in generalizing conflicts and overall algorithm runtime. Related work is discussed in Section 6.4.

6.1 Floating-Point Arithmetic

This section provides an informal introduction to floating-point numbers and some issues surrounding formal reasoning about floating-point. For a more in-depth treatment see [138, 133]. Floating-point numbers are representations of the real numbers that were designed to allow for effective encoding using bitvectors of fixed width. They form a finite subset of the rational numbers extended by a set of special values. The reference for modern floating-point implementations is the IEEE754 standard [50].

The IEEE754 standard represents floating-point values as a triple of bounded numbers (s, e, m), where s is the *sign bit* and is either 0 or 1, e is the *exponent* and m is the *significand.* The maximum and minimum size of e and m depends on the specific floating-point format used. The rational number represented by this pattern can be obtained as follows.

$$(-1)^S \cdot 2^{e-bias} \cdot m$$

The value *bias* is a format-dependent constant. An example of an IEEE754 `binary16` floating-point number and its encoding as a bitvector is given below.

$$\underbrace{\boxed{1}}_{s}\underbrace{\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}}_{e}\underbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}}_{m}\ \begin{aligned}&= -1 \cdot 2^{18-15} \cdot (1 + 2^{-2} + 2^{-4} + 2^{-6} + 2^{-7})\\ &= -10.6875\end{aligned}$$

The IEEE754 binary encodings specify several different classes of numbers; normal, subnormal, zeros, infinities and "not a number" (*NaN*). An example of a normal number is shown above. Subnormal numbers represent numbers close to zero and have a slightly different encoding. Due to the presence of the sign bit, there are two distinct values that represent the number 0, one signed, the other unsigned. There are two infinity values, ∞ and $-\infty$ which are the result of certain operations such as divisions by zero and operations that result in arithmetic overflows. Finally, the class of *NaN* values represent indeterminate forms, such as $0/0$ or $(-\infty) + \infty$.

IEEE754 specifies the semantics of arithmetic operations with respect to one of five rounding modes. Three of these are directed (rounding up, rounding down and rounding towards zero) and two round to the nearest number. Rounding is one of the features that increases the difficulty of both writing and verifying floating-point programs. As a result of rounding, floating-point addition and multiplication is not associative even when restricted to normal numbers.

Consider the two floating-point expressions below where, the numbers are represented by 32 bits with a 24 bit mantissa (the value 16777216 is 2^{24}).

$$(1 + 16777216) + -16777216 = 0 \qquad 1 + (16777216 + -16777216) = 1$$

25 bits are required to represent the sum of 1 and 2^{24}, so rounding is applied and 2^{24} is returned. Thus, the expression on the left evaluates to 0. On the right, no rounding is required and the result is 1.

Moreover, floating-point addition does not distribute over multiplication because of rounding effects, as the example below demonstrates.

$$2049 * (8189 + 1) = 16781310 \qquad (2049 * 8189) + (2049 * 1) = 16781308$$

In the absence of associativity and distributivity, several standard algebraic approaches to reasoning about arithmetic expressions are inapplicable.

6.1.1 Floating-Point Logic

The SMTLib theory of Floating-Point Arithmetic (FPA) is a language for expressing constraints in terms of floating-point numbers, variables and operators.

A term in FPA is constructed from floating-point variables, constants, standard arithmetic operators and special operators. Examples of special operators include square roots and combined multiply-accumulate operations used in signal processing. Each operation must be parameterized by one of the five rounding modes.

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 $\leq$. These comparisons have to handle all classes of numbers. Normal and subnormal numbers are compared in the expected way. The two zeros, $+0$ and -0 are regarded as equal (despite having distinct floating-point representations) as they correspond to the same number. Infinities are respectively above (∞) and below ($-\infty$) all of the preceding classes. Finally, NaN is regarded to be unordered and incomparable to all floating point numbers, thus all comparisons involving NaN, including $NaN =_{\mathbb{F}} NaN$, are false. Therefore, the standard standard equality predicate $=$ is reflexive but the floating-point equality predicate $=_{\mathbb{F}}$ is not.

Floating-point satisfiability as bottom-everywhere We denote by $\mathcal{F}_{\mathbb{F}}$ the set of floating-point formulas. Floating-point formulas are interpreted over the set of concrete floating-point assignments $Vars \to \mathbb{F}$, via a semantic entailment relation $\models$. Formally, FPA then forms the following SMT logic.

$$(\mathcal{F}_{\mathbb{F}}, \models, Vars \to \mathbb{F})$$

The satisfiability problem for a formula $\varphi \in \mathcal{F}_{\mathbb{F}}$, corresponds to the bottom-everywhere problem for the following bottom-everywhere algebra.

$$(\mathcal{P}(Vars \to \mathbb{F}), \subseteq, \cap, \cup, ded_{\varphi}, abd_{\varphi}, cded_{\varphi}, cabd_{\varphi})$$

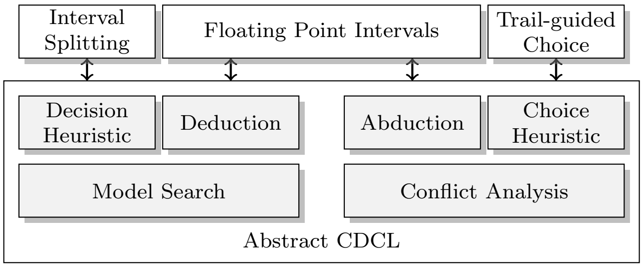


Figure 6.2: FP-ACDCL solver architecture.

6.2 ACDCL over Floating-Point Intervals

FP-ACDCL is an instantiation of ACDCL which communicates with the underlying abstract domain via a fixed interface. Figure 6.2 shows the overall architecture. The interface between ACDCL and the abstract domain of interval environments is an extension of the default abstract domain interface used by program analyzers, which includes functions to test for ordering, or to compute joins, meets and apply various transformers.

6.2.1 Floating-Point Intervals

We now define the floating-point interval abstraction used in FP-ACDCL. Intervals approximate sets of numbers by their closest enclosing range. In addition to the arithmetic ordering $\leq$, the IEEE754 standard dictates the existence of a total order $\preceq$ on all floating-point values, including special values such as NaN. The standard does not precisely define this order, but indicates that it should exist in any implementation and should agree with the arithmetic ordering. For our purposes, we can think of special values being larger than numeric values. The interval abstraction is defined with respect to this total order (we use the total order as a formal trick to simplify exposition, in practice, special values should be treated separately from numeric values). We write $\min_{\preceq}$ and $\max_{\preceq}$ for the minimum and maximum with respect to the $\preceq$ order.

Definition 6.2.1 (Floating-Point Intervals)**.** The lattice $(Itv_{\mathbb{F}}, \sqsubseteq, \sqcap, \sqcup)$ of *floating-point intervals* is defined below.

1. The set of lattice elements is $Itv_{\mathbb{F}} \mathrel{\hat{=}} \{[a,b] \mid a, b \text{ are in } \mathbb{F} \text{ and } a \preceq b\} \cup \{\bot\}$.
2. The meet $\bot \sqcap y = y \sqcap \bot = \bot$ for all y. The meet $[a,b] \sqcap [c,d]$ is the interval $[\max_{\preceq}(a,c), \min_{\preceq}(b,d)]$ if $\max_{\preceq}(a,c) \preceq \min_{\preceq}(b,d)$ holds and is $\bot$ otherwise.
3. The join $\bot \sqcup y = y \sqcup \bot = y$. The join $[a,b] \sqcup [c,d]$ is $[\min_{\preceq}(a,c), \max_{\preceq}(b,d)]$.

Definition 6.2.2 (Floating-Point Interval Environments)**.** Given a finite set of variables *Vars*, the *lattice of floating-point interval environments* is the lattice below.

$$(Vars \to Itv_{\mathbb{F}}, \sqsubseteq, \sqcap, \sqcup)$$

The ordering, meets and joins are lifted pointwise from $Itv_{\mathbb{F}}$.

As usual, we denote an element of the form $f : \{x, y\} \to Itv_{\mathbb{F}}$ as a tuple $\langle x{:}f(x), y{:}f(y)\rangle$ of variables paired with intervals. We omit from the tuple variables that map to $\top$. That is, if $f(x)$ is $\top$ and $f(y)$ is not, we write $\langle y{:}f(y)\rangle$.

The interval lattice overapproximates the powerset lattice of floating-point numbers w.r.t. to the Galois connection below.

$$\begin{array}{lll} \alpha : \mathcal{P}(\mathbb{F}) \to Itv_{\mathbb{F}} & \alpha(\emptyset) \mathrel{\hat{=}} \bot & \alpha(S) \mathrel{\hat{=}} [\min_{\preceq}(S), \max_{\preceq}(S)], \text{ for } S \neq \emptyset \\ \gamma : Itv_{\mathbb{F}} \to \mathcal{P}(\mathbb{F}) & \gamma(\bot) \mathrel{\hat{=}} \emptyset & \gamma(f) \mathrel{\hat{=}} \{\{x \mapsto v \mid x \in Vars, v \in f(x)\}\} \end{array}$$

The Galois connection for interval environments can be obtained in the usual way.

```
interface DecisionHeuristic
    function decide(elem: AbstractElement)
    return MeetIrreducible

    procedure update(elem : MeetIrreducible, added : Bool, level : Int)

interface Deduction
    function deduce(elem: AbstractElement, c : Constraint)
    return list⟨MeetIrreducible⟩

interface ChoiceHeuristic
    procedure update(elem : MeetIrreducible, added : Bool, level : Int)

interface Abduction
    function abduce(h : ChoiceHeuristic, elem : AbstractElement,
                    deduction : MeetIrreducible, reason : Constraint)
    return list⟨MeetIrreducible⟩
```

Figure 6.3: Pseudocode of extended domain interface functions for ACDCL.

Complementable Meet Irreducibles The lattice of floating-point interval environments admits complementable meet irreducibles. The meet irreducible elements of the interval lattices are upper or lower bounds on single variables, that is, elements that represent half-spaces in the concrete. These elements are of the form $\langle x{:}[\min_{\preceq}(\mathbb{F}), c]\rangle$ or of the form $\langle x{:}[c, \max_{\preceq}(\mathbb{F})]\rangle$ and they can be complemented precisely by $\langle x{:}[c_{+}, \max_{\preceq}(\mathbb{F})]\rangle$ and $\langle x{:}[\min_{\preceq}(\mathbb{F}), c_{-}]\rangle$, respectively. The constants c_{+} and c_{-} represent the values immediately above and below c in the order $\preceq$, respectively. For easier readability, we define the following shorthands for denoting meet-irreducible elements.

$$\begin{aligned}
\langle x \preceq c\rangle &\mathrel{\hat{=}} \langle x{:}[\underset{\preceq}{\min}(\mathbb{F}), c]\rangle \\
\langle x \succeq c\rangle &\mathrel{\hat{=}} \langle x{:}[c, \underset{\preceq}{\max}(\mathbb{F})]\rangle \\
\langle x \prec c\rangle &\mathrel{\hat{=}} \langle x{:}[\underset{\preceq}{\min}(\mathbb{F}), c']\rangle && \text{where } c' \text{ is maximal s.t. } c' \prec c \\
\langle x \succeq c\rangle &\mathrel{\hat{=}} \langle x{:}[c', \underset{\preceq}{\max}(\mathbb{F})]\rangle && \text{where } c' \text{ is minimal s.t. } c' \succ c
\end{aligned}$$

6.2.2 Extended Domain Interface

The extended interface functions used by our implementations of ACDCL are shown in Figure 6.3. We will now relate these functions to the formal requirements of Figure 6.1.

Downwards Extrapolation The downwards extrapolation function $\nabla_{\downarrow} : A \times A \to A$ is implemented via the `decide` function. The interface is simpler than that suggested by the formal definition since it implements a unary rather than a binary operator. This removes some flexibility from the framework, but simplifies the interface.

The deduction interface also allows for updates, which notify the decision heuristic when a meet irreducible is added or removed from the trail. This information can be used for heuristic purposes.

Set of Transformers F and Meet Factorization mdc The function `deduce` implements an overapproximate deduction transformer. The arguments to the function are an abstract element and a clause C. Note that our solver operates on formulas φ in CNF. The initial set of transformers F is then given by the set of functions $\{a \mapsto \texttt{deduce}(a, C) \mid \text{ Clause } C \in \varphi\}$. The return value of the functions is a set of meet irreducibles rather than an abstract element, that is, the function $\texttt{deduce}(a, C)$ computes $mdc \circ oded_C$ for some overapproximation $oded_C$ of ded_C. The meet decomposition in intervals is implemented by enumerating single lower and upper bound constraints on variables and satisfies condition (vi) in Figure 6.1.

Underapproximating $\tilde{f}$ and Upwards Interpolation Recall that AFIRSTUIP generalizes a deduction $a \to b$ by calling $\Delta_{\uparrow} \circ u\tilde{f}_{of,b}$. The call to $u\tilde{f}_{of,b}$ with an argument $\downarrow a$ finds a downset representing a number of candidate reasons for b that generalize a. The upwards interpolation function $\Delta_{\uparrow}$ picks a single reason from that set, which generalizes $\downarrow a$. The interface of our ACDCL implementation combines the computation of $\Delta_{\uparrow}$ and $u\tilde{f}_{of,b}$ in the interface function `abduce` below. The function takes the following arguments: (i) a choice-heuristic object, which informs the computation of the abduction result in the presence of multiple, incomparable candidates, (ii) a known reason for the deduction result which is to be generalized by the function application, (iii) a deduced meet-irreducible to which abduction should be applied and (iv) a constraint C. The function $u\tilde{f}_{oded_C,b}(m)$ is implemented by a call to $\texttt{abduce}(\cdot, b, m, C)$. Like the decision heuristic, the choice heuristic may be updated with newly deduced or removed meet irreducibles to enable dynamic heuristics that take into account search history.

6.2.3 Deduction and Decisions

We implement overapproximations of the deduction transformer ded_φ using standard Interval Constraint Propagation (ICP) techniques for floating-point numbers, defined e.g., in [127, 20]. The implementation operates on CNF formulas over floating-point predicates. The initial set of transformers F corresponds to a set of overapproximate transformers $oded_C$ of ded_C for each clause C of the formula. Propagation uses an occurrence-list approach which associates with each variable a list of the FPA clauses in which the variable occurs. Whenever a new meet irreducible is added to the trail, we scan the list of affected clauses and apply ICP to check for new deduction results. Learned unit transformers are stored as vectors of meet irreducible elements and are propagated in a similar way.

FP-ACDCL performs decisions by adding to the trail a meet irreducible element $\langle x \preceq b\rangle$ or $\langle x \succeq b\rangle$. There are a number of heuristic choices: (i) selecting the variable x, (ii) selecting the bound value b and (iii) choosing between bounding x from below and from above.

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 that *fairness* considerations are critical for performance. Decisions should be balanced across different variables and upper and lower bounds. In experiments, concentrating decisions on a single variable and bound shows inferior performance compared to a "breadth-first" exploration in which decisions and bounds are restricted more uniformly.

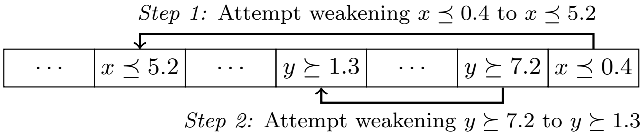
FP-ACDCL performs decisions as follows: (i) variables are statically ordered and the selection of which variable x to branch on is cyclic across this order; (ii) the bound b is chosen to be an approximation of the arithmetic average between the current bounds l and u on x; note that the arithmetic average is different from the median, since floating-point values are non-uniformly distributed across the reals (using the median instead did not yield good results); (iii) the choice between bounding a variable from above or from below is random. The above strategy is naive. We suspect that integrating fairness considerations with activity-based heuristics typically used in CDCL solvers could lead to significant performance improvements.

6.2.4 Abduction with Trail-Guided Choice

We perform abduction by relaxing bounds iteratively. For a given interval element, there may be many incomparable relaxations that are valid abduction results. Our experiments suggest that the 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. Learned unit rules are used to drive backjumping. Backjumps are an effective way for CDCL-style solvers to recover from bad decisions. The backjump potential is determined by the search depth of facts used to explain the conflict. If the conflict reason uses elements that occur early in the trail, then backjumping can reset the trail to an earlier decision level. A bad generalization heuristic might prematurely generalize some facts too far so that elements that lie late on the trail are required in order to guarantee a conflict. Trail-guided choice generalizes backwards in the order of the trail and thus heuristically increases backjump potential.

Our combined abduction step and choice heuristic, called *trail-guided choice*, is abstraction-independent and is both fair and aims to increase backjump potential. Recall that AFIRSTUIP applies underapproximate abduction and upwards interpolation to generalize an element q which represents the meet over a prefix $\mathsf{trail}[1] \dots \mathsf{trail}[j]$ of the trail with respect to a deduction result d.

We first weaken q by removing all bounds over variables that 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.


Step 1: Attempt weakening $x \preceq 0.4$ to $x \preceq 5.2$
$\cdots$
$x \preceq 5.2$
$\cdots$
$y \succeq 1.3$
$\cdots$
$y \succeq 7.2$
$x \preceq 0.4$
Step 2: Attempt weakening $y \succeq 7.2$ to $y \succeq 1.3$

When an element $\mathsf{trail}[j]$ is encountered such that $\mathsf{trail}[j]$ is a bound on a variable x that is used in q (that is, $q \sqsubseteq \mathsf{trail}[j]$), we weaken q by replacing the bound $\mathsf{trail}[j]$ with the most recent trail element more general than $\mathsf{trail}[j]$. If no such element exists, we weaken q by removing the bound $\mathsf{trail}[j]$ altogether. If the resulting element is still strong enough to deduce d, we store the resulting element as a candidate generalization and continue stepping backwards through the trail, in order to find more general reasons. If not, we undo the weakening and we use binary search to find an element that is weaker than q, stronger than $\mathsf{trail}[j]$ and sufficiently strong to deduce d. If the search takes too long, we abort the process after a fixed number search steps and return the most general element found so far that is still sufficient to deduce d.

We have experimented with adjusting the strength of generalization by varying the maximum number of binary search steps. Our experiments, reported in Section 6.3.1, suggest that keeping this number low gives the best trade-off between cost and quality of abductive generalizations. However, we believe that more sophisticated strategies might provide further benefits.

In addition to generalization by search, we also use generalization transformers in cases where it is trivial to compute them. For example, given a constraint $x = y$ and an element $d = \langle y \preceq 5.0 \rangle$, we immediately generalize an element $\langle x \preceq 0.0 \rangle$ to $\langle x \preceq 5.0 \rangle$.

6.3 Experiments

The FP-ACDCL tool was evaluated on 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 numerical computations and (iii) feasibility of systems of inequalities over bounded floating-point variables.

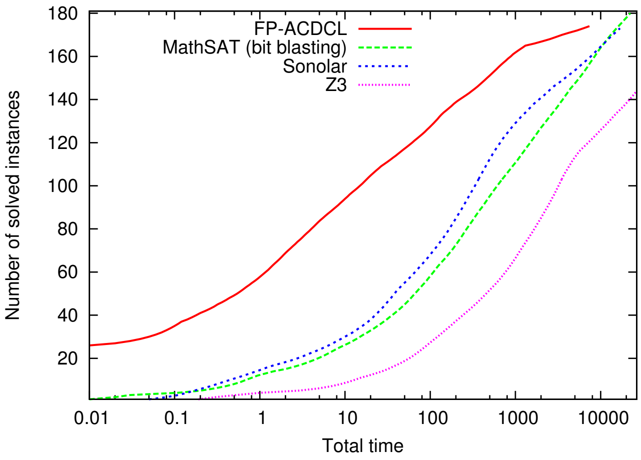


Figure 6.4: Comparison of FP-ACDCL against various SMT solvers using bit-vector encoding of floating-point operations.

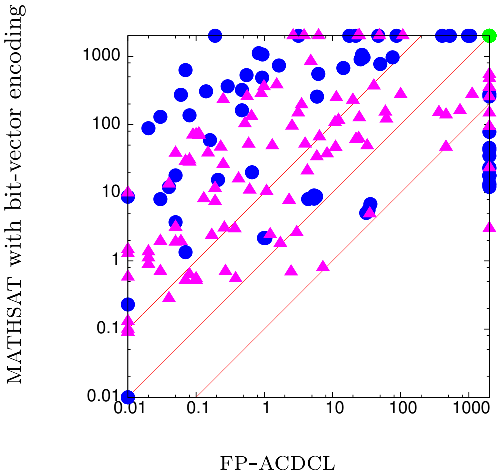


Figure 6.5: Detailed comparison of FP-ACDCL against MATHSAT using using bit-vector encoding of floating point operations. Circles indicate unsatisfiable instances, triangles satisfiable ones. Points on the borders indicate timeouts (1200 s).

The first two sets originate from verification problems on C programs performing numerical computations, whereas the instances in the third set are randomly generated. Our benchmarks and the FP-ACDCL tool are available at `http://es.fbk.eu/people/griggio/papers/FMSD-fmcad12.tar.bz2`. 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.

Comparison with bit-vector encodings

In the first set of experiments, we have compared FP-ACDCL against all SMT solvers supporting FPA that we were aware of, namely Z3 [55], SONOLAR [113] and MATHSAT [33]. All three solvers solve FPA via encoding into propositional logic. For each tool, we used the default options.

The results of the comparison are reported in Figures 6.4 and 6.5. The plot in Figure 6.4 shows the number of successfully solved instances for each system (on the y axis) and the total time needed for solving them (on the x axis). FP-ACDCL clearly outperforms Z3 in

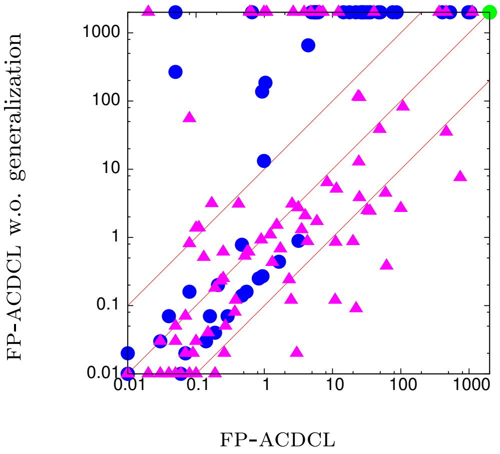


Figure 6.6: Effects of generalizations in conflict analysis. Circles indicate unsatisfiable instances, triangles satisfiable ones. Points on the borders indicate timeouts (1200 s).

terms of instances solved and in total execution time. FP-ACDCL solves the same number of instances as SONOLAR, but is faster overall. Compared to MATHSAT, the results are mixed, as can be seen in the scatter plot of Figure 6.5. FP-ACDCL is much faster than MATHSAT in the majority of instances that both tools can solve, but MATHSAT seems to still have an advantage in terms of scalability, solving overall 6 more instances than FP-ACDCL.

There are some instances that turn out to be relatively easy for solvers based on bit-blasting, but cannot be solved by FP-ACDCL. 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. 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.

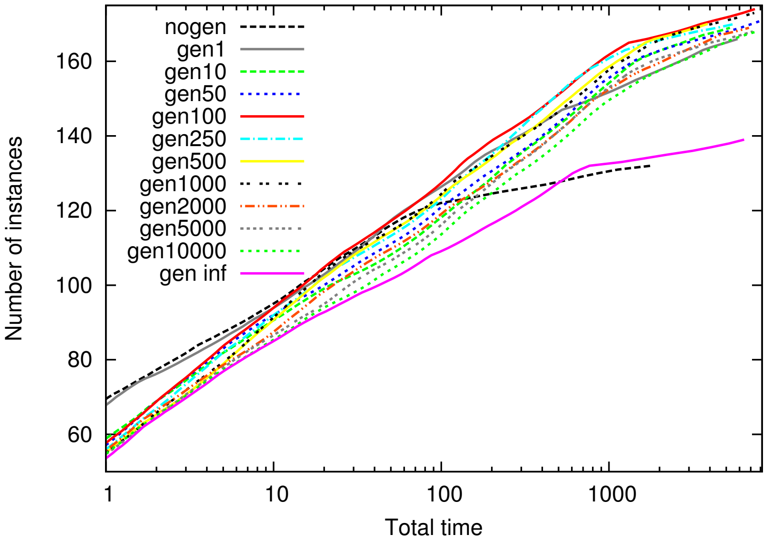
To handle such cases, our framework can be instantiated with abstract domains or combinations of abstract domains [43] that are better suited to the problems under analysis. Moreover, bit-blasting approaches can take advantage of highly efficient SAT solvers, which are the result of years of development, optimization and fine tuning, whereas our FP-ACDCL tool should still be considered a prototype.

6.3.1 Impact of Generalization

We present a second set of experiments that is aimed at evaluating the impact of our variable selection and generalization techniques. In order to evaluate our conflict analysis procedure, we first compare FP-ACDCL without abductive generalization with the default configuration of FP-ACDCL. FP-ACDCL without generalization does not use abductive transformers and implements a naive lifting of the propositional conflict analysis algorithm to lattices.

The results are summarized in Figure 6.6. From the plot, we can clearly see that generalization is crucial for the performance of FP-ACDCL. Every instance that can be solved without generalization can also be solved with generalization, but the configuration with generalization solves 42 more instances before the timeout. However, there are a number of instances for which performance degrades when using generalizations, sometimes significantly. This can be explained by observing that (i) generalizations come at a runtime cost, which sometimes induces a non-negligible overhead and (ii) the performance degradation occurs on satisfiable instances (shown in a lighter color in the plots), for which it is known that the behavior of CDCL-based approaches is typically unstable (even in the propositional case).

Next, we have performed a more in-depth evaluation of the effects of using different



Configuration	Num. instances	Total time
gen100	175	7409
gen1000	174	7348
gen50	172	8181
gen500	171	3876
gen250	171	5576
gen10	170	5263
gen2000	170	6827
gen5000	169	7204
gen10000	169	7574
gen1	167	5842
gen inf	140	6390
nogen	133	1783

Figure 6.7: Comparison of different strategies for conflict generalization in FP-ACDCL.

generalization strategies. In particular, we evaluate the impact of picking different cutoff values for the number of generalization attempts during the trail-guided search procedure.

The results are shown in Figure 6.7. In the figure, ‘genX’ labels the configuration with a maximum number X of generalization attempts, ‘nogen’ labels the configuration where no generalization is performed and ‘gen inf’ the one which does not impose any bound on the number of generalization attempts. From the plot, we can see that the strategies that impose a limit to the number of generalization attempts outperform both the unconstrained strategy and the naive one that uses no generalization. The results provide a powerful illustration of the trade-off between generalization quality and computational effort. For our benchmarks, the sweet spot lies at 100 generalization attempts per iteration of trail-guided search.

Finally, we have performed a further set of experiments in order to evaluate the impact of fairness in the variable selection heuristics for branching and conflict generalization. We have compared the default version of FP-ACDCL with a version in which variables are selected randomly. The results, shown in Figure 6.8, demonstrate that fairness is an important factor for the performance of FP-ACDCL. The use of fair selection strategies yields 23 more solved instances before the timeout.

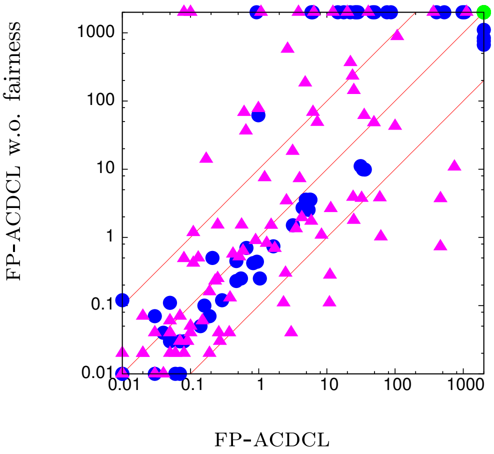


Figure 6.8: Effects of fairness in branching heuristic. Circles indicate unsatisfiable instances, triangles satisfiable ones. Points on the borders indicate timeouts (1200 s).

6.4 Related Work

This section briefly surveys work in interactive theorem proving, abstract interpretation and decision procedures that targets floating-point problems. For a discussion of the special difficulties that arise in reasoning about floating-point computations, see [133].

Theorem Proving Various floating-point axiomatizations and libraries for interactive theorem provers exist [51, 125, 131, 92]. Theorem provers have been applied extensively to proving properties of floating-point algorithms or hardware [93, 95, 96, 1, 105, 136, 154, 94]. 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., [5]. A preliminary attempt to integrate such techniques with SMT solvers has recently been proposed in [36].

Abstract Interpretation Analysis of floating-point computations has also been extensively studied in abstract interpretation. An approach to specifying floating-point properties of programs was proposed in [19]. A number of general purpose abstract domains have been constructed for the analysis of floating-point programs [129, 29, 30, 100, 28, 31]. In addition, specialized approaches exist which target-specific problem domains such as numerical filters [63, 132]. The approaches discussed so far mainly aim at establishing the result of a floating-point computation. An orthogonal line of research is to analyze the deviation of a floating-point computation from its real counterpart by studying the propagation of rounding errors [73, 67]. Case studies for this approach are given in [75, 57]. Abstract interpretation techniques provide a soundness guarantee, but may yield imprecise results.

Decision Procedures In the area of decision procedures, the study of floating-point problems is relatively scarce. Work in constraint programming [128] shows how approximation with real numbers can be used to soundly restrict the scope of floating-point values. In [20], 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 [153]. Recent decision procedures for floating-point logic are based on propositional encodings of floating-point constraints. Examples of this approach are implemented in MATHSAT [33], CBMC [34], Z3 [55] and SONOLAR [113]. 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 [27], which uses a combination of over- and underapproximate propositional abstractions in order to keep the size of the search space as small as possible.

The work in [64] presents a solver that is architecturally similar to ours and instantiates a version of the CDCL algorithm over integer intervals. Compared to [64], our solver targets a different logic, uses a more sophisticated conflict analysis procedure based on underapproximation of abduction transformers and is obtained as the result of interfacing a generic ACDCL implementation with a specific abstract domain.

Chapter 7

Learning Program Analyses via ACDCL

In the previous chapter, we have shown how the ACDCL framework can be used to instantiate CDCL-style decision procedures from abstract domains. The algebraic nature of ACDCL allows us to apply it to different problem domains. In this chapter, we instantiate ACDCL to refine abstract program analyses using value-based partitions both in theory and in practice.

It is important to note that this chapter presents one possible application of ACDCL to program analysis rather than an exhaustive framework that covers all possible analyses. We focus on using ACDCL as a partition-based refinement strategy that operates over abstract values at control locations. Given appropriately defined trace-based abstractions of program semantics, it is possible to instantiate ACDCL to offer a wider range of refinements beyond those considered in this chapter. For example, given a sufficiently expressive abstract domain, ACDCL can partition analyses based on control-flow paths. It is further important to note that the work in this chapter is intended as a theoretical demonstration that ACDCL can, in principle, be instantiated over program analyses. Further work is needed to discover *efficient* instantiations. At the end of this chapter, we present prototype that implements a radically simplified variant of the framework.

Motivation Since abstract-interpretation-based static analysis is sound by design, attaining sufficient precision to be complete on a given class of problems is the primary goal of designing novel analysis procedures. A source of imprecision is the inability of common abstract domains to precisely reason about disjunction. The culprit, is the use of abstract domains that are not disjunctively complete for efficiency reason.

One solution is to use distributive abstract domains such as those used in model checking. This typically comes at a high computational cost. Another option, is to use analysis refinement procedures, which adaptively adjust the precision of the analysis depending on the system and specification.

Case analysis may be understood as a refinement for non-distributive lattices. Consider two abstract elements a, b which together represent the semantics of a system under analysis and an abstract representation c of a set of behaviors that violates the specification. A standard way to prove that the system satisfies the specification is to check whether $(a \sqcup b) \sqcap c$ represents the empty set. In a distributive lattice, the following equality holds.

$$(a \sqcup b) \sqcap c = (a \sqcap c) \sqcup (b \sqcap c)$$

In terms of our example, this means that no precision is gained by checking c separately against a and b compared to checking it against $a \sqcup b$. in other words, case analysis brings no advantage over a monolithic analysis. In a non-distributive lattice, the following holds.

$$(a \sqcup b) \sqcap c \sqsupseteq (a \sqcap c) \sqcup (b \sqcap c)$$

So separately checking a and b against c may yield more precision than checking $a \sqcup b$. This is the essence of case analysis.

Example 7.0.1. *As an example, consider the following program.*

$$\texttt{if}(x > 0)\ y := 1;\ \texttt{else}\ y := -1;\ \texttt{assert}(y \neq 0);$$

In the interval lattice we approximate the value of y on the first branch using the element $a = [1, 1]$ and on the second with $b = [-1, -1]$. The disallowed value may be expressed by the element $c = [0, 0]$. A typical abstract analysis will check whether c is possible by checking whether the expression $(a \sqcup b) \sqcap c$ is bottom. We get $([1, 1] \sqcup [-1, -1]) \sqcap [0, 0] = [-1, 1] \sqcap [0, 0] = [0, 0]$. On the other hand, computing $(a \sqcap c) \sqcup (b \sqcap c)$, which corresponds to checking the property separately on each branch, yields $([1, 1] \sqcap [0, 0]) \sqcup ([-1, -1] \sqcap [0, 0]) = \bot \sqcup \bot = \bot$.

The lack of distributivity can also lead to imprecision in transformer computations.

Example 7.0.2. *Consider the following program.*

$$\texttt{if}(x = 0 \vee x = 1)\{\ x := x * 2;\ \texttt{assert}(x \neq 1);\ \}$$

Upon entering the branch, the interval $[0, 1]$ precisely represents the possible values of x. The value of x after multiplication is either 0 or 2, which can be presented abstractly by $[0, 0] \sqcup [2, 2] = [0, 2]$. We can check whether the assertion can be violated by computing the meet with the disallowed value $[1, 1]$. The check fails since $([0, 0] \sqcup [2, 2]) \sqcap [1, 1]$ is equal to $[1, 1]$. To increase precision, we may check each assignment to x separately against the disallowed value and compute $([0, 0] \sqcap [1, 1]) \sqcup ([2, 2] \sqcap [1, 1])$ which yields $\bot$.

As we have seen from the two examples above, the use of non-distributive abstractions leads to imprecision. This imprecision can be recovered by a variety of means. We focus on case analysis.

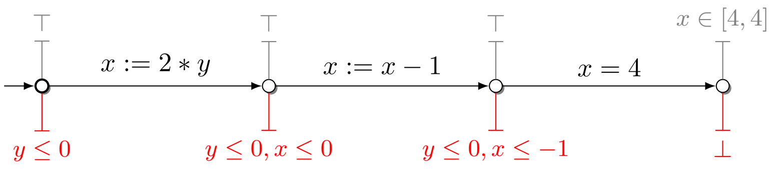
An important practical consideration is how case splitting should be applied in order to get sufficient precision. If the set of cases is too coarse, the resulting analysis is too imprecise. Ff too many cases are considered, the analysis is too expensive. As we will see, the ACDCL algorithm provides an answer to this question and systematically and intelligently decomposes the search space into a set of regions that is just precise enough to prove the property under consideration.

The following example illustrates this.

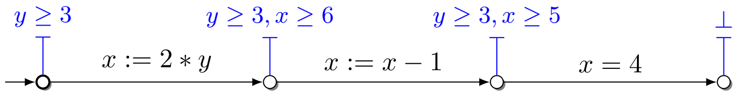
Example 7.0.3 (Value-Based Refinement with ACDCL)**.** *Consider the following program over the integer variables x and y.*

$$x := 2 * y;\ x := x - 1;\ \texttt{assert}(x \neq 4);$$

The results of interval analysis of the above program are shown as annotations situated above the control-flow nodes in the CFG below. The last location represents a special error location.



The error location is unreachable, but interval analysis is too weak to show this. ACDCL applies downwards extrapolation (i.e., a decision) and restricts the initial location by assigning it the abstract value $y \leq 0$. Using this restriction, interval analysis proves the error location unreachable, as shown in the node annotations below the CFG, which constitutes a *conflict. Analyzing the conflict, we find that the decision can be weakened to $y \leq 2$ and still produce a similar conflict. The program is safe, as long as initially $y \leq 2$ holds.* ACDCL *backtracks and explores the remaining case where, initially, $y \geq 3$ hold. This yields the following abstract analysis which proves the program safe.*



Since the program is safe whenever $y \geq 3$ or $y \leq 2$ the program is always safe.

In the above example, ACDCL finds the case split $\{\langle y \geq 3\rangle, \langle y \leq 2\rangle\}$ which is sufficiently precise to prove the property, while still requiring only two analysis steps. A naive case-splitting procedure may have analyzed an unnecessarily large set of cases, e.g., each assignment $\langle y{:}[c, c]\rangle$ of y to a constant c, or a set of cases that fails to prove the property, e.g., $\{\langle y \geq 1\rangle, \langle y \leq 0\rangle\}$.

This chapter provides a framework for instantiating ACDCL over program analyses, which formalizes the above example.

Outline We first recall some background on program analysis using the static analysis equation in Section 7.1. Section 7.2 casts safety checking over CFGs as an instance of the bottom-everywhere problem. Section 7.3 shows how ACDCL can be instantiated over static analysis equations by defining a class of abstract domains that have the required complementation properties. In Section 7.4, we discuss CDFPL, a program analyzer that incorporates learning by instantiating the ACDCL framework, and we present experimental results. Section 7.5 discusses work related to program analysis and learning.

7.1 Solving the Static Analysis Equation

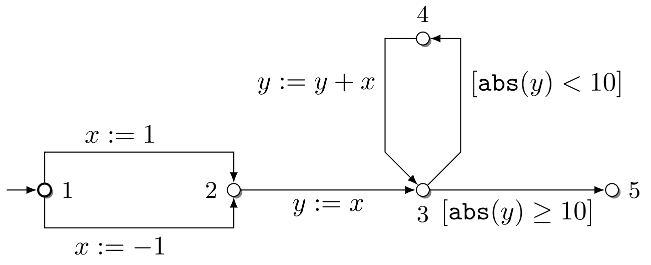
We first introduce some background material on program analysis over CFGs using the static analysis equation. Later sections will show how ACDCL can be applied to reason about an abstraction of this equation. Fix a CFG $G = (N, E, n_I, st)$ with a special error location $\lightning \in N$. Recall that the concrete trace semantics of G are defined with respect to a set of memory states Ω as the set of well-formed traces of the transition system $(\Sigma_G, \rightarrow_G, I_G)$ of G, where $\Sigma_G = N \times \Omega$.

The data-flow between individual control-flow nodes can be expressed using the *static analysis equation*, which introduces a set-valued variable S_n for each location $n \in N$ that takes values in $\mathcal{P}(\Omega)$. These set variables are related using transformers associated with individual program statements to express data flow along a CFG edge and joins to represent information flow along merging branches. An example of a CFG and its static analysis equation is shown in Figure 7.1. Program analyzers often approximate the semantics of G by overapproximating a vectorial fixed point of the static analysis equation of a program, rather than by operating on the unstructured transition system $(\Sigma_G, \rightarrow_G, I_G)$.

Building the static analysis equation is a form of trace partitioning ([151, 119], see Section 4.2.2) of the state semantics constructed with respect to the set of control locations. The following definition captures the resulting lattice.

Definition 7.1.1 (Concrete Control-Location Lattice)**.** Consider a CFG $G = (N, E, n_I, st)$. The *concrete control-location lattice* w.r.t. G is the complete lattice $\mathcal{N}(G)$ defined as follows.

$$\mathcal{N}(G) \mathrel{\hat{=}} N \rightarrow \mathcal{P}(\Omega) \qquad g \sqsubseteq h \text{ exactly if } h = \forall n \in N.\ g(n) \subseteq h(n)$$
$$g \sqcap h \mathrel{\hat{=}} x \mapsto g(x) \cap h(x) \quad g \sqcup h \mathrel{\hat{=}} x \mapsto g(x) \cup h(x)$$



$$S_1 = \top \qquad S_4 = post_{[\mathtt{abs}(y)<10]}(S_3)$$
$$S_2 = post_{x:=1}(S_1) \cup post_{x:=-1}(S_1) \qquad S_5 = post_{[\mathtt{abs}(y)\geq 10]}(S_3)$$
$$S_3 = post_{y:=x}(S_2) \cup post_{y:=y+x}(S_4)$$

Figure 7.1: A CFG and its corresponding static analysis equation.

Proposition 7.1.1. *The pair (α_N, γ_N) below forms a Galois connection.*

$$(\mathcal{P}(\Pi), \subseteq) \underset{\alpha_N}{\overset{\gamma_N}{\leftrightarrows}} (\mathcal{N}(G), \sqsubseteq)$$
$$\alpha_N(P)(n) \mathrel{\hat{=}} \{\omega \mid \exists \pi \in P.\ (n, \omega) \in \pi\}$$
$$\gamma_N(g) \mathrel{\hat{=}} \{\pi \in \Pi \mid \forall (n, \omega) \in \pi.\ \omega \in g(n)\}$$

Proof. Monotonicity of both transformers is easy to see from their definition. Consider a set of traces $P \subseteq \Pi$ and a trace $\pi \in P$. Then for every pair $(n, \omega) \in \pi$, we have that $\omega \in \alpha_N(P)(n)$. Therefore, $\pi \in \gamma_N \circ \alpha_N(P)$. It follows that $\gamma_N \circ \alpha_N$ is inflationary.

Now consider a function $g \in \mathcal{N}(G)$, let n be a control location in N and let ω be a memory state in $\alpha_N \circ \gamma_N(g)(n)$. Then (n, ω) is a trace in $\gamma_N(g)$. Therefore, it must hold that $\omega \in g(n)$. It follows that for every control location $n \in N$, it holds that $\alpha_N \circ \gamma_N(g)(n) \subseteq g(n)$. Since $\alpha_N \circ \gamma_N(g)$ is inflationary and $\alpha_N \circ \gamma_N$ is deflationary, it follows from Proposition 2.3.4 that the pair (α_N, γ_N) forms a Galois connection. □

Recall that each program statement s is associated with a transition relation $T_s^\Omega \subseteq \Omega \times \Omega$ and therefore also a state transformer $post_{T_s^\Omega}$. We can therefore associate with each edge (n_1, n_2) of the CFG a postcondition transformer.

$$post_{n_1,n_2}(M) \mathrel{\hat{=}} post_{T^\Omega_{st(n_1,n_2)}}(M) = \{\omega \mid \exists \omega' \in M.\ (\omega', \omega) \in T^\Omega_{st(n_1,n_2)}\}$$

In a similar way, we define an existential precondition $pre_{n_1,n_2} \mathrel{\hat{=}} pre_{T^\Omega_{st(n_1,n_2)}}$. The static analysis equation then corresponds to a transformer for the concrete control-location lattice.

Definition 7.1.2 (Control-Location Transformers)**.**

$$post_G, pre_G : \mathcal{N}(G) \to \mathcal{N}(G)$$
$$post_G(g)(n) \mathrel{\hat{=}} \bigcup_{(n',n) \in E} post_{n',n} \circ g(n') \qquad pre_G(g)(n) \mathrel{\hat{=}} \bigcup_{(n,n') \in E} pre_{n,n'} \circ g(n')$$

Proposition 7.1.2. *The transformers $post_G$ and pre_G soundly approximate, respectively, $tpost_G$ and $tpre_G$.*

We omit the proof since it is standard.

Abstracting the Static Analysis Equation

Analyzing a program by computing a fixed point over the static analysis equation is a popular abstract interpretation technique. The advantage of this approach compared to direct approximation of the trace semantics is that only abstractions for the concrete domain of memory states $\mathcal{P}(\Omega)$ need to be provided. Many popular abstract program analyses, such as constant analysis or interval, octagonal [130] or polyhedral [130] analyses follow this approach. The following definition formalizes the abstract domain that results when the static analysis equation is approximated using a memory state abstraction.

Definition 7.1.3 (Abstract Control-Location Lattice)**.** Let $G = (N, E, n_I, st)$ be a CFG and A an abstraction over memory states Ω with $(\mathcal{P}(\Omega), \subseteq) \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} (A, \sqsubseteq_A)$. The *abstract control-location lattice* w.r.t. G and A is the complete lattice $\mathcal{N}_A(G)$ defined as follows.

$$\mathcal{N}_A(G) \mathrel{\hat{=}} N \to A \qquad g \sqsubseteq h \text{ exactly if } h = \forall n \in N.\ g(n) \sqsubseteq_A h(n)$$
$$g \sqcap h \mathrel{\hat{=}} x \mapsto g(x) \sqcap h(x) \quad g \sqcup h \mathrel{\hat{=}} x \mapsto g(x) \sqcup h(x)$$

Proposition 7.1.3. *The pair $(\alpha_{N,A}, \gamma_{N,A})$ below forms a Galois connection.*

$$(\mathcal{N}(G), \sqsubseteq) \underset{\alpha_{N,A}}{\overset{\gamma_{N,A}}{\leftrightarrows}} (\mathcal{N}_A(G), \sqsubseteq)$$
$$\alpha_{N,A}(g) \mathrel{\hat{=}} \alpha_A \circ g \quad \gamma_{N,A}(h) \mathrel{\hat{=}} \gamma_A \circ h$$

By composition, we get a Galois connection to the concrete trace lattice.

$$(\mathcal{P}(\Pi), \subseteq) \underset{\alpha \hat{=} \alpha_{N,A} \circ \alpha_N}{\overset{\gamma \hat{=} \gamma_N \circ \gamma_{N,A}}{\leftrightarrows}} (\mathcal{N}_A(G), \sqsubseteq)$$

We can obtain abstract transformers for the lattice $\mathcal{N}_A(G)$ in a straightforward manner from abstract state transformers over A.

Definition 7.1.4 (Abstract Control-Location Transformers)**.** For every edge $(n, n') \in E$, let $apost_{n,n'}$ and $apre_{n,n'}$ in $A \to A$ be sound overapproximations of $post_{n,n'}$ and $pre_{n,n'}$, respectively. Then the *abstract control-location transformers* $apost_G, apre_G : \mathcal{N}_A(G) \to \mathcal{N}_A(G)$ are defined as follows.

$$apost_G(g)(n) \mathrel{\hat{=}} \bigsqcup_{(n',n) \in E} apost_{n,n'} \circ g(n')$$
$$apre_G(g)(n) \mathrel{\hat{=}} \bigsqcup_{(n,n') \in E} apre_{n,n'} \circ g(n')$$

Proposition 7.1.4. *The transformers $apost_G$ and $apre_G$ soundly approximate $post_G$ and pre_G, respectvely.*

Again, we omit the proof since it is standard.

We give an example of a fixed point iteration for $apost_G$ over the abstract control location abstraction instantiated over interval assignments.

Example 7.1.1. *We show the first couple of iterates of the upwards iteration sequence over $apost_G$ for the* CFG *G depicted in Figure 7.1. We abstract memory states using the abstract domain of interval assignments $Vars \to Itv$. We write elements in $\mathcal{N}_{Vars \to Itv}(G)$ as vectors, where the ith component denotes the value of the node i.*

$$\underbrace{\begin{pmatrix}\bot\\\bot\\\bot\\\bot\\\bot\end{pmatrix}}_{F_0}\underbrace{\begin{pmatrix}\top\\\bot\\\bot\\\bot\\\bot\end{pmatrix}}_{F_1}\underbrace{\begin{pmatrix}\top\\\langle x{:}[-1,1]\rangle\\\bot\\\bot\\\bot\end{pmatrix}}_{F_2}\underbrace{\begin{pmatrix}\top\\\langle x{:}[-1,1]\rangle\\\langle x,y{:}[-1,1]\rangle\\\bot\\\bot\end{pmatrix}}_{F_3}\underbrace{\begin{pmatrix}\top\\\langle x{:}[-1,1]\rangle\\\langle x,y{:}[-1,1]\rangle\\\langle x,y{:}[-1,1]\rangle\\\bot\end{pmatrix}}_{F_4}\underbrace{\begin{pmatrix}\top\\\langle x{:}[-1,1]\rangle\\\langle x{:}[-1,1],y{:}[-2,2]\rangle\\\langle x{:}[-1,1],y{:}[-1,1]\rangle\\\bot\end{pmatrix}}_{F_5}$$

7.2 CFG Safety as Bottom-Everywhere

We now recall the characterization of safety checking as an instance of the bottom-everywhere problem from Section 3.2.3. We specialize the safety problem and the transformer definitions of Section 3.2.3 to check if an error is reachable in a CFG with respect to a special error location $\lightning$. For convenience, we assume that the error location is a *sink node* of the CFG, that is, we assume that there is no edge $(\lightning, n) \in E$, for any $n \in N$.

Definition 7.2.1 (CFG Safety and Counterexamples)**.** Let $G = (N, E, n_I, st)$ be a CFG with a special sink node $\lightning \in N$ called the *error location.* A trace π of G is *safe* if it does not end in the error location. A *counterexample trace* is a trace of G that is well-formed and ends in an error state. A CFG G is *safe* w.r.t. $\lightning$ if all well-formed traces π of G are safe w.r.t. $\lightning$, i.e., if there is no counterexample trace.

Safety can be cast as solving the bottom-everywhere problem over the following bottom-everywhere algebra.

$$
\begin{aligned}
&(\Pi, \subseteq, \cap, \cup, \mathit{utrace}_G, \mathit{strace}_G, \mathit{cutrace}_G, \mathit{cstrace}_G)\\
\mathit{utrace}_G(P) &\mathrel{\hat{=}} \{\pi \in \Pi \mid \pi \in P \land \pi \text{ is well-formed and not safe}\}\\
\mathit{strace}_G(P) &\mathrel{\hat{=}} \{\pi \in \Pi \mid \pi \in P \lor \pi \text{ is not well-formed or safe}\}\\
\mathit{cutrace}_G(P) &\mathrel{\hat{=}} \{\pi \in \Pi \mid \pi \in P \land \pi \text{ is not well-formed or safe}\}\\
\mathit{cstrace}_G(P) &\mathrel{\hat{=}} \{\pi \in \Pi \mid \pi \in P \lor \pi \text{ is well-formed and not safe}\}
\end{aligned}
$$

Proposition 7.2.1. *The tuple* $(\Pi, \subseteq, \cap, \cup, \mathit{utrace}_G, \mathit{strace}_G, \mathit{cutrace}_G, \mathit{cstrace}_G)$ *is a bottom-everywhere algebra. The* CFG G *is safe exactly if* utrace_G *is bottom everywhere.*

Proof. The proof that the structure above is a bottom-everywhere algebra is similar to that of Theorem 3.2.15 and Theorem 3.2.18. The CFG G is safe exactly if there is no well-formed trace that is not safe. This is equivalent to stating that $\mathit{utrace}_G(\Pi) = \emptyset$, which is in turn equivalent to utrace_G being bottom-everywhere. □

7.2.1 Approximating the Unsafe Trace Transformer

We now show how the control-location transformers defined above may be used to compute approximations of the unsafe trace transformers. This is the first step towards instantiating counterwitness search and conflict generalization. The notion of a counterwitness, specialized to program safety checking, corresponds to the search for an abstract representation of a counterexample trace.

The set of unsafe traces can be characterized as the intersection of two sets, both of which have fixed point characterizations: (i) the set of traces that start in the initial location and respect the transition relation and (ii) the set of traces that end in the error location and respect the transition relation. We denote by Ξ the set of states $\{(n, \omega) \mid n \neq \lightning\}$ that are not at the error location. Also recall that I_G is the set of states $\{(n_I, \omega) \mid \omega \in \Omega\}$ that are at the initial location.

Proposition 7.2.2. *The unsafe trace transformer can be characterized as follows.*

$$\mathit{utrace}_G(P) = P \cap (\mathsf{lfp}\, X.\ I_G \cup \mathit{tpost}_{\rightarrow_G}(X)) \cap (\mathsf{lfp}\, X.\ \neg\Xi \cup \mathit{tpre}_{\rightarrow_G}(X))$$

Proof. The set $P_1 \mathrel{\hat{=}} \mathsf{lfp}\, X.\ I_G \cup \mathit{tpost}_{\rightarrow_G}(X)$ is the set of well-formed traces, and the set $P_2 \mathrel{\hat{=}} \mathsf{lfp}\, X.\ \neg\Xi \cup \mathit{tpre}_{\rightarrow_G}(X)$ is the set of traces that respect the transition relation and end in the error location. We skip the proofs for these characterizations since they are standard. The intersection $P_1 \cap P_2$ is the set of traces that are well-formed and end in the error location. It follows that $P \cap P_1 \cap P_2$ is equal to $\mathit{utrace}_G(P)$. □

Note that the first fixed point expression in the proposition above represents a forward analysis and the second a backward analysis. In the abstract, communication between forwards and backwards analysis can provide strictly better results than applying either in isolation [46]. We now provide a second characterization, in which the argument set of traces P is accessed inside the fixed point expression. This allows an abstraction to take into account the candidate set of traces while computing the two fixed points, which leads to more precision.

Proposition 7.2.3. *The unsafe trace transformer can be characterized as follows.*

$$\begin{aligned} \mathit{utrace}_G(P) = P \cap [&\mathsf{lfp}\, X.\ (\mathit{prefixcl}(P) \cap (I_G \cup \mathit{tpost}_{\to_G}(X)))] \cap \\ &[\mathsf{lfp}\, X.\ (\mathit{suffixcl}(P) \cap (\neg\Xi \cup \mathit{tpre}_{\to_G}(X)))] \end{aligned}$$

The sets $\mathit{prefixcl}(P)$ and $\mathit{suffixcl}(P)$ are defined as follows.

$$\begin{aligned} \mathit{prefixcl}(P) &\mathrel{\hat{=}} \{\pi \mid \exists \pi' \in P.\ \pi \text{ is a prefix of } \pi'\} \\ \mathit{suffixcl}(P) &\mathrel{\hat{=}} \{\pi \mid \exists \pi' \in P.\ \pi \text{ is a suffix of } \pi'\} \end{aligned}$$

Proof. We show that the right-hand side of the equality above is equivalent to $P \cap (\mathsf{lfp}\, X.\ I_G \cup \mathit{tpost}_{\to_G}(X)) \cap (\mathsf{lfp}\, X.\ \neg\Xi \cup \mathit{tpre}_{\to_G}(X))$. The result then follows from Proposition 7.2.2. We first show that the following equality holds.

$$\text{(i)} \qquad P \cap (\mathsf{lfp}\, X.\ I_G \cup \mathit{tpost}_{\to_G}(X)) = P \cap \mathsf{lfp}\, X.\ \mathit{prefixcl}(P) \cap (I_G \cup \mathit{tpost}_{\to_G}(X))$$

It is easy to see that the inclusion $\supseteq$ holds, we therefore only show the direction for $\subseteq$. Consider an element $\pi \in P \cap (\mathsf{lfp}\, X.\ I_G \cup \mathit{tpost}_{\to_G}(X))$. Then π is a well-formed, initialized trace, and so are all its prefixes. Consider the function, $g(X) \mathrel{\hat{=}} \mathit{prefixcl}(P) \cap (I \cup \mathit{tpost}_{\to_G}(X))$. We have that $\pi_1 \in \mathit{prefixcl}(P)$ and $\pi_1 \in I$, therefore $\pi_1 \in g(\emptyset)$. Now for an inductive argument, assume for $i < |\pi|$ that $\pi_1 \ldots \pi_i \in g^i(\emptyset)$. It is easy to see that $\pi_1 \ldots \pi_{i+1} \in g^{i+1}(\emptyset)$. Therefore, we have that $\pi \in g^k(\emptyset)$, where $k = |\pi|$. The function g is completely additive, and therefore also Scott-continuous. It follows from Kleene's fixed point theorem (Theorem 2.3.10) that $\mathsf{lfp}\, X.g(X)$ can be expressed as $\bigcup_{i \in \mathbb{N}_0} \{g^i(\emptyset)\}$. Since $\pi \in g^k(\emptyset)$ we then have that $\pi \in \mathsf{lfp}\, X.g(X)$ and from earlier, we know that $\pi \in P$. It follows that $\pi \in P \cap \mathsf{lfp}\, X.\ \mathit{prefixcl}(P) \cap (I_G \cup \mathit{tpost}_{\to_G}(X))$, which concludes the proof that the equality (i) holds.

By a symmetric argument one can prove the following equality.

$$\text{(ii)} \qquad P \cap \mathsf{lfp}\, X.\ \neg\Xi \cup \mathit{tpre}_{\to_G}(X) = P \cap \mathsf{lfp}\, X.\ (\mathit{suffixcl}(P) \cap (\neg\Xi \cup \mathit{tpre}_{\to_G}(X)))$$

From the equalities (i) and (ii) we have that

$$\begin{aligned} P \cap [&\mathsf{lfp}\, X.\ (\mathit{prefixcl}(P) \cap (I_G \cup \mathit{tpost}_{\to_G}(X)))] \cap \\ &[\mathsf{lfp}\, X.\ (\mathit{suffixcl}(P) \cap (\neg\Xi \cup \mathit{tpre}_{\to_G}(X)))] \\ = P \cap (&\mathsf{lfp}\, X.\ I_G \cup \mathit{tpost}_{\to_G}(X)) \cap (\mathsf{lfp}\, X.\ \neg\Xi \cup \mathit{tpre}_{\to_G}(X)) \end{aligned}$$

By Proposition 7.2.2, the above is then equivalent to utrace_G, which completes the proof. □

The following transformers abstract the characterization above by computing vectorial forward and backwards fixed points in the static analysis equation.

Definition 7.2.2 (Global Unsafe-Trace Approximation)**.** We define the following transformers.

$$\begin{aligned} \mathit{autrace}_G, \mathit{autrace}_G^{\rightarrow}, \mathit{autrace}_G^{\leftarrow} &: \mathcal{N}_A(G) \to \mathcal{N}_A(G) \\ \mathit{autrace}_G^{\rightarrow}(g) &\mathrel{\hat{=}} \mathsf{lfp}\, X.\ g \sqcap (g_I \sqcup \mathit{apost}_G(X)) \\ \mathit{autrace}_G^{\leftarrow}(g) &\mathrel{\hat{=}} \mathsf{lfp}\, X.\ g \sqcap (g_{\neg\Xi} \sqcup \mathit{apre}_G(X)) \\ \mathit{autrace}_G(g) &\mathrel{\hat{=}} \mathit{autrace}_G^{\rightarrow}(g) \sqcap \mathit{autrace}_G^{\leftarrow}(g) \end{aligned}$$

The functions g_I and $g_{\neg\Xi}$ are defined as follows.

$$g_I(n) \mathrel{\hat{=}} \begin{cases} \top & \text{if } n = n_I \\ \bot & \text{otherwise} \end{cases} \qquad g_{\neg\Xi} \mathrel{\hat{=}} \begin{cases} \top & \text{if } n = \lightning \\ \bot & \text{otherwise} \end{cases}$$

Proposition 7.2.4. *The transformers $autrace^{\rightarrow}_{G}$, $autrace^{\leftarrow}_{G}$ and $autrace_G$ soundly approximate $utrace_G$.*

Proof. We show the case of $autrace^{\rightarrow}_{G}$. The proof for $autrace^{\leftarrow}_{G}$ is similar. Soundness of $autrace_G$ then follows from soundness of $autrace^{\rightarrow}_{G}$ and $autrace^{\leftarrow}_{G}$.

We prove that $autrace^{\rightarrow}_{G}$ soundly overapproximates $utrace_G$. Recall that $\mathcal{N}_A(G)$ abstracts the concrete via the Galois connection $(\gamma, \alpha) \mathrel{\hat{=}} (\gamma_N \circ \gamma_{N,A}, \alpha_{N,A} \circ \alpha_N)$. Note that $g_I = \alpha(I_G)$ and that $apost_G$ soundly approximates $tpost_{\rightarrow_G}$. Therefore, for any $g \in \mathcal{N}_A(G)$, the abstract transformer $X \mapsto g \sqcap (g_I \sqcup apost_G(X))$ soundly approximates the concrete transformer $X \mapsto \gamma(g) \cap (g_I \cup tpost_{\rightarrow_G}(X))$. Note that $\gamma(g)$ is prefix-closed, and therefore $prefixcl(\gamma(g)) = \gamma(g)$. Hence, the abstract transformer above also abstracts the concrete transformer $X \mapsto prefixcl(\gamma(g)) \cap (g_I \cup tpost_{\rightarrow_G}(X))$. From fixed point transfer, it follows that $\mathsf{lfp}\, X.\, g \sqcap (g_I \sqcup apost_G(X))$ soundly approximates $\mathsf{lfp}\, X.\, prefixcl(\gamma(g)) \cap (g_I \cup tpost_{\rightarrow_G}(X))$. From the characterization in Proposition 7.2.3 we then get that the abstract transformer soundly approximates $utrace_G$. This completes the proof that $autrace^{\rightarrow}_{G}$ overapproximates $utrace_G$. □

The transformers defined above approximate $utrace_G$, which is a lower closure operator. In the abstract, lower closure operators may be approximated by computing a greatest fixed point. Since each of the transformers above is computed using a least fixed point, the result is a nested fixed point computation. This nested computation of a greatest fixed point using forward and backwards analysis based on least fixed points is a well-known technique in program analysis [46].

7.3 ACDCL for Static Analysis Equations

In the preceding section, we have shown that checking if a CFG is safe is an instance of checking if the transformer $utrace_G$ is bottom everywhere and that this transformer can be approximated using classic program analysis techniques. We now describe how ACDCL can be instantiated over the abstract control-location lattice $\mathcal{N}_A(G)$. The requirements for ACDCL are recalled in Figure 7.3.3.

7.3.1 The Straight-Line Abstraction

The first necessary requirement for instantiating ACDCL is a lattice whose meet irreducibles are complementable relative to the transformer $utrace_G$. In this section, we show that the abstract control-location lattice $\mathcal{N}_A(G)$ does not always have complementable meet irreducibles and we define a new abstract lattice which does have this property.

Whether the meet irreducibles of the abstract control-location lattice $\mathcal{N}_A(G)$ are complementable depends on the underlying memory state abstraction A, as well as on the CFG G. The meet irreducibles of $\mathcal{N}_A(G)$ are the elements of the form below.

$$\mathbb{I}_{\sqcap}(\mathcal{N}_A(G)) = \{n \mapsto \begin{cases} m & \text{if } n' = n \\ \top & \text{otherwise} \end{cases} \mid n' \in N \wedge m \in \mathbb{I}_{\sqcap}(A)\}$$

We denote the function that maps the node n to an element a of A as $g_{n \mapsto a}$. Meet irreducibles are of the form $g_{n \mapsto m}$, where m is a meet irreducible of A. As a precise complement of $g_{n \mapsto m}$, we could consider an element $g_{n \mapsto \overline{m}}$, where $\overline{m}$ is the precise complement of m. But as we can see below, the two elements do not complement each other. The

(i) An overapproximate lattice with f-complementable meet irreducibles

$$(A, \sqsubseteq, \sqcap, \sqcup),$$

(ii) a downwards extrapolation

$$\nabla_{\downarrow} : A \times A \to A,$$

(iii) a set of overapproximations of f

$$F \subseteq A \to A,$$

(iv) for each $\mathit{of} \in F$ and $a \in A$, an inflationary underapproximation

$$\mathit{uf}_{\mathit{of},a} : \mathcal{D}(A) \to \mathcal{D}(A)$$

of the transformer $\tilde{f}_a \mathrel{\hat{=}} c \mapsto \tilde{f}(c \cup \gamma(a))$,

(v) an ACDCL (upwards) interpolation

$$\Delta_{\uparrow} : \mathcal{D}(A) \times \mathcal{D}(A) \to A,$$

(vi) a finite meet factorization $\mathit{mdc} : A \to \mathcal{P}(A)$ such that the following holds for all sets $M \in \mathbb{I}_{\sqcap}(A)$ and elements $a \in A$.

$$\text{If } \bigsqcap M \sqsubseteq a \text{ then } \forall q \in \mathit{mdc}(a).\ \exists r \in M.\ r \sqsubseteq q.$$

Figure 7.2: Requirements for applying ACDCL to the bottom-everywhere problem for the algebra $(B, \subseteq, \cap, \cup, f, \tilde{f}, cf, c\tilde{f})$.

concretization of an element $g_{n\mapsto m}$ corresponds to the set of traces such that *all* occurrences of n are contained within $\gamma_A(m)$. Its complement is the set where *some* occurrence of m is not contained within $\gamma_A(m)$, which cannot be precisely represented by an abstract element.

$$\begin{aligned}\neg\gamma(g_{n\mapsto m}) &= \neg\{\pi \in \Pi \mid \forall(n,\omega) \in \pi.\ \omega \in \gamma_A(m)\}\\ &= \{\pi \in \Pi \mid \exists(n,\omega) \in \pi.\ \omega \notin \gamma_A(m)\}\\ \neg\gamma(g_{n\mapsto\overline{m}}) &= \neg\{\pi \in \Pi \mid \forall(n,\omega) \in \pi.\ \omega \notin \gamma_A(m)\}\\ &= \{\pi \in \Pi \mid \exists(n,\omega) \in \pi.\ \omega \in \gamma_A(m)\}\end{aligned}$$

Even though precise complementation is not supported, some meet irreducibles of the lattice support the weaker notion of complementation relative to the unsafe trace transformer.

Proposition 7.3.1. *Let A be a memory state abstraction with complementable meet irreducibles and $G = (N, E, n_I, st)$ be a* CFG *with error location $\lightning$. If for every counterexample trace π the control location $n \in N$ occurs exactly once on path(π) then the abstract elements $g_{n\mapsto m}$ and $g_{n\mapsto\overline{m}}$ of $\mathcal{N}_A(G)$ are precise complements relative to utrace$_G$.*

Proof. We show that under the conditions above, it holds that

$$\mathit{utrace}_G(\gamma(g_{n\mapsto m})) = \mathit{utrace}_G(\neg\gamma(g_{n\mapsto\overline{m}})).$$

Consider a trace $\pi \in \mathit{utrace}_G(\gamma(g_{n\mapsto m}))$. Then π is a counterexample trace such that there is exactly one state of the form (n, ω) on π, and it holds that $\omega \in \gamma_A(m)$ and therefore $\omega \notin \gamma_A(\overline{m})$. Then π is not in $\gamma(g_{n\mapsto\overline{m}})$. Since π is a counterexample trace it follows that $\pi \in \mathit{utrace}_G(\neg\gamma(g_{n\mapsto\overline{m}}))$. Therefore, $\mathit{utrace}_G(\gamma(g_{n\mapsto m})) \subseteq \mathit{utrace}_G(\neg\gamma(g_{n\mapsto\overline{m}}))$ holds. The other direction can be proved in a similar manner. □

In order to obtain a lattice where all meet irreducibles have this complementation property, we construct a simplified abstraction which only stores information about control locations that are guaranteed to occur exactly once on every counterexample trace.

Definition 7.3.1 (Straight-Line Abstraction)**.** Consider a CFG $G = (N, E, n_I, st)$ and let $N_{once} \subseteq N$ be a set of control-locations which occur exactly once on every path $n_I \ldots \lightning$ between the initial and error location of the CFG. Let A be an abstraction of memory states Ω with $(\mathcal{P}(\Omega), \subseteq) \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} (A, \sqsubseteq_A)$. The *straight-line lattice* w.r.t. G and A is the complete lattice $\mathcal{SL}_A(G)$ defined as follows.

$$\mathcal{SL}_A(G) \mathrel{\hat=} N_{once} \to A \qquad g \sqsubseteq h \text{ exactly if } h = \forall n \in N.\ g(n) \sqsubseteq_A h(n)$$
$$g \sqcap h \mathrel{\hat=} x \mapsto g(x) \sqcap h(x) \quad g \sqcup h \mathrel{\hat=} x \mapsto g(x) \sqcup h(x)$$

The straight-line abstraction only tracks control-locations in N_{once} that occur exactly once on every path to the error. An example is shown in Figure 7.3; the nodes in N_{once} are highlighted. This excludes control-locations that are in the body of a loop, which may be visited more than once, and those nodes on control-flow branches that may not occur on a path to the error. We define the following total ordering for two nodes $n_1, n_2 \in N_{once}$.

$$n_1 \preceq n_2 \text{ exactly if } n = n' \text{ or if } n \text{ occurs before } n' \text{ on every path from } n_I \text{ to } \lightning$$

The straight-line abstraction stores less information than the control-location abstraction. The following statement makes this precise.

Proposition 7.3.2. *The functions $(\alpha_{\mathcal{SL}}, \gamma_{\mathcal{SL}})$ below form a Galois connection.*

$$\mathcal{N}_A(G) \underset{\alpha_{\mathcal{SL}}}{\overset{\gamma_{\mathcal{SL}}}{\leftrightarrows}} \mathcal{SL}_A(G)$$

$$\alpha_{\mathcal{SL}}(g)(n) \mathrel{\hat=} g(n) \quad \gamma_{\mathcal{SL}}(h)(n) \mathrel{\hat=} \begin{cases} h(n) & \text{if } n \in N_{once} \\ \top & \text{otherwise} \end{cases}$$

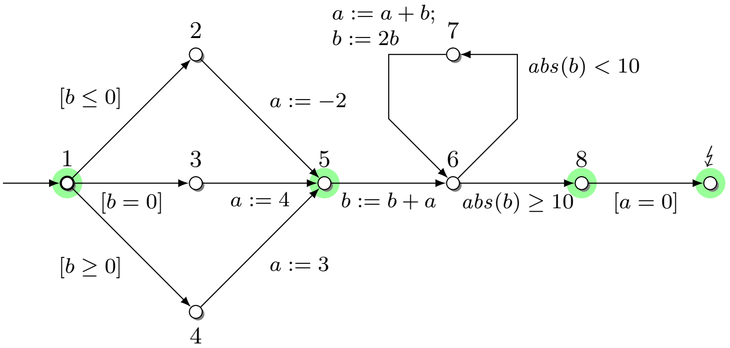


Figure 7.3: Example CFG. Nodes in N_{once} are highlighted.

Proof. It is easy to see that $\alpha_{\mathcal{SL}}$ and $\gamma_{\mathcal{SL}}$ are monotone. The composition $\alpha_{\mathcal{SL}} \circ \gamma_{\mathcal{SL}}$ is the identity function, and $\gamma_{\mathcal{SL}} \circ \alpha_{\mathcal{SL}}$ is inflationary, since it simply replaces the mappings for all control locations not in N_{once} by $\top$. The result follows from Proposition 2.3.4. □

Proposition 7.3.3. *If A has complementable meet irreducibles, then the meet irreducibles of the straight-line abstraction $\mathcal{SL}_A(G)$ have precise complements relative to $utrace_G$.*

Proof. All meet irreducibles $h_{n \mapsto m} \in \mathcal{SL}_A(G)$ concretize to corresponding meet irreducibles $g_{n \mapsto m} \in \mathcal{N}_A(G)$. Since n occurs exactly once on every control-flow path to the error location, it also occurs exactly once on every counterexample trace. The result then follows from Proposition 7.3.1. □

The straight-line abstraction allows instantiation of ACDCL. Transformers over the straight-line abstractions can be computed by first approximating a solution to the static analysis equation and then forgetting information about nodes that are not in N_{once}. The rest of this section formalizes this approach for both counterwitness search and witness search.

7.3.2 Deduction and Decisions in Programs

Recall that instantiation of counterwitness search requires a downwards extrapolation function (which serves as a generalized decision operator), as well as a set of transformers F that soundly approximate the $utrace_G$ transformer. We first give an example run of a counterwitness search procedure on $\mathcal{SL}_A(G)$ and then formalize the necessary transformers.

Example 7.3.1. *Consider the CFG in Figure 7.3. We give an example run of the counterwitness search procedure* `ACDCL-non-⊥`.

The set N_{once} is given by $\{1, 5, 8, ↯\}$. For each (i, j) in $\{(1, 5), (5, 8), (8, ↯)\}$ we use the transformer $autrace^{\rightarrow}_{i,j}$ to deduce information about counterexamples at location j using forward analysis and a transformer $autrace^{\leftarrow}_{i,j}$ to deduce information about counterexamples at location i using backward analysis. We will define these transformers later.

Recall that elements of $\mathcal{SL}_A(G)$ are functions in $N_{once} \to A$. In our case A is the lattice of integer interval environments which $Vars \to Itv$. The meet irreducibles of $\mathcal{SL}_A(G)$ are those functions h in $\mathcal{SL}_A(G)$ which impose a single lower or upper bound of one variable at one location and assign every other location to $\top$. We introduce notation for these meet irreducibles. For a node $m \in N_{once}$ and a program variable x we denote meet irreducibles by $m \mapsto \langle x \Box c \rangle$ where $\Box$ is one of $\{<, >, \leq, \geq\}$. For example, $m \mapsto \langle x \leq c \rangle$ denotes the *following function.*

$$n \mapsto \begin{cases} \top & \text{if } n \neq m \\ \langle x{:}[-\infty, c]\rangle & \text{if } n = m \end{cases}$$

Below, the result of executing `ACDCL-non-⊥` *for a number of steps is shown in the form of an implication graph whose nodes are meet irreducibles.*

$$\begin{tikzcd}
\top \arrow[r, "autrace^{\rightarrow}_{1,5}"] \arrow[dr, "autrace^{\rightarrow}_{1,5}"] \arrow[ddr, "autrace^{\rightarrow}_{8,\lightning}"] \arrow[dddr, "autrace^{\rightarrow}_{8,\lightning}"] & 5 \mapsto \langle a \geq -2\rangle & \\
& 5 \mapsto \langle a \leq 3\rangle & \\
& \lightning \mapsto \langle a \geq 0\rangle \arrow[r, "autrace^{\leftarrow}_{8,\lightning}"] & 8 \mapsto \langle a \geq 0\rangle \\
& \lightning \mapsto \langle a \leq 0\rangle \arrow[r, "autrace^{\leftarrow}_{8,\lightning}"] & 8 \mapsto \langle a \leq 0\rangle
\end{tikzcd}$$

At location 5*, we can determine that the variable a is between* -2 *and* 3 *by running forward analysis. Computing a forward fixed point over the loop yields no new information, so the only other fact that can be discovered is that at the error location* $\lightning$*, the value of a is exactly zero. Backwards analysis can be used to determine that the same must hold at location* 8*.*

The result is not precise enough to exclude the error, so we apply downwards extrapolation to increase precision of the analysis at the cost of soundness. We do this by adding a heuristically chosen meet irreducible to the graph; we pick the element $5 \mapsto \langle b \geq 7\rangle$*. Applying the forwards and backwards transformers yields the following graph.*

$$\begin{tikzcd}
5 \mapsto \langle b \geq 7\rangle \arrow[rr, "autrace^{\rightarrow}_{5,8}"] \arrow[dr, "autrace^{\leftarrow}_{5,1}"'] & & 8 \mapsto \langle a \geq 3\rangle \arrow[r, dashed, "autrace^{\rightarrow}_{8,\lightning}"'] & \text{conflict} \\
& 1 \mapsto \langle b \geq 7\rangle \arrow[r, "autrace^{\rightarrow}_{1,5}"'] \arrow[dr] & 5 \mapsto \langle a \leq 3\rangle \arrow[u, "autrace^{\rightarrow}_{5,8}"'] & \\
& & 5 \mapsto \langle a \geq 3\rangle &
\end{tikzcd}$$

The element $\langle b \geq 7\rangle$ *is propagated via backward analysis to node* 1*. This allows forward analysis to conclude that at node* 5 *the variable a is equal to* 3*. Since both a and b are positive, running forward analysis over the loop concludes that a is greater than* 3 *at node* 8*. This leads to a conflict, since forward analysis finds that if a is greater than* 3 *at node* 8*, the error location* $\lightning$ *is unreachable.*

We now formally specify the initial set of transformers F over the straight-line abstraction used for instantiating ACDCL. We specify two transformers for each successive pair of control nodes in N_{once}, one that uses a strongest postcondition and another which uses the existential precondition. This breaks the global problem of approximating $utrace_G$ down into small subtasks and allows ACDCL to use chaotic iteration strategies and dynamic interleaving of forward and backward analysis.

Definition 7.3.2 (Straight-Line Unsafe-Trace Transformers)**.** For $n_1, n_2 \in N_{once}$ such that n_2 immediately follows n_1 in the ordering $\preceq$, we define the following transformers.

$$autrace^{\rightarrow}_{n_1,n_2}, autrace^{\leftarrow}_{n_1,n_2} : \mathcal{SL}_A(G) \rightarrow \mathcal{SL}_A(G)$$

$$autrace^{\rightarrow}_{n_1,n_2}(h) \mathrel{\hat{=}} n \mapsto \begin{cases} h(n) & \text{if } n \neq n_2 \\ h(n) \sqcap autrace^{\rightarrow}_{G}(g_{n_1 \mapsto h(n_1)})(n) & \text{if } n = n_2 \end{cases}$$

$$autrace^{\leftarrow}_{n_1,n_2}(h) \mathrel{\hat{=}} n \mapsto \begin{cases} h(n) & \text{if } n \neq n_1 \\ h(n) \sqcap autrace^{\leftarrow}_{G}(g_{n_2 \mapsto h(n_2)})(n) & \text{if } n = n_1 \end{cases}$$

$$\text{where } g_{n' \mapsto a} \mathrel{\hat{=}} n \mapsto \begin{cases} a & \text{if } n = n' \\ \top & \text{otherwise} \end{cases}$$

Proposition 7.3.4. *The transformers* $autrace^{\rightarrow}_{n_1,n_2}$ *and* $autrace^{\leftarrow}_{n_1,n_2}$ *soundly approximate* $utrace_G$.

Proof. We prove the statement for $autrace^{\rightarrow}_{n_1,n_2}$, the other proof is similar. $\mathcal{SL}_A(G)$ overapproximates the concrete via $(\alpha_{\mathcal{SL}} \circ \alpha, \gamma \circ \gamma_{\mathcal{SL}})$, where (α, γ) is the Galois connection $\mathcal{P}(\Pi) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{N}_A(G)$. We show the following.

$$utrace_G \circ \gamma \circ \gamma_{\mathcal{SL}} \subseteq \gamma \circ \gamma_{\mathcal{SL}} \circ autrace^{\rightarrow}_{n_1,n_2}$$

Consider a trace π in $utrace_G \circ \gamma \circ \gamma_{\mathcal{SL}}(h)$. Then π is a counterexample trace such that for all $n \in N_{once}$ and $(n, \omega) \in \pi$, it holds that $\omega \in \gamma_A \circ h(n)$. Since $autrace^{\rightarrow}_{G}$ soundly overapproximates $utrace_G$, we then have $autrace_G(g_{n_1 \mapsto h(n_1)})$ represents an overapproximation of the set of counterexample traces π' such that for all $(n_1, \omega) \in \pi'$, $\omega \in \gamma_A \circ h(n_1)$. Therefore $\pi \in autrace_G(g_{n_1 \mapsto h(n_1)})$. It follows that for every state $(n_2, \omega) \in \pi$, it holds that $\omega \in autrace_G(g_{n_1 \mapsto h(n_1)})(n)$. We have that for all $n \neq n_2$ and for all $(n, \omega) \in \pi$ that $\omega \in \gamma_A \circ h(n)$ holds. It is also the case that for all $(n_2, \omega) \in \pi$, $\omega \in \gamma_A \circ h(n_2)$ and $\omega \in \gamma_A \circ autrace_G(g_{n_1 \mapsto h(n_1)})$ hold. It follows that $\pi \in \gamma \circ \gamma_{\mathcal{SL}} \circ autrace^{\rightarrow}_{n_1,n_2}(h)$ holsd, which completes the proof. □

The above transformers provide the set F for ACDCL. As a downwards extrapolation, we restrict a single, heuristically-chosen control-location in N_{once} with a meet irreducible of A as shown in the example.

7.3.3 Conflict Analysis in Programs

The set of overapproximations F of the unsafe trace transformer is given by transformers of the form $autrace^{\rightarrow}_{n_1,n_2}$ and $autrace^{\leftarrow}_{n_1,n_2}$ described in the previous section. These are computed via classic forward analysis using the strongest postconditions and backward analysis using existential precondition, respectively. In order to implement effective conflict analysis, each of these operators needs to be associated with an underapproximation of the safe-trace transformer $strace_G$ over the downset abstraction (condition (iv) of) $\mathcal{D}(\mathcal{SL}_A(G))$ which can be used to generalize the analysis results.

We now work our way towards developing such a transformer. As a basic building block, we will use underapproximate weakest precondition and universal postcondition state transformers. This will allow us to break down the computation of the safe trace transformer into local steps that underapproximate weakest preconditions and universal postconditions along the CFG edges. Recall that for each edge (n_1, n_2), $st(n_1, n_2)$ denotes the statement along the edge and $T^{\Omega}_{st(n_1,n_2)}$ is the transition relation associated with that statement. We can associate with each edge (n_1, n_2) a weakest precondition $\tilde{pre}_{n_1,n_2} \mathrel{\hat{=}} \tilde{pre}_{T^{\Omega}_{st(n_1,n_2)}}$ and a universal postcondition $\tilde{post}_{n_1,n_2} \mathrel{\hat{=}} \tilde{post}_{T^{\Omega}_{st(n_1,n_2)}}$ in $\mathcal{P}(\Omega) \to \mathcal{P}(\Omega)$.

In order to do conflict analysis, we require underapproximations of the above transformers. The lattice of memory states $\mathcal{P}(\Omega)$ is underapproximated by the downset completion $\mathcal{D}(A)$ of A. We assume that for all transformers $\tilde{pre}_{n_1,n_2}$ and $\tilde{post}_{n_1,n_2}$ we have sound underapproximations $a\tilde{pre}_{n_1,n_2}$ and $a\tilde{post}_{n_1,n_2}$ with signature $\mathcal{D}(A) \to \mathcal{D}(A)$.

From these functions, we can build underapproximate transformers for the lattice $\mathcal{N}(G)$. We underapproximate $\mathcal{N}(G)$ using the lattice $\mathcal{N}_{\mathcal{D}(A)}(G)$ which is the set of mappings from control locations to downwards closed subsets of A.

Proposition 7.3.5. *The lattice $\mathcal{N}_{\mathcal{D}(A)}(G)$ underapproximates the lattice $\mathcal{N}(G)$ w.r.t. the pair $(\alpha_{\mathcal{D}}, \gamma_{\mathcal{D}})$ defined below.*

$$(\mathcal{N}(G), \sqsupseteq) \overset{\gamma_{\mathcal{D}(A)}}{\underset{\alpha_{\mathcal{D}(A)}}{\leftrightarrows}} (\mathcal{N}_{\mathcal{D}(A)}(G), \sqsupseteq)$$
$$\alpha_{\mathcal{D}(A)}(f)(n) \mathrel{\hat{=}} \{a \in A \mid \gamma_A(a) \subseteq f(n)\}$$
$$\gamma_{\mathcal{D}(A)}(g)(n) \mathrel{\hat{=}} \bigcup_{a \in g(n)} \gamma_A(a)$$

Proof. It is easy to see that $\alpha_{\mathcal{D}(A)}$ and $\gamma_{\mathcal{D}(A)}$ are monotone. Now let g be an element of $\mathcal{N}_{\mathcal{D}(A)}(G)$, n be a control location in N and $a \in A$ be an element of $g(n)$. Then $\gamma_{\mathcal{D}(A)}(g)(n) \supseteq \gamma_A(a)$ and therefore $a \in \alpha_{\mathcal{D}(A)} \circ \gamma_{\mathcal{D}(A)}(g)(n)$. This shows that $\alpha_{\mathcal{D}(A)} \circ \gamma_{\mathcal{D}(A)}$ is inflationary.

Now let f be an element of $\mathcal{N}(G)$, n be a control location in N and ω be a memory state in $\gamma_{\mathcal{D}(A)} \circ \gamma_{\mathcal{D}(A)}(f)(n)$. Then there exists an element a in $\gamma_{\mathcal{D}(A)}(f)(n)$ such that ω is in $\gamma_A(a)$ and therefore ω is also in $f(n)$ by definition of $\gamma_{\mathcal{D}(A)}(f)(n)$. It follows that $\gamma_{\mathcal{D}(A)} \circ \gamma_{\mathcal{D}(A)}$ is deflationary. The result follows. □

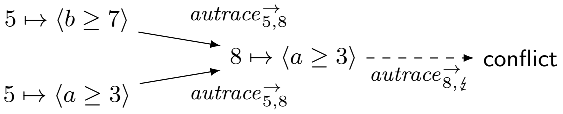
We can now use the abstract edge transformers to define corresponding transformers for the whole CFG G.

Definition 7.3.3 (Universal Control-Location Approximations)**.**

$$\tilde{apre}_G, \tilde{apost}_G : \mathcal{N}_{\mathcal{D}(A)}(G) \to \mathcal{N}_{\mathcal{D}(A)}(G)$$
$$\tilde{apre}_G(g)(n) \mathrel{\hat{=}} \bigcap_{(n,n') \in E} \tilde{apre}_{n,n'} \circ g(n') \quad \tilde{apost}_G(g)(n) \mathrel{\hat{=}} \bigcap_{(n',n) \in E} \tilde{apost}_{n',n} \circ g(n')$$

We show using an example how generalization can be achieved using a greatest fixed point computation over the above transformers.

Example 7.3.2. *We recall part of the implication graph from Example 7.3.1 below.*


$5 \mapsto \langle b \geq 7 \rangle$
$autrace^{\rightarrow}_{5,8}$
$8 \mapsto \langle a \geq 3 \rangle$
conflict
$autrace^{\rightarrow}_{8,\lightning}$
$5 \mapsto \langle a \geq 3 \rangle$
$autrace^{\rightarrow}_{5,8}$

The element at location 8 was obtained by approximating a least fixed point using an overapproximate strongest postcondition. We generalize this element by computing a greatest fixed point over an underapproximate weakest precondition in the abstract domain $\mathcal{N}_{\mathcal{D}(A)}(G)$. This corresponds to collecting and iteratively refining a set of elements that are interpreted disjunctively at each control-location. We first generalize the edge $(8, \lightning)$ which is labeled with $[a = 0]$, below we show the iteration sequence as a vector whose components are annotated with the corresponding control-location.

$$\begin{pmatrix} \vdots \\ 8 : \top \\ \lightning : \bot \end{pmatrix} \begin{pmatrix} \vdots \\ 8 : \tilde{apre}_{n_8,\lightning}(\top) = \downarrow\{\langle a < 0 \rangle, \langle a > 0 \rangle\} \\ \lightning : \bot \end{pmatrix}$$

Note that location 8 records two possible reasons. Recall that ACDCL *uses an upwards interpolation to pick a single element among a number of candidate reasons. We apply upwards interpolation to choose an element c that is, in terms of precision, between the original conflict and the result of transformer application, i.e., $\downarrow\langle a < 3 \rangle \subseteq \downarrow c \subseteq \downarrow\{\langle a < 0 \rangle, \langle a > 0 \rangle\}$,* *and we obtain $\langle a < 0\rangle$. We now generalize the elements at location 5 with respect to this new marking for 8, which requires computing a greatest fixed point over the loop. Note that in the following, we use weakest-precondition transformers that strictly underapproximate their concrete counterparts.*

$$\begin{pmatrix} \vdots \\ 5 : \top \\ 6 : \top \\ 7 : \top \\ 8 : \downarrow\langle a > 0\rangle \\ \vdots \end{pmatrix} \begin{pmatrix} \vdots \\ 5 : \top \\ 6 : \downarrow\langle a > 0\rangle \\ 7 : \top \\ 8 : \downarrow\langle a > 0\rangle \\ \vdots \end{pmatrix} \begin{pmatrix} \vdots \\ 5 : \downarrow\langle a > 0\rangle \\ 6 : \downarrow\langle a > 0\rangle \\ 7 : \downarrow\langle a > 0\rangle \\ 8 : \downarrow\langle a > 0\rangle \\ \vdots \end{pmatrix} \begin{pmatrix} \vdots \\ 5 : \downarrow\langle a > 0\rangle \\ 6 : \downarrow\langle a > 0\rangle \\ 7 : \downarrow\langle a > 0, b > 0\rangle \\ 8 : \downarrow\langle a > 0\rangle \\ \vdots \end{pmatrix}$$

$$\begin{pmatrix} \vdots \\ 5 : \downarrow\langle a > 0\rangle \\ 6 : \downarrow\langle a > 0, b > 0\rangle \\ 7 : \downarrow\langle a > 0, b > 0\rangle \\ 8 : \downarrow\langle a > 0\rangle \\ \vdots \end{pmatrix} \begin{pmatrix} \vdots \\ 5 : \downarrow\langle a > 0, b > 0\rangle \\ 6 : \downarrow\langle a > 0, b > 0\rangle \\ 7 : \downarrow\langle a > 0, b > 0\rangle \\ 8 : \downarrow\langle a > 0\rangle \\ \vdots \end{pmatrix}$$

In the original implication graph, the node 5 was labeled with $\langle a \geq 3\rangle$ and $\langle b \geq 7\rangle$. Based on the above iteration sequence, we may then generalize these markings to $\langle a \geq 0\rangle$ and $\langle b \geq 0\rangle$, respectively.

We provide a formalization that captures the example above. A slight difficulty is that $\mathcal{N}_{\mathcal{D}(A)}(G)$ underapproximates $\mathcal{N}(G)$, which in turn overapproximates $\mathcal{P}(\Pi)$. Because of the combination of over- and underapproximation, we may not use the composition of the two Galois connections to derive a sound underapproximation of the safe trace transformers. Instead, we define a transformer over $\mathcal{D}(\mathcal{SL}_A(G))$ which internally uses fixed point computation over $\mathcal{N}_{\mathcal{D}(A)}(G)$ and manually prove that this approach yields sound results.

This internal fixed point is computed by applying the following transformers.

Definition 7.3.4 (Global Safe-Trace Transformers)**.**

$$\begin{aligned} \mathit{astrace}_G^{\leftarrow}, \mathit{astrace}_G^{\rightarrow} &: \mathcal{N}_{\mathcal{D}(A)}(G) \rightarrow: \mathcal{N}_{\mathcal{D}(A)}(G) \\ \mathit{astrace}_G^{\leftarrow}(g) &\mathrel{\hat{=}} \mathsf{gfp}\, X.\ g \sqcup (g_{\Xi} \sqcap \widetilde{\mathit{apre}}_G(g)) \\ \mathit{astrace}_G^{\rightarrow}(g) &\mathrel{\hat{=}} \mathsf{gfp}\, X.\ g \sqcup (g_{\neg I} \sqcap \widetilde{\mathit{apost}}_G(g)) \end{aligned}$$

The functions g_{Ξ} and $g_{\neg I}$ are defined as follows.

$$g_{\Xi}(n) \mathrel{\hat{=}} \begin{cases} \emptyset & \text{if } n = \lightning \\ \downarrow\top & \text{otherwise} \end{cases} \qquad g_{\neg I} \mathrel{\hat{=}} \begin{cases} \emptyset & \text{if } n = n_I \\ \downarrow\top & \text{otherwise} \end{cases}$$

Note that the lattice $\mathcal{N}_{\mathcal{D}(A)}(G)$ underapproximates $\mathcal{N}_A(G)$, which in turn overapproximates $\mathcal{P}(\Pi)$. Because of this, we cannot use standard soundness arguments to build underapproximations of strace_G. We will use the following lemma to establish soundness later.

Lemma 7.3.6. *Let $n, n' \in N_{once}$ such that $n' \prec n$ and let $g_{n\mapsto R} \in \mathcal{N}_{\mathcal{D}(A)}(G)$ be the function that maps n to R and every other node to $\downarrow\top$. Then for every counterexample trace π that is not in $\gamma_{\mathcal{D}(A)}(g_{n\mapsto R})$ and for every memory state $\omega \in \Omega$, if (n', ω) is in π then ω is not in $\gamma_{\mathcal{D}(A)} \circ \mathit{astrace}_G^{\leftarrow}(g_{n\mapsto R})(n')$.*

Proof. Let π be a counterexample trace that is not in $\gamma_{\mathcal{D}(A)}(g_{n\mapsto R})$. Both locations n and n' occur exactly once on π. Let $\omega, \omega' \in \Omega$ be the memory states such that $(n, \omega), (n', \omega') \in \pi$ and let j be the index such that $\pi_j = (n, \omega)$.

We use induction to show that for all $1 \leq i \leq j$ where $\pi_i = (n_i, \omega_i)$, it holds that $\omega_i \notin \gamma_{\mathcal{D}(A)} \circ \mathit{astrace}^{\leftarrow}_{G}(g_{n\mapsto R})(n_i)$. The base case is simple, since by assumption we have that $(n_j, \omega_j) = (n, \omega)$ and that $\pi \notin \gamma_{\mathcal{D}(A)}(g_{n\mapsto R})$. Now assume that it holds that $\omega_{i+1} \notin \gamma_{\mathcal{D}(A)} \circ \mathit{astrace}^{\leftarrow}_{G}(g_{n\mapsto R})(n_{i+1})$. Then $\gamma_{\mathcal{D}(A)} \circ \mathit{astrace}^{\leftarrow}_{G}(g_{n\mapsto R})(n_i) \subseteq \tilde{pre}_{n_i,n_{i+1}}(\gamma_{\mathcal{D}(A)} \circ \mathit{astrace}^{\leftarrow}_{G}(g_{n\mapsto R})(n_{i+1}))$ holds. Now recall that π is well-formed and therefore each successive pair of states is related by the transition relation. Then, if ω_i were in $\gamma_{\mathcal{D}(A)} \circ \mathit{astrace}^{\leftarrow}_{G}(g_{n\mapsto R})(n_i)$, then also $\omega_i \in \tilde{pre}_{n_i,n_{i+1}}(\gamma_{\mathcal{D}(A)} \circ \mathit{astrace}^{\leftarrow}_{G}(g_{n\mapsto R})(n_{i+1}))$ and we could conclude that $\omega_{i+1} \in \gamma_{\mathcal{D}(A)} \circ \mathit{astrace}^{\leftarrow}_{G}(g_{n\mapsto R})(n_{i+1})$, which would violate the induction hypothesis. Therefore $\omega_i \notin \gamma_{\mathcal{D}(A)} \circ \mathit{astrace}^{\leftarrow}_{G}(g_{n\mapsto R})(n_i)$, which completes the inductive argument.

Since $n' \prec n$, we have that $(n', \omega') = \pi_i$ for some $i < j$. Therefore, $\omega' \notin \gamma_{\mathcal{D}(A)} \circ \mathit{astrace}^{\leftarrow}_{G}(g_{n\mapsto R})(n')$, which concludes the proof. □

We now define the generalization transformers required by ACDCL, over the downset domain $\mathcal{D}(\mathcal{SL}_A(G))$. These transformers first compute greatest fixed points over $\mathcal{N}_{\mathcal{D}(A)}(G)$ and then extract the abstract element at the control-location of interest.

Definition 7.3.5 (Straight-Line Safe-Trace Transformers)**.** For $n_1, n_2 \in N_{once}$ such that n_2 immediately follows n_1 in the ordering $\preceq$ and for an element $h \in \mathcal{SL}_A(G)$, we define the following transformers.

$$\begin{gathered}
\mathit{astrace}^{\leftarrow}_{n_1,n_2,h}, \mathit{astrace}^{\rightarrow}_{n_1,n_2,h} : \mathcal{D}(\mathcal{SL}_A(G)) \to \mathcal{D}(\mathcal{SL}_A(G)) \\
\mathit{astrace}^{\leftarrow}_{n_1,n_2,h}(R) \hat{=} R \cup \{h'[n_1 \mapsto a] \mid h' \in R \wedge a \in \mathit{astrace}^{\leftarrow}_{G}(g_{n_2\mapsto\{h(n_2)\}})(n_1)\} \\
\mathit{astrace}^{\rightarrow}_{n_1,n_2,h}(R) \hat{=} R \cup \{h'[n_2 \mapsto a] \mid h' \in R \wedge a \in \mathit{astrace}^{\rightarrow}_{G}(g_{n_1\mapsto\{h(n_1)\}})(n_2)\} \\
\text{where } g_{n\mapsto S}(n') \hat{=} \begin{cases} S & \text{if } n = n' \\ \top & \text{otherwise} \end{cases}
\end{gathered}$$

Recall that a requirement for AFIRSTUIP is that every overapproximate transformer needs to be associated with an underapproximate dual transformer. We associate with each transformer $\mathit{autrace}^{\rightarrow}_{n_1,n_2}$ the generalization transformer $\mathit{astrace}^{\leftarrow}_{n_1,n_2,h}$ and with each transformer $\mathit{autrace}^{\leftarrow}_{n_1,n_2}$ the generalization transformer $\mathit{astrace}^{\rightarrow}_{n_1,n_2,h}$. In other words, we generalize applications of the strongest postcondition using an underapproximate weakest precondition and applications of the existential precondition using an underapproximate universal postcondition. We now show that the generalization transformers satisfy the required property for instantiating ACDCL.

Proposition 7.3.7. *The transformers $\mathit{astrace}^{\rightarrow}_{n_1,n_2,h}$ and $\mathit{astrace}^{\leftarrow}_{n_1,n_2,h}$ underapproximate the concrete transformer $P \mapsto \mathit{strace}_G(P \cup \gamma \circ \gamma_{\mathcal{SL}}(h))$.*

Proof. We give the proof for $\mathit{astrace}^{\leftarrow}_{n_1,n_2,h}$. The other proof is similar.

For this proof, we denote by γ the concretization function of the lattice $\mathcal{D}(\mathcal{SL}_A(G))$ which can be defined as follows.

$$\begin{gathered}
\gamma : \mathcal{D}(\mathcal{SL}_A(G)) \to \mathcal{P}(\Pi) \\
\gamma(R) \hat{=} \{\pi \in \Pi \mid \forall n \in N_{once}, \omega \in \Omega.\ (n, \omega) \in \pi \implies \exists r \in R.\ \omega \in \gamma_A \circ r(n)\}
\end{gathered}$$

Let $\pi \in \gamma \circ \gamma_{\mathcal{SL}} \circ \mathit{astrace}^{\leftarrow}_{n_1,n_2,h}(P)$. For a contradiction, assume that $\pi \notin \mathit{strace}_G(P \cup \gamma \circ \gamma_{\mathcal{SL}}(h))$. Then $\pi \notin P$, $\pi \notin \gamma \circ \gamma_{\mathcal{SL}}(h)$ and π is a counterexample trace, that is, a well-formed trace that reaches the error location.

Then from the definition of $\mathit{astrace}^{\leftarrow}_{n_1,n_2,h}$ and the fact that $\pi \notin P$, we have that $\pi \in \gamma(\downarrow h'[n_1 \mapsto a])$ where $h' \in R$ and $a \in \mathit{astrace}^{\rightarrow}_{G}(g_{n_2\mapsto\{h(n_2)\}})(n_1)$. From Lemma 7.3.6 we then have that for every state $(n_1, \omega) \in \pi$ that $\omega \notin \gamma_A(a)$. But then $\pi \notin \gamma(\downarrow h'[n_1 \mapsto a])$, which contradicts the above.

It follows that $\gamma \circ \gamma_{\mathcal{SL}} \circ \mathit{astrace}^{\leftarrow}_{n_1,n_2,h}(P) \subseteq \mathit{strace}_G(P \cup \gamma \circ \gamma_{\mathcal{SL}}(h))$. □

7.4 A Learning Interval Analysis

We have implemented a prototype of ACDCL for ANSI-C programs which we call Conflict-Driven Fixed Point Learning (CDFPL) [60]. The abstract domain used is the product of IEEE754 floating-point and bounded integer intervals, which allows for the approximation of programs that mix machine integer and floating-point variables. Our prototype is a limited instantiation of the framework presented in the previous section: it uses only forwards analysis and restricts decision making and learning to the initial control-flow node. CDFPL is able to efficiently prove correctness of several programs where a standard interval analysis yields a false alarm and is considerably more efficient than bounded-model checking.

In case our procedure fails to prove correctness, it returns an interval environment that assigns variables at the initial control-flow node to constants. This assignment either leads to an error, or helps localise the imprecision of the abstract analysis by providing an initial state for which abstract analysis fails. If the program under analysis contains only arithmetic operations, does not have loops and uses deterministic rounding, then our analysis is a complete way to prove correctness. We apply our analysis to verify properties on floating-point programs from various sources and show that in many cases, our analysis is as efficient as static analysis, but provides the precision of a floating-point decision procedure.

CDFPL uses standard forward analysis during counterwitness search, that is, least fixed point computation with a strongest postcondition transformer. Decisions iteratively restrict the initial control location n_I by placing upper and lower bounds on variables, i.e., meet irreducibles of the form $\langle x \leq c \rangle$ or $\langle x \geq c \rangle$. The decision variable x and whether a lower or an upper bound is used in the decision element is decided at random. The bound constant c is chosen as the midpoint the between current lower and upper bound of the variable x. Decisions and deductions iterate until safety can be established or a counterexample is found which corresponds to a conflict.

Conflict generalization computes an underapproximation of the weakest precondition using search. During this search, variable bounds on the initial state are iteratively weakened and forward analysis is run to see if the resulting element is still sufficiently strong to show safety. A process similar to binary search is used to find maximal sufficient weakenings, which correspond to maximal underapproximations of the weakest precondition. The computation has upwards interpolation built-in: when multiple, incomparable generalizations of an element exist, one is chosen at random.

We compare our tool to the static analyzer ASTRÉE [48], which uses interval analysis, and to the bounded model checker CBMC [34], which uses a bit-precise floating-point decision procedure based on propositional encoding. Our benchmarks use non-linear floating-point computations. ASTRÉE is not optimized for the kinds of programs we consider and introduces a high degree of imprecision. CBMC translates the floating-point arithmetic to large circuits which are hard for SAT solvers. As benchmarks, we use ANSI-C code originating from (i) controller code auto-generated from a Simulink model with varying loop bounds; (ii) examples from the Functional Equivalence Verification Suite [162]; (iii) benchmarks presented at the 2010 Workshop on Numerical Software Verification; (iv) code presented by Goubault and Putot [74]; (v) hand-crafted instances that implement Taylor expansions of sine and square functions, as well as Newton-Raphson approximation. In order to allow comparison to bounded model checking, only benchmark programs with bounded loops were chosen, which were completely unrolled prior to analysis. All of our 57 benchmarks, more detailed benchmark results, together and the prototype tool, are available online[1].

We discuss the following results. (i) CDFPL is as precise as a full floating-point decision procedure while still being orders of magnitudes faster; (ii) learning and the choice of decision heuristic yield a speed-up of more than an order of magnitude; (iii) dynamic precision adjustment is observed frequently.

[1] `http://www.cprover.org/cdfpl/`

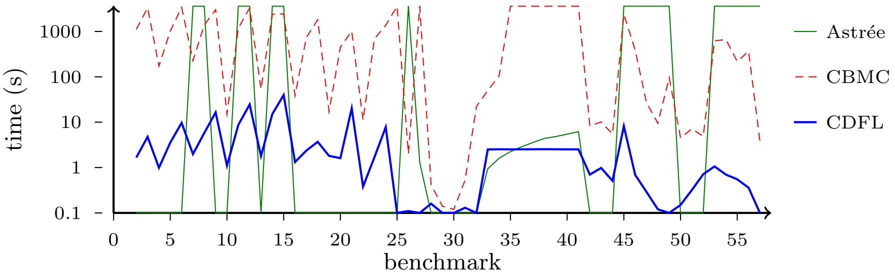


Figure 7.4: Execution times of ASTRÉE, CBMC and CDFPL; wrong results set to 3600s.

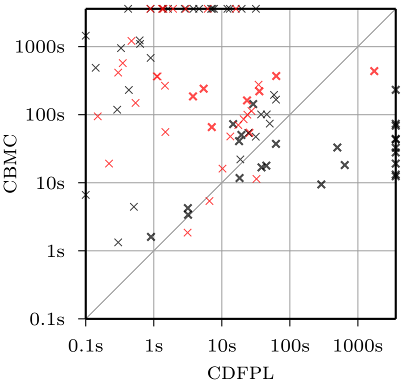


Figure 7.5: Comparison CDFPL and CBMC. Safe instances shown in red.

Efficient and Precise Analysis

In Figure 7.4, we show execution times for ASTRÉE, CBMC and CDFPL. To highlight imprecise verification results or out-of-memory errors, we set their time to the timeout of 3600 seconds. We make several observations: on average, our analysis is at least 264 times faster than CBMC. The number 264 is a lower bound, since some runs of CBMC were aborted due to timeouts or errors. The maximum speed-up is a factor of 1595. A detailed comparison with CBMC is shown in Figure 7.4. Although ASTRÉE is often faster than our prototype, its precision is insufficient in many cases – we obtained 16 false alerts for the 33 safe benchmarks.

Effects of Decision Heuristics and Learning

Figure 7.6 visualizes the effects of learning and decision heuristics. Learning has a significant influence on runtime, as does the choice of a decision heuristic. We compare a random heuristic which picks a restriction of a random variable, with a range-based one, which always aims to restrict the least restricted variable. Random decision making outperforms range-based.

Dynamic Precision Adjustment

Instantiating ACDCL as a program analysis yields a program and property-dependent analysis refinement algorithm, in which the precision of the analysis is dynamically adapted to match the precision required to prove the property. This is illustrated in Figure 7.7 where we check bounds on the result of computing a sine approximation under the input range $[-\frac{\pi}{2}, \frac{\pi}{2}]$. The curve in the figure shows an input-output mapping of a function implemented in ANSI-C. The input value is shown on the x-axis, the output value on the y-axis. If the input value is outside the allowed input range, the function immediately returns. Otherwise the function

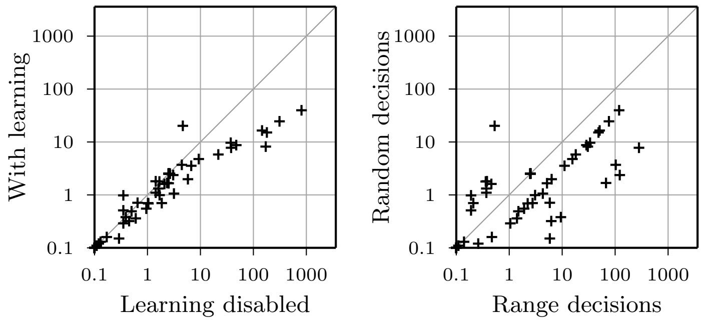


Figure 7.6: Effects of learning and decision heuristics.

#	**safety bound**	**safe**	**slack**	**sec.**	**iter.**	**back jumps**	**unit rule**	**avg. b.j. lvl.**
1	x <1.0	n	-0.00921	0.52	159	10	15	14.1
2	x <1.009	n	-0.00021	2.60	490	52	86	17.85
3	x <1.0099	y	0.00069	7.41	1091	152	832	13.27
4	x <1.01	y	0.00079	5.26	738	104	532	8.77
5	x <1.05	y	0.04079	0.96	159	19	34	2.21
6	x <1.1	y	0.09079	0.59	75	12	22	1.5

Table 7.1: Effects of bound tightness on runtime.

computes an output value and checks whether it satisfies the specification. The specification is expressed as a range of allowed values (the boundaries of the allowed range are shown as two horizontal red lines).

A run of the algorithm iteratively refines the input range of the program until a proof of correctness is achieved using standard forward analysis. It then uses conflict generalization to find a maximal safe input range, which is subsequently never explored again. A successful run will therefore produce a covering of the input space using overlapping input ranges. The partition boundaries, extracted from a run of CDFPL, are shown as black vertical lines in the figure. The actual maximum of the function lies at about 1.00921. Figure 7.7 shows how the number of partitions dynamically increases as the specification approaches this value. The precision of the analysis is always higher at the edges, since the function output is closer to the boundaries of the allowed range.

In the figure, the number of partitions increases monotonically with the tightness of the specification. The last graph of the figure is an exception: There, a counterexample is found at the point where the black lines converge, which causes search to be abandoned early.

The effect of bound tightness on runtime behavior is also demonstrated in Table 7.1. *Slack* denotes the difference between the upper bound of the allowed range and the maximal output value of the program. Less slack leads to a higher number of iterations and conflicts, a deeper average backjump level and a higher number of unit transformer applications.

Limitations of CDFL

CDFPL instantiates the ACDCL framework using interval analysis as an abstract domain. There are simple programs that are not amenable to interval analysis, even when case splits and partition based refinement are used. Consider, for example, the one-line program $x := y$ together with the specification $x = y$. Intervals are non-relational, hence CDFPL would enumerate all singleton intervals over y. Similar enumeration behavior can be found in propositional SAT solvers, which may perform badly when applied to certain, highly

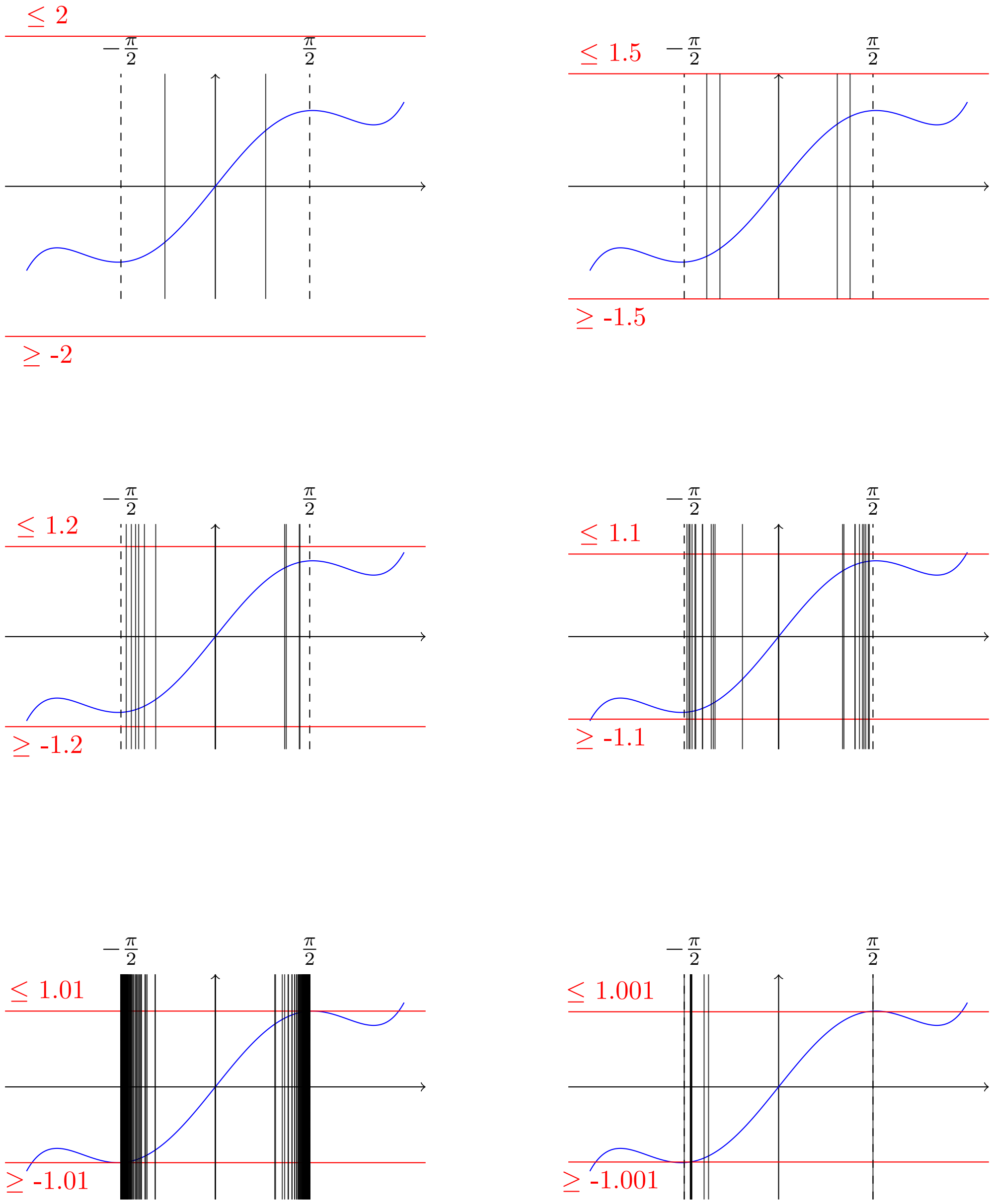


Figure 7.7: Partitions explored during analysis. The bound to be checked is shown in red.

relational problems. This can be fixed by instantiating ACDCL over a richer domain. The implementation is also a prototype and restricts learning to the initial control-flow node, which limits performance on deep programs.

7.5 Related Work

In this chapter, we have shown how ACDCL can be instantiated to obtain automatic and problem-directed partition-based refinement of an abstract program analysis. The practical results we present in Section 7.4 for floating-point programs using our prototype tool CDFPL [60] were state-of-the-art at the time of publication, but were recently improved upon by a static analysis method that combines abstract interpretation with advanced deduction techniques from constraint programming [148, 147]. These advances are orthogonal to the advantages introduced in ACDCL. We expect that instantiating ACDCL using the advanced deduction techniques in [148, 147] will bring further improvements.

CDFPL can be understood as refining an abstract analysis using a dynamically created set of partitions. We now review other approaches that aim to increase precision of an abstract analysis using partitions.

Partitioning in Abstract Analyses ACDCL for program correctness implements a form of dynamic partition-based analysis refinement. Abstractly, partition-based refinement corresponds to construction of the reduced cardinal power lattice $A \rightarrow B$ of two abstract domains A and B, which is one of the basic forms of domain combinations presented in [43]. Partitioning of control-systems for additional precision was also introduced early [39].

Partition-based refinements for program analyses have been explored extensively in the literature. An exhaustive overview is provided in [151]. In addition to approaches which are explicitly formalized as applications of partitioning, such as [21, 98, 99], static analyses that increase precision by selectively distinguishing between control-flow paths or by rewriting the underlying CFG, such as [126], can be considered instances of partitioning.

Trace partitioning refers to approaches where partitions consist of sets of program traces. A formal framework for trace partitioning was introduced in [90] and later generalized to a wider range of partitions and to include dynamic partitions in [119, 151]. In the case of dynamic partitions, a strategy for selecting and changing partitions is required. The authors of [119, 151] suggest the use of manual program annotations combined with simple automatic heuristics. ACDCL provides an automatic framework for creating partitions in a manner that is program and property directed: additional precision is introduced only where needed.

Lifting Satisfiability Architectures to Program Analysis Most closely related to our work is the *Satisfiability Modulo Path Programs* (SMPP) [91] algorithm, which was one of the initial inspirations for the work in this dissertation. SMPP presents an informal lifting of the DPLL(T) framework to program analysis. In DPLL(T), a propositional SAT solver enumerates candidate solutions which are then checked using a theory solver. In SMPP, a propositional SAT solver is used to enumerate paths through a CFG that may approximate a counterexample. These candidate CFG paths are then checked for correctness using an abstract interpreter. If the proof succeeds, a sophisticated conflict analysis phase generalizes the results of the abstract analysis using a search procedure. The goal is to identify a generalized set of paths that contain no error. This set is encoded propositionally and added to the SAT formula in a learning step. In terms of abstract satisfaction, SMPP can be viewed as an instance of ACDCL that operates on a product abstraction similar to that used by DPLL(T) (see Section 4.5).

The technique of [123] (extended in [2, 3]) presents a program analysis that is inspired by DPLL which explores a CFG in way that is similar to how a CDCL solver explores the search space of propositional solutions. The state of the analysis is represented by an unwinding of the CFG, which is viewed as analogous to the search tree of a SAT solver. The tool extends the unwinding by advancing along a single path through the CFG. If an error location is reached, an SMT solver is used to check the feasibility of the path, i.e., to check whether the path approximates some counterexample trace. If this is not the case, the procedure extracts logical interpolants from the path formula, which constitute underapproximations of the weakest precondition and are then used to label the nodes of the unwinding. This process is viewed as analogous to learning. After learning, the procedure backtracks and explores a different CFG path. The labellings obtained by interpolation provide generalized reasons for correctness and prevent redundant exploration of control-flow edges. The relation between CDCL and the analysis is established informally by analogy. Abstract satisfaction and abstract conflict-driven learning make such informal analogies precise.

Using Decision Procedures in Abstract Analysis The work in [114] uses an SMT solver to perform a top-down refinement of loop invariants. Starting from an overapproximate invariant, a constraint solver identifies sets of problematic behaviors that are approximated by the candidate invariant and lead to an error. Each of these sets of behaviors becomes a part of an iteratively constructed partition and is analyzed individually using an abstract interpreter, leading to a disjunctive invariant.

In [135], an SMT solver is used to find well-formed traces of a program that are not yet covered by a candidate invariant; an abstract representation of these traces is then used to compute the next candidate. The process terminates when all well-formed traces are covered, at which point the resulting abstract element is a sound approximation of program semantics.

The YOGI tool [12, 144] uses a decision procedure to construct an underapproximation of program semantics that consists of a set of program traces. This underapproximation is combined with a partition-based overapproximation of the set of program states. The overapproximation guides the underapproximation by determining how the underapproximate set of traces is extended. Conversely, when a trace in the underapproximation cannot be extended further, the set of partitions is refined.

ACDCL and Counterexample-Driven Abstraction Refinement A natural question is whether ACDCL is a form of Counter-Example Guided Abstraction Refinement (CEGAR) [35]. CEGAR computes an abstract semantics over a program and triggers refinement when a spurious counterexample is found, that is, a witness of imprecise overapproximation in the abstract domain. In every refinement step, CEGAR computes a new abstract domain consisting of a lattice and new transformers. Computation of new transformers is an expensive operation [7].

In contrast, ACDCL iteratively constrains an overapproximate fixed point computation until a "spurious" conflict is derived, that is, a contradiction resulting from excessive underapproximation. The resulting partial proof is used to implement proof-guided transformer refinement. ACDCL never changes the domain. This immutability is crucial for efficiency, because the implementations of the abstract domain and transformers can be highly optimized.

A variant of CEGAR has been proposed for abstract program analysis [79, 78, 80]. Refinement occurs along two axes. First, widening operators are dynamically refined if they cause too much imprecision during fixed point iteration. Second, the set of control-locations is refined when the current set is found to be too imprecise, which amounts to the construction of a dynamic trace partitioning. The technique is orthogonal to ACDCL. Refinement in [79, 78, 80] changes the abstract lattice and dynamically reduces the overapproximation caused by a widening. On the other hand, refinement in ACDCL operates in a fixed lattice and refines a transformer by making explicit information that is already contained within it.

Chapter 8

Future Work and Conclusion

In this dissertation, we have shown the following thesis:

> Satisfiability decision procedures are abstract interpreters. This insight allows technology transfer between program analysis and satisfiability research and can be used to define algebraic generalizations of satisfiability architectures. The resulting generalizations have practical applications.

We have made a number of conceptual, mathematical and practical contributions to the state-of-the-art in satisfiability checking and program analysis.

The conceptual contributions include the framework of abstract satisfaction and the bottom-everywhere problem, which provide a bridge between research in abstract interpretation and satisfiability checking and lead to the development of a novel methodology for the analysis and systematic construction of satisfiability solvers and program verifiers. Our analysis of satisfiability procedures using abstract satisfaction shows the first claim, namely that decision procedures are abstract interpreters.

The mathematical contributions of this dissertation include the construction of novel decision procedure frameworks such as ACDL and ACDCL together with general soundness and completeness arguments. These contributions show the second claim, since ACDCL is an algebraic generalization that is the result of transferring solver technology from satisfiability solving to static analysis.

To show the third claim of the thesis, we presented two practical contributions which instantiate the ACDCL framework: the state-of-the-art satisfiability decision procedure FP-ACDCL and the program verifier CDFPL. Both procedures outperform competing approaches. This shows the last part ot the thesis, namely that the techniques developed in this dissertation have practical applications.

We now briefly discuss a number of interesting questions that are not addressed in this dissertation and avenues for future work.

Technology Transfer Since satisfiability solvers can be characterized as abstract interpreters, sufficiently general advances in one area can be translated to the other. An example of this is our instantiation of a program analysis that incorporates learning in Chapter 7. It stands to reason that there may be many more opportunities for technology transfer. This includes extraction of analysis procedures from other decision procedure architectures and the use of abstract-interpretation techniques in satisfiability solvers. We discuss two concrete avenues of research below.

The first avenue concerns alternative ways to implement deduction in solvers. Deduction in decision procedures corresponds to fixed point computation via Kleene iteration. Abstract interpretation research has studied alternatives to Kleene iteration such as strategy iteration [66]. Strategy iteration techniques compute precise fixed points over infinite domains without the need for acceleration operators such as widenings. It may be possible to adapt these techniques for use in satisfiability solvers, which may be useful in logics that admit infinite deductive sequences.

The second avenue of research is to further the application of reduced product constructions in satisfiability procedures. A central aspect of the DPLL(T) framework is theory combination, which is an instance of the reduced product [49]. In addition to combining solvers for different logical theories, it is sometimes useful to combine solvers that implement different techniques that offer different strengths and weaknesses to solve the same theory. Reduced products may prove a useful framework for such combinations. The DPLL(T) framework is one instance of this idea, since both the propositional as well as the theory solver reason about the same underlying first-order theory.

Exploring a wider range of product constructions may have a beneficial effect on solver performance, especially when considering special classes of problem instances that may be more amenable to some analyses than to others. As an example, consider the FP-ACDCL solver presented in Chapter 6, which operates over the abstract domain of floating point intervals. Floating point intervals are effective on many numeric problems, but ineffective when the problem instance involves reasoning about bit-vector encodings of floating-point numbers. On the other hand, propositional SAT operates over a data structure that expresses facts about the encoding. Reduced products could be used to instantiate a procedure that can simultaneously track numeric properties of a floating-point number and bit-precise information about its encoding.

Instantiations and Extensions of ACDCL In this dissertation, we have presented two instantiations of ACDCL: a decision procedure for floating point arithmetic that uses interval constraint propagation, and a program analysis that incorporates learning which operates over the abstract domain of floating-point and machine-integer environments. The interval domain is one of the earliest and simplest numeric domains and cannot express relations between multiple variables. The abstract interpretation community has developed sophisticated numeric relational abstract domains, including octagons [130], polyhedra [29], ellipsoids [152] and zonotopes [67]. Instantiating ACDCL with these domains may bring additional performance benefits. In cases where an abstract domain does not satisfy the required complementation properties, product constructions similar to DPLL(T) could be useful where decision making and learning happen in one domain, while additional deduction is performed in another.

Depending on the underlying lattice, conflict analysis in ACDCL may offer more options for generalizing reasons than conflict analysis in CDCL. In propositional satisfiability, the discovery of activity-based decision heuristics provided significant performance benefits to SAT solvers. An open question is whether similar gains can be made by developing activity-based interpolation heuristics.

In Chapter 7, we presented an ACDCL-based program analyzer. ACDCL is not limited to numeric domains, but can also be instantiated over trace-based domains. This includes, for example, domains that track control-flow history such as the recently taken branches, or information about whether a loop was traversed an even or an odd number of times. Trace partitioning is an ideal framework for developing such abstract domains. An instantiation of ACDCL over such a domain may explore the search space by focusing on paths through control-flow unwindings similar to CDCL-like program analysis algorithms such as [123].

Learning Beyond Clause Learning In Chapter 5, we showed that conflict-driven learning is an overapproximation of concrete complementation. We discussed two examples of learning operators. The first was the tabu learning operator, which returns $\bot$ when the search enters a region that was previously identified to be conflicting. The second learning operator generalized the unit rules and drives "nearly conflicting" elements away from previously identified conflicts. While tabu learning can be instantiated in all lattices, learning with generalized unit rules works only over lattices with complementable meet irreducibles. An open problem is the identification of other learning techniques and classes of abstract lattices that support these techniques.

ACDCL and CEGAR ACDCL and CEGAR are orthogonal approaches. Informally, ACDCL can be viewed to refine transforms in response to conflicts whereas CEGAR refines abstractions in response to abstract counterexamples. If ACDCL fails, it returns a minimal abstract element that is not precise enough to cause a conflict or to be an abstract counterwitness. Such minimal, imprecise elements bear some similarity to the abstract counterexamples in [121] and could be used by an appropriate algorithm to refine the underlying abstraction. This suggests a framework that combines ACDCL and CEGAR, which can perform both transformer and abstraction refinement, triggered respectively, by conflicts and spurious elements.

We believe that the work described in this dissertation significantly extends the state-of-the-art in both satisfiability research and abstract interpretation. Much of the work in this dissertation has been conceptual in nature. Our work stands in the context of broader efforts within the research community to bring together program verification and satisfiability research both conceptually and in practice. A number of recent developments attempt to unify the two fields by placing program verification firmly on logical foundations [77, 18, 112, 4]. Despite the obvious utility of these approaches, we feel that purely logic-based accounts of program verification and satisfiability provide an incomplete picture, since they relegate those aspects which cannot be modeled well logically to the secondary role of implementation concerns. This makes it hard to systematically address questions of data structures, representation and high-level algorithmic patterns.

We believe that the algebraic perspective brought to bear by abstract interpretation complements the logical perspective and addresses its shortcomings. As a methodology, it is useful to organize and understand existing work and to aid in the design of new procedures and practical verification frameworks. Abstract interpretation has been an immensely valuable tool in the area of program analysis. We have extended these benefits to logical satisfiability and to the intersection of satisfiability and verification research.

The work we presented in this dissertation eliminates the conceptual boundary between program analysis and satisfiability solving in theory and practice. Theoretically, we showed that satisfiability procedures are instances of analysis refinement procedures over abstract lattices and we provided algebraic recipes that capture both decision procedures and analysis refinement algorithms. In practice, we have shown how to build satisfiability solvers that are based on abstract domains and how to build program analyzers that use techniques from satisfiability checking. The work in this dissertation is a significant step towards enabling technology transfer between abstract interpretation and satisfiability research and towards enabling practical combinations of procedures that go beyond black-box integration.

Bibliography

[1] B. Akbarpour, A. Abdel-Hamid, S. Tahar, and J. Harrison. Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. *Computation Journal*, 53(4):465–488, 2010.

[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. From under-approximations to over-approximations and back. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 157–172. Springer, 2012.

[3] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction-and interpolation-based software verification. In *Proceedings of Computer Aided Verification*, pages 672–678. Springer, 2012.

[4] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Reachability modulo theory library. In *SMT Workshop 2012*, page 66.

[5] A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In *Proceedings of International Joint Conference on Automated Reasoning*, pages 127–141. Springer, 2010.

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

[7] T. Ball, A. Podelski, and S. K. Rajamani. *Boolean and Cartesian Abstraction for Model Checking C Programs.* Springer, 2001.

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

[9] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

[10] C. Barrett, A. Stump, and C. Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[11] R. J Bayardo Jr and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of Innovative Applications of Artificial Intelligence*, pages 203–208, 1997.

[12] N. E Beckman, A. V. Nori, S. K. Rajamani, R. J Simmons, S. Tetali, and A. V Thakur. Proofs from tests. *IEEE Transactions on Software Engineering*, 36(4):495–508, 2010.

[13] F. Benhamou. Interval constraint logic programming. In *Constraint programming: basics and trends*, pages 1–21. Springer, 1995.

[14] F. Benhamou. Heterogenous constraint solving. In *Proceedings of Algebraic and Logic Programming*, pages 62–76. Springer, 1996.

[15] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Proceedings of Computer Aided Verification*, pages 184–190. Springer, 2011.

[16] A. Biere. The evolution from Limmat to Nanosat. Technical Report ETH-TR-444, Dept. of Computer Science, ETH, 2004.

[17] G. Birkhoff. *Lattice Theory*, volume 25. AMS Bookstore, 1967.

[18] N. Bjørner, K. L. McMillan, and . Rybalchenko. On solving universally quantified horn clauses. In *Proceedings of Static Analysis Symposium*, pages 105–125. Springer, 2013.

[19] S. Boldo and J. Filliâtre. Formal verification of floating-point programs. In *Proceedings of Computer Arithmetic*, pages 187–194. IEEE Computer Society Press, 2007.

[20] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

[21] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992.

[22] M. Brain. *Towards a Language for Declarative NP Problem-Solving Systems.* PhD thesis, University of Bath, December 2010.

[23] M. Brain. An algebra of search spaces. In *Proceedings of the Workshop on Constraint Solving and Constraint Logic Programming*, pages 72–86. Springer, 2011.

[24] M. Brain. Using algebra to understand search spaces. In *Proceedings of the Automated Reasoning Workshop*, pages 45–46. School of Computer Science, The University of Manchester, 2012.

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

[26] M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening. Interpolation-based verification of floating-point programs with Abstract CDCL. In *Proceedings of Static Analysis Symposium*, pages 412–432. Springer, 2013.

[27] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *Proceedings of Formal Methods in Computer-Aided Design*, pages 69–76. IEEE Computer Society Press, 2009.

[28] A. Chapoutot. Interval slopes as a numerical abstract domain for floating-point variables. In *Proceedings of Static Analysis Symposium*, pages 184–200. Springer, 2010.

[29] L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *Proceedings of Asian Symposium on Programming Languages*, pages 3–18. Springer, 2008.

[30] L. Chen, A. Miné, J. Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *Proceedings of Static Analysis Symposium*, pages 309–325. Springer, 2009.

[31] L. Chen, A. Miné, J. Wang, and P. Cousot. An abstract domain to discover interval linear equalities. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, pages 112–128. Springer, 2010.

[32] A. Cimatti and A. Griggio. Software model checking via IC3. In *Proceedings of Computer Aided Verification*, pages 277–293. Springer, 2012.

[33] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[34] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[35] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[36] 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.

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

[38] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'état ès sciences mathématiques, Grenoble University, 1981.

[39] P. Cousot. Semantic foundations of program analysis. In *Program Flow Analysis*, pages 303–342. Prentice Hall, 1981.

[40] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics*, pages 138–156. Springer, 2001.

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

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

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

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

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

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

[47] P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.

[48] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *Proceedings of European Symposium on Programming*, pages 21–30. Springer, 2005.

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

[50] M. Cowlishaw, editor. *IEEE Standard for Floating-Point Arithmetic*. IEEE Computer Society Press, 2008.

[51] M. Daumas, L. Rideau, and L. Théry. A generic library for floating-point numbers and its application to exact computing. In *Proceedings of Theorem Proving in Higher Order Logics*, pages 169–184. Springer, 2001.

[52] B. A. Davey and H. A. Priestley. *Introduction to lattices and order.* Cambridge University Press, 2002.

[53] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[54] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[55] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[56] L. de Moura and D. Jovanović. A model-constructing satisfiability calculus. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, pages 1–12. Springer, 2013.

[57] 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 *Proceedings of Formal Methods for Industrial Critical Systems*, pages 53–69. Springer, 2009.

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

[59] V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. In *Proceedings of Principles of Programming Languages*, pages 143–154. ACM Press, 2013.

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

[61] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[62] B. Dutertre and L. De Moura. The YICES SMT solver. *Tool paper*, 2:2, 2006.

[63] J. Feret. Static analysis of digital filters. In *Proceedings of European Symposium on Programming*, pages 33–48. Springer, 2004.

[64] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal of Satisfiability*, 1(3-4):209–236, 2007.

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

[66] T. Gawlitza and H. Seidl. Precise fixpoint computation through strategy iteration. In *Proceedings of Programming Languages and Systems*, pages 300–315. Springer, 2007.

[67] K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain Taylor1+. In *Proceedings of Computer Aided Verification*, pages 627–633. Springer, 2009.

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

[69] R. Giacobazzi and F. Ranzato. Example-guided abstraction simplification. In *Automata, Languages and Programming*, pages 211–222. Springer, 2010.

[70] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[71] D. Goldwasser, O. Strichman, and S. Fine. A theory-based decision heuristic for DPLL(T). In *Proceedings of Formal Methods in Computer-Aided Design*, pages 1–8. IEEE Computer Society Press, 2008.

[72] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of Innovative Applications of Artificial Intelligence*, pages 431–437, 1998.

[73] E. Goubault. Static analyses of the precision of floating-point operations. In *Proceedings of Static Analysis Symposium*, pages 234–259. Springer, 2001.

[74] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proceedings of Static Analysis Symposium*, pages 18–34. Springer, 2006.

[75] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *Proceedings of Formal Methods for Industrial Critical Systems*, pages 3–20. Springer, 2007.

[76] P. Granger. Improving the results of static analyses of programs by local decreasing iteration. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, pages 68–79. Springer, 1992.

[77] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A software verifier based on horn clauses - (competition contribution). In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 549–551. Springer, 2012.

[78] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–458. Springer, 2008.

[79] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Refining abstract interpretations. *Information Processing Letters*, 110(16):666–671, 2010.

[80] B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 474–=488. Springer, 2006.

[81] S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *Proceedings of Principles of Programming Languages*, pages 239–251. ACM Press, 2009.

[82] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of Principles of Programming Languages*, pages 235–246. ACM Press, 2008.

[83] S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *Proceedings of Static Analysis Symposium*, pages 212–227. Springer, 2004.

[84] S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *The 15th European Symposium on Programming*, pages 279–293. Springer, March 2006.

[85] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *Proceedings of Programming Language Design and Implementation*, pages 376–386. ACM Press, 2006.

[86] S. Gulwani and A. Tiwari. Assertion checking unified. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*. Springer, 2007.

[87] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *Proceedings of European Symposium on Programming*. Springer, 2007.

[88] S. Gulwani, A. Tiwari, and G. C. Necula. Join algorithms for the theory of uninterpreted functions. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, pages 311–323. Springer, 2004.

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

[90] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Proceedings of Static Analysis Symposium*, pages 200–214. Springer, 1998.

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

[92] J. Harrison. A machine-checked theory of floating point arithmetic. In *Proceedings of Theorem Proving in Higher Order Logics*, pages 113–130. Springer, 1999.

[93] J. Harrison. Floating point verification in HOL light: The exponential function. *Formal Methods in Systems Design*, 16(3):271–305, 2000.

[94] J. Harrison. Formal verification of floating point trigonometric functions. In *Proceedings of Formal Methods in Computer-Aided Design*, pages 217–233. IEEE Computer Society Press, 2000.

[95] J. Harrison. Formal verification of square root algorithms. *Formal Methods in Systems Design*, 22(2):143–153, 2003.

[96] J. Harrison. Floating-point verification. *Journal of Universal Computer Science*, 13(5):629–638, 2007.

[97] H. Jain, F. Ivančić, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Proceedings of Computer Aided Verification*, pages 137–151. Springer, 2006.

[98] B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in Systems Design*, 23(1):5–37, 2000.

[99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Proceedings of Static Analysis Symposium*, pages 39–50. Springer, 1999.

[100] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of Computer Aided Verification*, pages 661–667. Springer, 2009.

[101] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21, 2009.

[102] D. Jovanovic. *SMT Beyond DPLL(T): A New Approach to Theory Solvers and Theory Combination*. PhD thesis, New York University, 2012.

[103] D. Jovanović, C. Barrett, and L. M. de Moura. The design and implementation of the model constructing satisfiability calculus. In *Proceedings of Formal Methods in Computer-Aided Design*. IEEE Computer Society Press, 2013.

[104] D. Jovanović and L. M. de Moura. Cutting to the chase solving linear integer arithmetic. In *Proceedings of the Conference on Automated Deduction*, pages 338–353. Springer, 2011.

[105] R. Kaivola and M. Aagaard. Divider circuit verification with model checking and theorem proving. In *Proceedings of Theorem Proving in Higher Order Logics*, pages 338–355. Springer, 2000.

[106] H. Katebi, K. A. Sakallah, and J. Marques-Silva. Empirical study of the anatomy of modern SAT solvers. In *Proceedings of Theory and Applications of Satisfiability Testing*, pages 343–356. Springer, 2011.

[107] G. A. Kildal. A unified approach to global program optimization. In *Proceedings of Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[108] A. King and H. Søndergaard. Automatic abstraction for congruences. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, pages 197–213. Springer, 2010.

[109] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[110] B. Knaster and A. Tarski. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématiques*, 5, 1928.

[111] R. Kowalski. Predicate logic as programming language. In *Proceedings of the IFIP congress*, volume 74, pages 569–544, 1974.

[112] A. Lal and S. Qadeer. Reachability modulo theories. In *Proceedings of Reachability Problems (Workshop)*, pages 23–44. Springer, 2013.

[113] F. Lapschies, J. Peleska, E. Gorbachuk, and T. Mangels. Sonolar SMT-solver. SMT-COMP 2012 System Description, 2012.

[114] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *Proceedings of Asian Symposium on Programming Languages*, pages 119–134. Springer, 2005.

[115] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[116] R. Manevich, J. Field, T. A Henzinger, G. Ramalingam, and M. Sagiv. Abstract counterexample-based refinement for powerset domains. In *Program analysis and compilation, theory and practice*, pages 273–292. Springer, 2007.

[117] J. Marques-silva. Practical applications of boolean satisfiability. In *Workshop on Discrete Event Systems*. IEEE Computer Society Press, 2008.

[118] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[119] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of European Symposium on Programming*, pages 5–20. Springer, 2005.

[120] S. A McIlraith. Logic-based abductive inference. Technical Report KSL-98-19, Knowledge Systems Laboratory, 1998.

[121] K. McMillan and L. D. Zuck. Abstract counterexamples for non-disjunctive abstractions. In *Proceedings of Workshop on Reachability Problems*, pages 176–188. Springer, 2009.

[122] K. L. McMillan. Lazy abstraction with interpolants. In *Proceedings of Computer Aided Verification*, pages 123–136. Springer, 2006.

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

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

[125] G. Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012.

[126] D. Melski and T. Reps. The interprocedural express-lane transformation. In *Proceedings of Compiler Construction*, pages 200–216. Springer, 2003.

[127] C. Michel. Exact projection functions for floating point number constraints. In *Proceedings of Artificial Intelligence and Mathematics*, 2002.

[128] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *Proceedings of Constraint Programming*, pages 524–538. Springer, 2001.

[129] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proceedings of European Symposium on Programming*, pages 3–17. Springer, 2004.

[130] A. Miné. The octagon abstract domain. *Higher Order and Symbolic Computation*, 19(1):31–100, 2006.

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

[132] D. Monniaux. Compositional analysis of floating-point linear numerical filters. In *Proceedings of Computer Aided Verification*, pages 199–212. Springer, 2005.

[133] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3), 2008.

[134] D. Monniaux. Automatic modular abstractions for linear constraints. In *Proceedings of Principles of Programming Languages*, pages 140–151. ACM Press, 2009.

[135] D. Monniaux and L. Gonnord. Using bounded model checking to focus fixpoint iterations. In *Proceedings of Static Analysis Symposium*, pages 369–385. Springer, 2011.

[136] 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. *IEEE Transactions on Computers*, 47, 1996.

[137] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of Design Automation Conference*, pages 530–535. ACM Press, 2001.

[138] J. Muller, N. Brisebarre, F. de Dinechin, C. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 1 edition, 2010.

[139] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[140] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[141] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *Proceedings of Logic for Programming, Artificial Intelligence and Reasoning*, pages 36–50. Springer, 2005.

[142] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM*, 53(6):937–977, 2006.

[143] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(t). *Journal of the ACM*, 53(6):937–977, 2006.

[144] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The yogi project: Software property checking via static analysis and testing. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–181. Springer, 2009.

[145] M. Pelleau, A. Miné, C. Truchet, and F. Benhamou. A constraint solver based on abstract domains. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, pages 434–454. Springer, 2013.

[146] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):292–304, 1986.

[147] O. Ponsini, C. Michel, and M. Rueher. Combining constraint programming and abstract interpretation for value analysis of floating-point programs. In *Proceedings of International Conference on Software Testing, Verification and Validation*, pages 775–776, 2012.

[148] O. Ponsini, C. Michel, and M. Rueher. Refining abstract interpretation based value analysis with constraint programming techniques. In *Proceedings of Constraint Programming*, pages 593–607. Springer, 2012.

[149] J. Régin. Global constraints and filtering algorithms. In *Constraint and Integer Programming*, pages 89–135. Springer, 2004.

[150] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, pages 252–266. Springer, 2004.

[151] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.

[152] P. Roux, R. Jobredeaux, P. Garoche, and É. Féron. A generic ellipsoid abstract domain for linear time invariant systems. In *Proceedings of Hybrid Systems: Computation and Control*, pages 105–114. ACM Press, 2012.

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

[154] 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. *AMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[155] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3), 2002.

[156] D. A. Schmidt. Closed and logical relations for over- and under-approximation of powersets. In *Proceedings of Static Analysis Symposium*, pages 22–37. Springer, 2004.

[157] D. A. Schmidt. Comparing completeness properties of static analyses and their logics. In *Proceedings of Asian Symposium on Programming Languages*, pages 183–199. Springer, 2006.

[158] D. A. Schmidt. A calculus of logical relations for over- and underapproximating static analyses. *Science of Computer Programming*, 64(1):29–53, 2007.

[159] D. A. Schmidt. Extracting program logics from abstract interpretations defined by logical relations. *Electronic Notes in Theoretical Computer Science*, 173:339–356, 2007.

[160] D. A. Schmidt. Internal and external logics of abstract interpretations. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, pages 263–278. Springer, 2008.

[161] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In *Formal Methods in Systems Design*, pages 82–99. Springer, 1998.

[162] S. F. Siegel and T. K. Zirkel. A functional equivalence verification suite for high-performance scientific computing. Technical Report UDEL-CIS-2011/02, Department of Computer and Information Sciences, University of Delaware, 2011.

[163] A. Tarski. Remarques sur les notions fondamentales de la méthodologie des mathématiques. *Annales de la Société Polonaise de Mathématiques*, 6:270–271, 1928.

[164] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–209, 1955.

[165] A. Thakur and T. Reps. A generalization of Stålmarck's method. Technical report 1699r, University of Wisconsin, 2012.

[166] A. Thakur and T. Reps. A method for symbolic computation of abstract operations. In *CAV*, pages 174–192. Springer, 2012.

[167] A. V. Thakur, M. Elder, and T. W. Reps. Bilateral algorithms for symbolic abstraction. In *Proceedings of Static Analysis Symposium*, pages 111–128. Springer, 2012.

[168] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, 1983.

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

[170] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[171] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 145–156. ACM, 2006.

[172] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design*, pages 279–285, 2001.