AN ABSTRACT OF THE DISSERTATION OF

Jeffrey M. Young for the degree of Doctor of Philosophy in Computer Science
presented on May 28, 2021.

Title: Variational Satisfiability Solving

Abstract approved: ____

Eric Walkingshaw

Over the last two decades, satisfiability and satisfiability-modulo theory (SAT/SMT) solvers have grown powerful enough to be general purpose reasoning engines throughout software engineering and computer science. However, most practical use cases of SAT/SMT solvers require not just solving a single SAT/SMT problem, but solving sets of related SAT/SMT problems. This discrepancy was directly addressed by the SAT/SMT community with the invention of incremental SAT/SMT solving. However, incremental SAT/SMT solvers require end-users to hand write a program which dictates the terms that are shared between problems and terms which are unique. By placing the onus on end-users, incremental solvers couple the end-users' solution to the end-users' *exact* sequence of SAT/SMT problems—making the solution overly specific—and require the end-user to write extra infrastructure to coordinate or handle the results.

This dissertation argues that the aforementioned problems result from accidental complexity produced by solving a problem that is *variational* without the concept of *variation*, similar to problematic use of GOTO statements in the absence of WHILE loop constructs. To demonstrate the argument, this thesis applies theory from *variational* programming to the domain of SAT/SMT solvers to create the first variational SAT solver and solve aforementioned problems. To do so, the thesis formalizes a variational propositional logic and specifies variational SAT solving as a compiler, which compiles variational SAT problems to non-variational SAT that are processed by an industrial strength SAT solver. It shows that the compiler is an instance of a variational fold and uses that fact to extend the variational SAT solver to an asynchronous variational SMT solver.

Variational Satisfiability Solving

by

Jeffrey M. Young

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented May 28, 2021
Commencement June 2021

Doctor of Philosophy dissertation of Jeffrey M. Young presented on May 28, 2021.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Jeffrey M. Young, Author

ACKNOWLEDGEMENTS

do it

I would like to acknowledge the Starting State and the Transition Function.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

Figure Page

LIST OF TABLES

Table Page

Todo list

Chapter 1: Introduction

Controlling complexity is a central goal of any programming language, especially as software written in that language grows. The burgeoning field of *variation theory* and *variational programming* [51, 50, 65, 33, 132] attempt to control a kind of complexity which is induced into software when many *similar yet distinct* kinds of the same software must coexist. For example, software is often *ported* to other platforms, creating similar, yet distinct instances of that software which must be maintained. Such instances of variation are ubiquitous: Web applications are tested on multiple servers; programming languages maintain backwards compatibility and so do software libraries; databases evolve over time, locale and data; and device drivers must work with varying processors and architectures. Variation theory and variational programming have been successful in small systems [49, 119, 97], yet it has not been tested in a performance demanding practical domain. In the words of Joe Armstrong[8], "No theory is complete without proof that the ideas work in practice"; this is the project of this thesis, to put the ideas of *variation* and *variational programming* to the test in the practical domain of satisfiability solving (SAT).

this is the place for variational system definition

The major contribution of this thesis is the formalization of a *VPL*, *variational satisfiability solving*, and the construction of a *variational SAT solver*. In the next section I motivate the use of variation theory and variational techniques in satisfiability solving. In addition to work on variational SAT several other contributions are made. The thesis extends variational satisfiability solving to variational SMT. It demonstrates reusable techniques and architecture for constructing *variational or variation-aware* systems using the non-variational counterparts of these systems for other domains. It shows that, with the concept of variation, the variational SMT and SAT solvers can be trivially parallelized. Lastly, the thesis provides a general algorithm to construct variational strings from a set of non-variational strings and argues for the proliferation of variation theory to other domains in computer science.

1.1 Motivation and Impact

Classic SAT, which solves the boolean satisfiability problem [21] has been one of the largest success stories in computer science over the last two decades. Although SAT solving is known to be NP-complete [38], SAT solvers based on conflict-driven clause learning (CDCL) [89, 117, 17] have been able to solve boolean formulae with millions variables quickly enough for use in real-world applications [127]. Leading to their proliferation into several fields of scientific inquiry ranging from software engineering to Bioinformatics [88, 56].

The majority of research in the SAT community focuses on solving a single SAT problem as fast as possible, yet many practical applications of SAT solvers [118, 115, 133, 24, 47, 45, 54] require solving a set of related SAT problems [115, 118, 45]. To take just one example, SPL utilizes SAT solvers for a diverse range of analyses including: automated feature model analysis [18, 55, 122], feature model sampling [93, 128], anomaly detection [4, 77, 91], and dead code analysis [120].

This misalignment between the SAT research community and the practical use cases of SAT solvers is well known. To address the misalignment, modern solvers attempt to propagate information from one solving instance, on one problem, to future instances in the problem set. Initial attempts focused on clause sharing (CS) [115, 133] where learned clauses from one problem in the problem set are propagated forward to future problems. Although, modern solvers are based on a major breakthrough that occurred with *incremental SAT under assumptions*, introduced in `Minisat` [46].

Incremental SAT under assumptions, made two major contributions: a performance contribution, where information including learned clauses, restart and clause-detection heuristics are carried forward. A usability contribution; `Minisat` exposed an interface that allowed the end-user to directly program the solver. Through the interface the user can add or remove clauses and dictate which clauses or variables are shared and which are unique to the problem set, thus directly addressing the practical use case of SAT solvers.

Despite the its success, the incremental interface introduced a programming language that required an extra input, the set of SAT problems, *and* a program to direct the solver with side-effectual statements. This places further burden on the end-user: the system is less-declarative as the user must be concerned with the internals of the solver. A new class of errors is possible as the input program could misuse the introduced side-effectual statements. By requiring the user to direct the solver, the users' solution is specific to the exact set of satisfiability problems at hand, thus the programmed solution is specific to the problem set and therefore to the solver input. Should the user be interested in the assignment of variables under which the problem at hand was found to be satisfiable, then the user must create additional infrastructure to track results; which again couples to the input and is therefore difficult to reuse.

We argue that solving a set of related SAT problems *is a variational programming problem* and that by directly addressing the problem's variational nature the incremental SAT interface and performance can be improved. The essence of variational programming is a formal language called the *choice calculus*. With the choice calculus, sets of problems in the SAT domain can be expressed syntactically as a single *variational artifact*. The benefits are numerous:

1. The side-effectual statements are hidden from the user, recovering the declarative nature of non-incremental SAT solving.
2. Malformed programs built around the control flow operators become syntactically impossible.
3. The end-user's programmed solution is decoupled from the specific problem set, increasing software reuse.
4. The solver has enough syntactic information to produce results which previously required extra infrastructure constructed by the end-user.
5. Previously difficult optimizations can be syntactically detected and applied before the runtime of the solver.

This work is applied programming language theory in the domain of satisfiability solvers. Due to the ubiquity of satisfiability solvers estimating the impact is difficult although the surface area of possible applications is large. For example, many analyses in the software product-lines community use incremental SAT solvers. By creating a variational SAT solver such analyses directly benefit from this work, and thus advance the state of the art. For researchers in the incremental satisfiability solving community, this work serves as an avenue to construct new incremental SAT solvers which efficiently solve classes of problems that deal with variation.

For researchers studying variation the significance and impact is several fold. By utilizing results in variational research, this work adds validity to variational theory and serves as an empirical case study. At the time of this writing, and to my knowledge, this work is the first to directly use results in the variational research community to parallelize a variation unaware tool. Thus by directly handling variation, this work demonstrates direct benefits to be gained for researchers in other domains and magnifies the impact of any results produced by the variational research community. Lastly, the result of my thesis, a variational SAT solver, provides a new logic and tool to reason about variation itself.

1.2 Contributions and Outline of this Thesis

The high-level goal of this thesis is to use variation theory to formalize and construct a variational satisfiability solver that understands and can solve SAT problems that contain *variational values* in addition to boolean values. It is our desire that the work not only be of theoretical interest but of practical use. Thus, the thesis provides numerous examples of variational SAT and variational SMT problems to motivate and demonstrate the solver. The rest of this section outlines the thesis and expands on the contributions of each chapter:

1. Chapter 2 (*Background*) provides the necessary material for a reader to understand the contributions of the thesis. This section provides an overview of satisfiability solving, satisfiability-modulo theories solving, incremental SAT and SMT solving . Several important concepts are introduced: The definition of satisfiability and definition of the boolean satisfiability problem. The internal data structure incremental SAT solvers utilize to provide incrementality, and the side-effectual operations which manipulate the incremental solver and form the basis of variational satisfiability solving. Lastly, the definition of the output of a SAT or SMT solver which has implications for variational satisfiability solving and variational SMT.

2. Chapter 3 (*Variational Propositional Logic*) introduces a variational logic that a variational SAT solver operates upon. This section introduces the essential aspects of variation using propositional logic and in the process presents the first instance of a *variational system recipe* to construct a *variation-aware* system using a non-variational version of that system. Several variational concepts are defined and formalized which are used throughout the thesis, such as *variant*, *configuration* and *variational artifact*. Lastly, the section proves theorems that are required to prove the soundness of variational satisfiability solving. Major portions of this section are adapted from previous work [137].

3. Chapter 4 (*Variational Satisfiability Solving*) makes the central contribution of the thesis. In this chapter we define the general approach and architecture of a vari-

ational satisfiability solving. The general approach is the second presentation of the aforementioned recipe; in this case using a SAT solver rather than propositional logic. This section has been expanded from previous work [137], provides a rationale for our design and makes several important contributions:

(a) A formal semantics of variational satisfiability solving. A variational SAT problem is a description of the problem in variational propositional logic that is translated to an incremental SAT program which is suitable for execution on an incremental SAT solver.

(b) A formal definition of concepts such as a *variational core* which are transferable to domains other than SAT. Variational cores are key to our approach, specifically preserving shared terms between variants.

(c) A definition of a *variational compiler*. The compiler is defined as a variational fold which is the basis for the performance gains presented in the thesis. The folding algorithm has three phases to ensure that non-variational terms are shared across SAT problems and thus redundant computation is mitigated.

(d) A definition of the variational output that is returned to the user. The output presents several unique challenges that must be overcome while still being useful for the end user. We present and consider these concerns and provide a salable solution.

4. Chapter 5 (*Variational Satisfiability-Modulo Theory Solving*) extends the variational solving algorithm to consider SMT theories and propositions which include

numeric values such as Integers and Reals in addition to Booleans. We present the requisite extensions to the variational propositional logic, the variational compiler and solving algorithm, and extend the output to support types other than just Booleans. We demonstrate that our method fully generalizes to the core SMT theories in the SMTLIB2 standard.

5. Chapter 6 (*Case Studies*) The central project of this thesis is to evaluate the ideas of variational programming in satisfiability solving. Having defined and constructed a variational SAT and SMT solver in the previous chapters this chapter empirically prototype variational solvers. This chapter is adapted from work currently under peer review.

cite submission to emse

6. Chapter 7 (*Related Work*) is split into two sections. First, this thesis is situated into a lineage of recent variational-aware systems, thus this section collects this research and provides a comparison of our method to create a variational-aware system with previous methods. Second, this work is related to numerous SAT solvers that attempt to reuse information, solve sets of SAT problems and implement incremental SAT solving. We situate this work in the context of these solvers and compare their methods.

7. Chapter 8 (*Conclusion and Future Work*) summarizes the contributions of the thesis and relates the work to the central project of the thesis. In addition to the conceptual point, numerous areas of future work are discussed, such as further variational extensions, faster implementation strategies and the possibility to reuse our findings to create a variational Prolog [134, 78].

Chapter 2: Background

This section provides background on SAT and incremental SAT solving. It is intended as a general introduction to these concepts. Specific techniques or algorithms are not discussed in detail. All descriptions follow the SMT-LIB2 [15] standard and describe incremental solvers as a black box eliding internal details of any specific solver which adheres to the standard.

2.1 SMTLIB2 and Satisfiability Solving

This section provides assumes knowledge of propositional logic, and provides background to satisfiability solving and SMTLIB2 (smtlib); the standardized language for interacting with SAT solvers. Following the notation from the many-valued logic community [107] we refer to propositional logic as C_2, which denotes a two-valued logic.

A satisfiability solver is a software system that solves the Boolean Satisfiability Problem [108]. One of the oldest problems in computer science[1] and famously NP-complete [38], the Boolean satisfiability problem is the problem of determining if a formula (sometimes called a sentence) in propositional logic has an assignment of Boolean values to variables, such that under substitution the formula evaluates to `T`. We formalize the problem and terms in the following definitions:

[1] see Biere et al. [21] for a complete history from the ancients, through to George Boole to the modern day.

Definition 2.1.1 (Model)**.** Given a formula in propositional logic: $f \in C_2$, which contains a set of Boolean variables vs. A model, m, is a set of assignments of Boolean values to variables in f such that f evaluates to $\texttt{T}$, i.e., $m = \{(v \coloneqq b) \mid v \in vs, b \in \mathbb{B}\}$.

Corollary 2.1.0.1 (Validity)**.** *In propositional logic a formula or sentence is* valid *if it is true in* all *possible models [108]. That is, a valid formula or sentence is also a tautology.*

Definition 2.1.2 (Satisfiable)**.** Given a formula in propositional logic, f, which contains a set of Boolean variables vs. If there exists an assignment of variables to Boolean values such that f evaluates to $\texttt{T}$, then we say f is *satisfiable*.

For example, we can show that the formula $good = (a \wedge b) \vee c$ is satisfiable with the model: $\{(a \coloneqq \texttt{T}), (b \coloneqq \texttt{T}), (c \coloneqq \texttt{F})\}$, because $(\texttt{T} \wedge \texttt{T}) \vee \texttt{F}$ results in $\texttt{T}$. However, a formula such as $bad = (a \vee b) \wedge \texttt{F}$ is not satisfiable as no assignment of $\texttt{F}$ or $\texttt{T}$ to the variables a and b would allow bad to evaluate to $\texttt{T}$. With the preliminaries concepts we can now define the Boolean Satisfiability Problem:

Definition 2.1.3 (Boolean Satisfiability Problem)**.** Given a formula in propositional logic, f, determine if f is satisfiable.

While the formal definition of the Boolean Satisfiability Problem requires a formula in propositional logic, expressing a SAT problem in propositional logic can be cumbersome. Thus, modern satisfiability solvers accept programs written in domain specific programming languages to express SAT problems, communicate the problems to other people and dictate the problems to the solver. In recent years these programming languages have coalesced into a single standard via an international initiative called SMTLIB2.

The SMTLIB2 [15] standard formalizes a set of programming languages that define interactions with a SAT or SMT solver. The standard defines four languages, of which only two are used throughout this thesis: a *term* language; which defines a language for defining variables, functions and formulas in propositional and first-order logic. The *command* language; which defines a programming language to interact with the solver. The command language is used to add or remove formulas, query the solver for a model or check for satisfiability and other side-effectual interactions such as printing output.

For the remainder of this section we provide informal examples intended for a general audience and cover only the commands and concepts required for subsequent sections of this thesis. For a full language specification please see Barrett et al. [15].

define a listing language for smtlib

Consider this SMTLIB2 program which verifies peirce's law implies the law of excluded middle for propositional logic:

```
(declare-const a Bool)                     ;; variable declarations
(declare-const b Bool)
(define-fun ex-middle ((x Bool)) Bool      ;; excluded middle: x ∨ ¬x
  (or x
      (not x)))
(define-fun peirce ((x Bool) (y Bool)) Bool ;; peirce's law: ((x → y) → x) → x
  (=>
    (=> (=> x y)
        x)
    x))
(define-fun peirce-implies-ex-middle () Bool
  (=> (peirce a b)
      (ex-middle a)))
(assert (not peirce-implies-ex-middle))    ;; add assertion
(check-sat)                                ;; check SAT of all assertions
```

Comments begin with a semi-colon (;) and end at a new line. The program, and every SMTLIB2 program, is a sequence of *commands* that interact with the solver. For example, the above program consists of five commands, two variable declarations, a function definition, an assertion and a command to check satisfiability. Each command is formulated as an *s-expressions* [92] to simplify parsing [15]. For our purposes, one only needs to understand that commands and functions are called by opening parentheses; the first element after the opening parenthesis is the name of the command or function symbol, and subsequent elements are arguments to that command. Thus (declare-const a Bool) is an s-expression with three elements that defines the C_2 variable a of *sort* (called *type* in programming language literature) Bool. The first element, declare-const is the command name, the second is the user defined name for the variable and the third is its sort. Similarly, the s-expression (and a b) passes the variables a and b to the function and which returns the conjunction of these two variables. Lastly, the function definition define-fun takes four arguments: the user defined name; peirce-is-ex-middle, an s-expression that defines argument names and their sorts; ((x Bool) (y Bool)), a return sort; Bool and the body of the function. The command check-sat begins the solving subroutine to solve the described SAT or SMT problem. check-sat can return two values: SAT or UNSAT which corresponding to finding the SAT or SMT problem satisfiable or unsatisfiable.

Internally, a compliant solver such as z3 [43] maintains an stack called the *assertion stack* that tracks user provided variable and formula declarations and definitions. The elements of the assertion stack are called *levels* and are sets of *assertions*. An assertion is a logical formula, a declaration of a sort, or a definition of a function symbol. In the example, both variable declarations and the peirce-is-ex-middle definition are included in the assertion set. Sets of assertions are placed on the stack via the assert command.

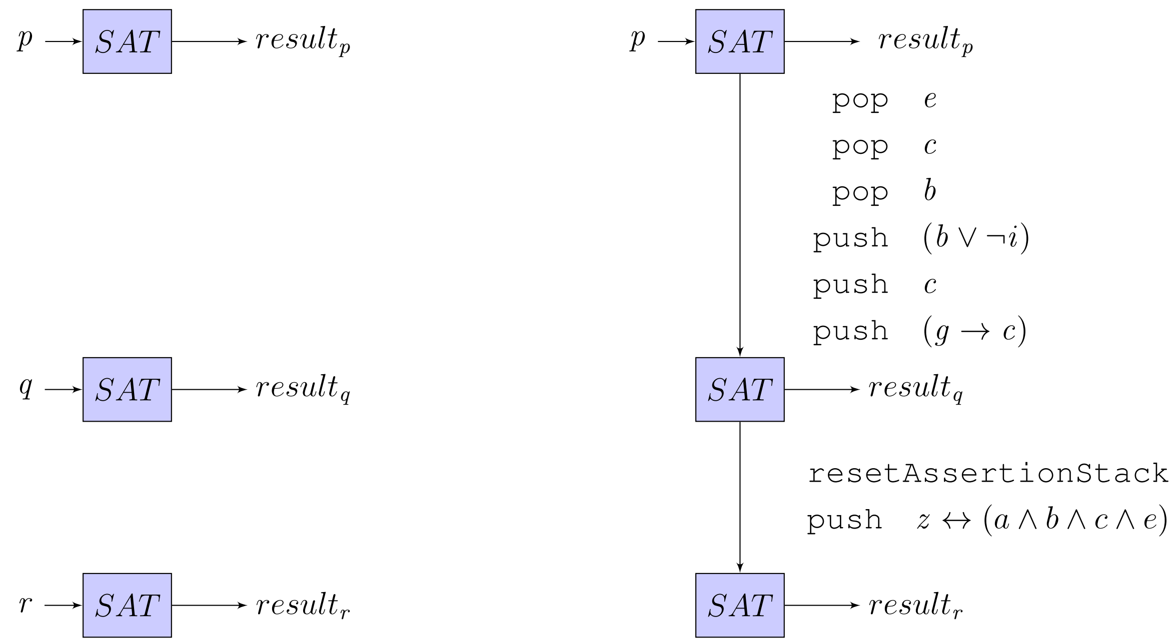
The assert command takes a term as input[2], collects all associated definitions and declarations and places the assertion set on the assertion stack.

The example demonstrates a common verification pattern in SAT and SMT solving. In the example, we construct a constraint that asserts (not peirce-is-ex-middle) rather than peirce-is-ex-middle because we need to verify that peirce's law implies the law of excluded middle for all possible models. Had we elided the not then the first model which satisfied the theorem would be returned, thus providing a *single* model where the theorem holds. However, to prove the theorem we need to show that it holds *for all* possible models. The not negates the theorem thus asking the solver to discover a counter-example to the theorem. If such a model exists, then the solver has discovered a counter-example to the theorem. If no such model exists—that is UNSAT is returned by the solver—then the negated theorem always evaluates to `F` and thus the theorem always evaluates to `T` and hence is logically valid.

Satisfiability and logical validity are closely related. Conceptually, satisfiability attempts to find a model that solves the constraints of a formula, while logical validity tries to show that the formula's truth-value is independent of it's variables and thus the formula is tautological. Similarly, where satisfiability is concerned with solving constraints, validity is concerned with finding a proof. Thus it is common to negate a formula to *query* the solver to search for a counter-example, when no such model is found and UNSAT is returned, we can be sure that the negation of a formula which is always `F` is a formula which is always `T`, and thus is logically valid.

[2]by the standard this is a *well-sorted term* of type *Bool*, however we elide this description for simplicity

2.2 Incremental Satisfiability Solving



(a) Brute force procedure, no reuse between solver calls.

(b) Incremental procedure, reuse defined by POP and PUSH.

Figure 2.1

Suppose, we have three related propositional formulas that we desire to solve.

$$p = a \wedge b \wedge c \wedge e \quad q = a \wedge (b \vee \neg i) \wedge c \wedge (g \rightarrow c) \quad r = z \leftrightarrow (a \wedge b \wedge c \wedge e)$$

p is simply a conjunction of variables. In q, relative to p, we see that two variables are added, i, g, e is removed, and there are two new clauses: $(b \vee \neg i)$ and $(g \rightarrow c)$, both of which possibly affect the values of b and c. In r, the variables and constraints introduced in p are further constrained to a new variable, z.

Suppose one wants to find a model for each formula. Using a non-incremental SAT solver results in the procedure illustrated in Fig. 2.1a; where SAT solving is a batch process and no information is reused. Alternatively, a procedure using an incremental SAT solver is illustrated in Fig. 2.1b; in this scenario, all formulas are solved by single solver instance where terms are programmatically added or removed from the solver.

The ability to add and remove terms is enabled by manipulating the *assertion stack*. The previous example demonstrated a single level on the stack, this example demonstrates three. The incremental interface provides two new commands: PUSH to create a new variable *scope* and add a level to the stack and POP to remove the level. The following program follows the procedure outlined in Fig. 2.1b and solves p, q and r:

```
(declare-const a Bool)            ;; variable declarations for p
(declare-const b Bool)
(declare-const c Bool)
(declare-const e Bool)
(assert a)                        ;; a is shared between p and q
(push)                            ;; solve p
  (assert e)
  (assert c)
  (assert b)
  (check-sat)                     ;; check-sat on p
(pop)                             ;; remove e, c, and b assertions
(push)                            ;; solve for q
  (declare-const i Bool)          ;; new variables
  (declare-const g Bool)
  (assert (or b (not i)))         ;; new clause
  (assert c)                      ;; re-add c
  (assert (=> g c))               ;; new clause
  (check-sat)                     ;; check sat of q
(pop)                             ;; i and g out of scope
(reset)                           ;; reset the assertion stack
(declare-const a Bool)            ;; variable declarations for r
(declare-const b Bool)
(declare-const c Bool)
```

```
(declare-const e Bool)
(declare-const z Bool)
(assert (= z (and a (and b (and c (and e ))))))
(check-sat)                          ;; check-sat on r
```

In this example we begin by defining p, we assert a outside of a new scope so that it can be reused for q. We reuse a by exploiting the conjunction of assertions per level on the assertion stack during a CHECK-SAT call. Had we asserted `(and a (and b (and c (and e))))` then we would not be able to reuse the assertion on a. The first PUSH command enters a new level on the assertion stack, the remaining variables are asserted and we issue a check-sat call. After the POP command, all assertions and declarations from the previous level are removed. Thus, after we solve q the variables i and g cannot be referenced as they are no longer in scope.

In an efficient process one would initially add as many *shared* terms as possible, such as a from p and then reuse that term as many times as needed. Thus, an efficient process should perform only enough manipulation of the assertion stack as required to reach the next SAT problem of interest from the current one. However, notice that doing so is not entirely straight forward; we were only able to reuse a from p in q because we defined p in a non-intuitive way by utilizing the internal behavior of the assertion stack. This situation is exacerbated by SAT problems such as r where due to the equivalence we were forced to completely remove everything on the stack in order to construct r. Thus incremental SAT solvers provide the primitive operations required to solve related SAT problems efficiently, yet writing the SMTLIB2 program to solve the set efficiently is not straightforward.

Chapter 3: Variational Propositional Logic

In this chapter, we present the logic of variational satisfiability problems. The logic is a conservative extension of classic two-valued logic (C_2) with a *choice* construct from the choice calculus [50, 131], a formal language for describing variation. We call the new logic VPL, short for variational propositional logic, and refer to VPL expressions as *variational formulas*. This chapter defines the syntax and semantics of VPL and concludes with a set of definitions, lemmas and theorems for the logic.

3.1 Syntax

The syntax of variational propositional logic is given in Fig. 3.1a. It extends the propositional formula notation of C_2 with a single new connective called a *choice* from the choice calculus. A choice $D\langle f_1, f_2\rangle$ represents either f_1 or f_2 depending on the Boolean value of its *dimension* D. We call f_1 and f_2 the *alternatives* of the choice. Although dimensions are Boolean variables, the set of dimensions is disjoint from the set of variables from C_2, which may be referenced in the leaves of a formula. We use lowercase letters to range over variables and uppercase letters for dimensions.

The syntax of VPL does not include derived logical connectives, such as $\rightarrow$ and $\leftrightarrow$. However, such forms can be defined from other primitives and are assumed throughout the thesis.

$$
\begin{array}{rcll}
t & ::= & r \quad | \quad \mathtt{T} \quad | \quad \mathtt{F} & \textit{Variables and Boolean literals} \\[4pt]
f & ::= & t & \textit{Terminal} \\
 & | & \neg f & \textit{Negate} \\
 & | & f \vee f & \textit{Or} \\
 & | & f \wedge f & \textit{And} \\
 & | & D\langle f, f\rangle & \textit{Choice}
\end{array}
$$

(a) Syntax of VPL.

$$
\begin{aligned}
[\![\cdot]\!] : f \to C \to f & \qquad\qquad \text{where } C : D \to \mathbb{B}_\perp \\
[\![t]\!]_C &= t \\
[\![\neg f]\!]_C &= \neg [\![f]\!]_C \\
[\![f_1 \wedge f_2]\!]_C &= [\![f_1]\!]_C \wedge [\![f_2]\!]_C \\
[\![f_1 \vee f_2]\!]_C &= [\![f_1]\!]_C \vee [\![f_2]\!]_C \\
[\![D\langle f_1, f_2\rangle]\!]_C &= \begin{cases} [\![f_1]\!]_C & C(D) = \mathsf{true} \\ [\![f_2]\!]_C & C(D) = \mathsf{false} \\ D\langle [\![f_1]\!]_C, [\![f_2]\!]_C \rangle & C(D) = \perp \end{cases}
\end{aligned}
$$

(b) Configuration semantics of VPL.

$$
\begin{aligned}
D\langle f, f\rangle &\equiv f && \textsc{Idemp} \\
D\langle D\langle f_1, f_2\rangle, f_3\rangle &\equiv D\langle f_1, f_3\rangle && \textsc{Dom-L} \\
D\langle f_1, D\langle f_2, f_3\rangle\rangle &\equiv D\langle f_1, f_3\rangle && \textsc{Dom-R} \\
D_1\langle D_2\langle f_1, f_2\rangle, D_2\langle f_3, f_4\rangle\rangle &\equiv D_2\langle D_1\langle f_1, f_3\rangle, D_1\langle f_2, f_4\rangle\rangle && \textsc{Swap} \\
D\langle \neg f_1, \neg f_2\rangle &\equiv \neg D\langle f_1, f_2\rangle && \textsc{Neg} \\
D\langle f_1 \vee f_3,\ f_2 \vee f_4\rangle &\equiv D\langle f_1, f_2\rangle \vee D\langle f_3, f_4\rangle && \textsc{Or} \\
D\langle f_1 \wedge f_3,\ f_2 \wedge f_4\rangle &\equiv D\langle f_1, f_2\rangle \wedge D\langle f_3, f_4\rangle && \textsc{And} \\
D\langle f_1 \wedge f_2, f_1\rangle &\equiv f_1 \wedge D\langle f_2, \mathtt{T}\rangle && \textsc{And-L} \\
D\langle f_1 \vee f_2, f_1\rangle &\equiv f_1 \vee D\langle f_2, \mathtt{F}\rangle && \textsc{Or-L} \\
D\langle f_1, f_1 \wedge f_2\rangle &\equiv f_1 \wedge D\langle \mathtt{T}, f_2\rangle && \textsc{And-R} \\
D\langle f_1, f_1 \vee f_2\rangle &\equiv f_1 \vee D\langle \mathtt{F}, f_2\rangle && \textsc{Or-R}
\end{aligned}
$$

(c) VPL equivalence laws.

Figure 3.1: Formal definition of VPL.

3.2 Semantics

Conceptually, a variational formula represents several propositional logic formulas at once, which can be obtained by resolving all of the choices. For software product-line researchers, it is useful to think of VPL as analogous to `#ifdef`-annotated C_2, where choices correspond to a disciplined [85] application of `#ifdef` annotations. From a logical perspective, following the many-valued logic of Kleene [76, 107], the intuition behind VPL is that a choice is a placeholder for two equally possible alternatives that is deterministically resolved by reference to an external environment. In this sense, VPL deviates from other many-valued logics, such as modal logic [57], because a choice *waits* until there is enough information in an external environment to choose an alternative (i.e., until the formula is *configured*).

The *configuration semantics* of VPL is given in Fig. 3.1b and describes how choices are eliminated from a formula. The semantics is parameterized by a *configuration* C, which is a partial function from dimensions to Boolean values. The first four cases of the semantics simply propagate configuration down the formula, terminating at the leaves. The case for choices is the interesting one: if the dimension of the choice is defined in the configuration, then the choice is replaced by its left or right alternative corresponding to the associated value of the dimension in the configuration. If the dimension is undefined in the configuration, then the choice is left intact and configuration propagates into the choice's alternatives.

If a configuration C eliminates all choices in a formula f, we call C *total* with respect to f. If C does *not* eliminate all choices in f (i.e., a dimension used in f is undefined in C), we call C *partial* with respect to f. We call a choice-free formula *plain*, and call the set of all plain formulas that can be obtained from f (by configuring it with every possible total configuration) the *variants* of f.

To illustrate the semantics of VPL, consider the formula $p \wedge A\langle q, r\rangle$, which has two variants: $p \wedge q$ when $C(A) = \mathsf{true}$ and $p \wedge r$ when $C(A) = \mathsf{false}$. From the semantics, it follows that choices in the same dimension are *synchronized* while choices in different dimensions are *independent*. For example, $A\langle p, q\rangle \wedge B\langle r, s\rangle$ has four variants, while $A\langle p, q\rangle \wedge A\langle r, s\rangle$ has only two ($p \wedge r$ and $q \wedge s$). It also follows from the semantics that nested choices in the same dimension contain redundant alternatives; that is, inner choices are *dominated* by outer choices in the same dimension. For example, $A\langle p, A\langle r, s\rangle\rangle$ is equivalent to $A\langle p, s\rangle$ since the alternative r cannot be reached by any configuration. As the previous example illustrates, the representation of a VPL formula is not unique; that is, the same set of variants may be encoded by different formulas. Fig. 3.1c defines a set of equivalence laws for VPL formulas. These laws follow directly from the configuration semantics in Fig. 3.1b and can be used to derive semantics-preserving transformations of VPL formulas. For example, we can simplify the formula $A\langle p \vee q, p \vee r\rangle$ by first applying the OR law to obtain $A\langle p, p\rangle \vee A\langle q, r\rangle$, then applying the IDEMP law to the first argument to obtain $p \vee A\langle q, r\rangle$ in which the redundant p has been factored out of the choice.

3.3 Formalisms

Having defined the syntax and semantics of VPL the rest of this chapter will define useful functions and properties. We conclude the chapter with an example of encoding a set of C_2 formulas to a single VPL formula. First define useful functions to retrieve interesting aspects of VPL formulas.

Definition 3.3.1 (Dimensions)**.** Given a formula $f \in$ VPL, let $\mathit{Dimensions}(f)$ be the set of unique dimensions in the formula: $\mathit{Dimensions}(f) = \{D \mid D \in f\}$.

For example, $\mathit{Dimensions}(A\langle p, q\rangle \wedge B\langle r, s\rangle)$ = $\{A, B\}$ and $\mathit{Dimensions}(A\langle p, q\rangle \wedge A\langle r, s\rangle)$ = $\{A\}$. Similarly we define a notion of *cardinality* over VPL formulas.

Definition 3.3.2 (Dimension-cardinality)**.** The dimension-cardinality or d-cardinality of a formula $f \in$ VPL is the cardinality of the set of unique dimensions in a formula. We use the following notation as shorthand: $|f|_D = |\mathit{Dimensions}(f)|$.

Similarly to $\mathit{Dimensions}$ it is useful to have projections from a VPL formula to possible variants:

Definition 3.3.3 (Variants)**.** Given a formula $f \in$ VPL, let $\mathit{Variants}(f)$ be the set of all possible *variants* of f. Thus, $\mathit{Variants}(f) = \{\ v \mid \exists C.\ [\![f]\!]_C = v,\ v \neq f\ \}$

and we can define a projection for all plain variants as well:

Definition 3.3.4 (C_2 Variants)**.** Given a formula $f \in$ VPL, let $\mathit{Variants}_{C_2}(f)$ be the set of all possible *plain variants* of f. Thus, $\mathit{Variants}_{C_2}(f) = \{\ v \mid \exists C.\ [\![f]\!]_C = v,\ v \in C_2\ \}$

Using $Dimensions$ we can define a more precise property on configurations.

Definition 3.3.5 (Minimal Configuration)**.** We say a configuration C is minimal with respect to some formula $f \in$ VPL if and only if $Dom(C) = Dimensions(f)$.

Note that a minimal configuration can only be $\varnothing$ for formulas which are plain. One may think of a minimal configuration as a total configuration with *nothing extra*. For example, the configuration $C = \{(A, \mathsf{true}), (B, \mathsf{false}), (E, \mathsf{true})\}$ is total with respect to the formula $f = A\langle p, q\rangle \wedge B\langle r, s\rangle$ because C eliminates all choices in f. However C is not minimal with respect to f because $Dom(C) = \{A, B, E\}$ and $Dimensions(f) = \{A, B\}$, thus $\{A, B, E\} \neq \{A, B\}$ since C contains an extra binding for E that is not needed to configure f.

With these functions and definitions we can prove useful lemmas and theorems. We'll begin by proving that the configuration semantics over a VPL formula is *confluent*, i.e., each configuration with respect to a VPL formula precisely specifies one variant:

Lemma 3.3.1 ($[\![f]\!]_C$ is deterministic)**.** *For any configuration C, if $[\![f]\!]_C = g$, and $[\![f]\!]_C = h$, then $g = h$.*

Proof. As shown in Fig. 3.1b, there is only one applicable case for each relation in VPL. Furthermore, by the property of synchronization, each choice must be configured to the same alternative, thus $[\![f]\!]_C$ is deterministic for some C and f. □

Similarly, clearly VPL reduces to C_2:

Theorem 3.3.2 (VPL reducible to C_2)**.** *For any configuration C and any formula $f \in$ VPL, if C is total with respect to f, then $[\![f]\!]_C \in C_2$*

Proof. This follows directly from the semantics of configuration in Fig. 3.1b, the definition of a total configuration, and Lemma 3.3.1. The proof is by structural induction on f and case analysis. The only interesting case is the case for choices. Since C is total we have $C : D \to \mathbb{B}$ instead of $C : D \to \mathbb{B}_{\perp}$, thus the last case for choices, where $C(D) = \perp$, can never happen and therefore configuration of a formula f with a total configuration is a total function. The other base case is over terminals, which are in C_2 by definition, and each other case propagate the total configuration to a base case. Thus each choice and its' alternatives are recursively reified for f, and by definition a VPL formula which lacks choices is $\in C_2$. □

3.4 Example

To demonstrate the application of VPL and conclude the chapter, we encode the incremental example from Chapter 2. Our goal is to construct a single variational formula that encodes the related plain formulas p, q, r. Ideally, this variational formula should maximize sharing among the plain formulas in order to avoid redundant analyses during a variational solving. We reproduce the formulas below for the convenience:

$$p = a \wedge b \wedge c \wedge e \quad q = a \wedge (b \vee \neg i) \wedge c \wedge (g \to c) \quad r = z \leftrightarrow (a \wedge b \wedge c \wedge e)$$

Every set of plain formulas can be encoded as a variational formula systematically by first constructing a nested choice containing all of the individual variables as alternatives, then factoring out shared subexpressions by applying the laws in Fig. 3.1c.

Unfortunately, the process of globally minimizing a variational formula in this way is hard[1] since we must apply an arbitrary number of laws right-to-left in order to set up a particular sequence of left-to-right applications that factor out commonalities.

Due to the difficulty of minimization, we instead demonstrate how one can build such a formula *incrementally*. Our variational formula will use the dimensions P, Q, R to represent the respective variants. Unique portions of each variant will be nested into the left alternative so that the unique portion is considered when the dimension is bound to true in the configuration.

We begin by combining p and r since the differences[2] between the two are smaller than between other pairs of feature models in our example. Feature models may be combined in any order as long as the variants in the resulting formula correspond to their plain counterparts. The only change between p and r is the addition of z and thus we wrap the leaf in a choice with dimension R. This yields the following variational formula.

$$f_{pr} = R\langle z, \mathtt{T}\rangle \leftrightarrow (a \wedge b \wedge c \wedge e)$$

We exploit the fact that $\wedge$ forms a monoid with $\mathtt{T}$ to recover a formula equivalent to p for configurations where R is false.

Next we combine f_{pr} with q to obtain a variational formula that encodes the propositional formulas p, q, r. There are two sub-trees that must be wrapped in choices. First, we must encode the difference between $b \vee \neg i$ from q and b. Second, we must en-

[1]. We hypothesize that it is equivalent to BDD minimization, which is NP-complete, but the equivalence has not been proved; see [132].

[2]There are many ways to assess the difference of two formulas. For now the reader may consider it reducible to the Levenshtein distance of two strings [84]. We return to this discussion in Section 8.2

sure synchronization and thus use the same dimension to encode the difference between $g \rightarrow c$ and e. Thus the resulting variational formula is:

$$f = R\langle z, \texttt{T}\rangle \leftrightarrow (a \wedge Q\langle b \vee \neg i, b\rangle \wedge c \wedge Q\langle g \rightarrow c, e\rangle)$$

Now that we have constructed the variational formula we need to ensure that it encodes all variants of interest and nothing else. Notice that only 2 dimensions are used to encode 3 variants, because $|f|_D$ = 2 we have are 4 possible variants and thus one extra variant. We can observe this by enumerating the variants and possible configurations:

$$\begin{aligned} p =&\ \texttt{T} \leftrightarrow (a \wedge b \wedge c \wedge e) & C = \{(R, \mathsf{false}), (Q, \mathsf{false})\} \\ q =&\ \texttt{T} \leftrightarrow (a \wedge (b \vee \neg i) \wedge c \wedge (g \rightarrow c)) & C = \{(R, \mathsf{false}), (Q, \mathsf{true})\} \\ r =&\ z \leftrightarrow (a \wedge b \wedge c \wedge e) & C = \{(R, \mathsf{true}), (Q, \mathsf{false})\} \\ \mathit{extra} =&\ z \leftrightarrow (a \wedge (b \vee \neg i) \wedge c \wedge (g \rightarrow c)) & C = \{(R, \mathsf{true}), (Q, \mathsf{true})\} \end{aligned}$$

Notice the extra variant and that p and q are only recovered through equivalency laws from propositional logic. While it is undesirable that there exists extra variants, the important constraint: $\{p, q, r\} \subseteq \mathit{Variants}_{C_2}(f)$ is satisfied. We'll return to the case of extra variants in the next chapter by showing how to prevent a variational SAT solver from solving these variants.

Chapter 4: Variational Satisfiability Solving

This chapter presents the variational satisfiability solving algorithm. Section 4.1 provides an overview of the algorithm and introduces the notion of *variational models* as solutions to variational satisfiability problems. Section 4.5 provides the formal specification. We conclude the chapter by proving that the variational solving algorithm has the property of variation preservation, and is therefore sound up to the soundness of the underlying incremental SAT solver.

4.1 General Approach

VPL formulas are solved recursively; decoupling the handling of plain terms from the handling of variational terms. The intuition behind the algorithm is to first process as many plain terms as possible (e.g. by pushing those terms to the underlying solver) while skipping choices, yielding a *variational core* that represents only the variational parts of the original formula. We then alternate between configuring choices in the variational core and processing the new plain terms produced by configuration until the entire term has been consumed. A variant in of the original VPL formula is found every time the entire term is consumed, since all choices will have been configured. Once a variant has been found the algorithm queries the underlying solver to obtain a model, then backtracks to solve a different variant by configuring the same choices differently. The

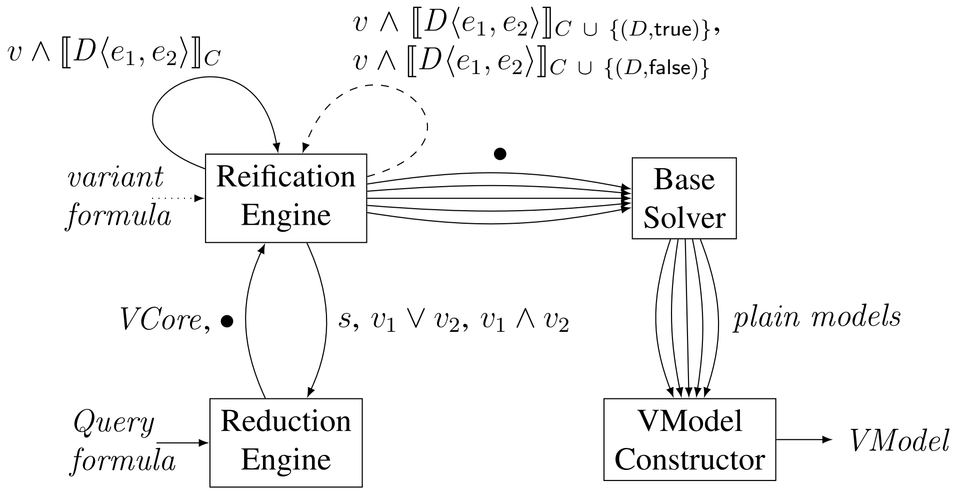


Figure 4.1: System overview of the variational solver.

models for each variant are combined into a single *variational model* that captures the result of solving all variants of the original VPL formula. Crucially building a variational model is associative, and thus the order the variants' models are found is not important to the correctness of the final model.

We present an overview of the variational solver as a state diagram in Fig. 4.1 that operates on the input's abstract syntax tree. Labels on incoming edges denote inputs to a state and labels on outgoing edges denote return values; we show only inputs for recursive edges; labels separated by a comma share the edge. We omit labels that can be derived from the logical properties of connectives, such as commutativity of $\vee$ and $\wedge$. Similarly, we omit base case edge labels for choices and describe these cases in the text.

The solver has four subsystems: The *reduction engine* processes plain terms and generates the variational core, which is ready for reification. The *reification engine* configures choices in a variational core. The *base solver* is the incremental solver used to produce plain models. Finally, the *variational model constructor* synthesizes a single variational model from the set of plain models returned by the base solver.

The solver takes a VPL formula called a *query formula* and an optional input called a *variation context* (vc). A vc is a propositional formula of dimensions that restricts the solver to a subset of variants, thus prevents computation on extra variants. The variational solver translates the query formula to a formula in an intermediate language (IL) that the reduction and reification engines operate over. The syntax of the IL is given below.

$$v \;::=\; \bullet \mid t \mid r \mid s \mid \neg v \mid (v \wedge v) \mid v \vee v \mid D\langle e, e\rangle$$

The IL includes two kinds of terminals not present in the input query formulas: plain subterms that can be reduced symbolically will be replaced by a reference to a *symbolic value* s, and subterms that have been sent to the base solver will be represented by the unit value $\bullet$. Note that choices contain unprocessed expressions (e) as alternatives.

4.2 Derivation of a Variational Core

A variational core is an IL formula that captures the variational structure of a query formula. Plain terms will either be placed on the assertion stack or will be symbolically reduced, leaving only logical connectives, symbolic references, and choices.

The variational core for a VPL formula is computed by a reduction engine illustrated in Fig. 4.2. The reduction engine has two states: *evaluation*, which communicates to the base solver to process plain terms, and *accumulation*, which is called by evaluation to

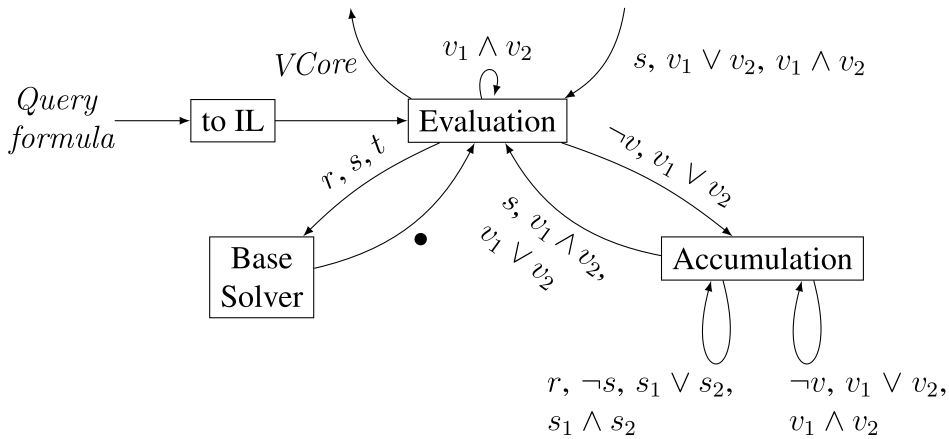


Figure 4.2: Overview of the reduction engine.

create symbolic references and reduce plain formulas.

To illustrate how the reduction engine computes a variational core, consider the query formula $f = ((a \wedge b) \wedge A\langle e_1, e_2\rangle) \wedge ((p \wedge \neg q) \vee B\langle e_3, e_4\rangle)$. Translated to an IL formula, f has four references (a, b, p, q) and two choices. The reduction engine will ultimately produce a variational core that asserts $(a \wedge b)$ in the base solver, thus pushing it onto the assertion stack, and create a symbolic reference for $(p \wedge \neg q)$.

Generating the core begins with evaluation. Evaluation matches on the root $\wedge$ node of f and recurs following the $v_1 \wedge v_2$ edge, where $v_1 = (a \wedge b) \wedge A\langle e_1, e_2\rangle$ and $v_2 = (p \wedge \neg q) \vee B\langle e_3, e_4\rangle$. The recursion processes the left child first. Thus, evaluation again matches on $\wedge$ of v_1 creating another recursive call with $v_1' = (a \wedge b)$ and $v_2' = A\langle e_1, e_2\rangle$. Finally, the base case is reached with a final recursive call where $v_1'' = a$, and $v_2'' = b$. At the base case, both a and b are references, so evaluation sends a to the base solver following the r, s, t edge, which returns $\bullet$ for the left child. The right child follows the same process yielding $\bullet \wedge \bullet$. Since the assertion stack implicitly conjuncts all assertions, $\bullet \wedge \bullet$ will be further reduced to $\bullet$ and returned as the result of v_1', indicating that both children have been pushed to the base solver. This leaves $v_1' = \bullet$ and $v_2' = A\langle e_1, e_2\rangle$. v_2' is a base case for choices and cannot be reduced in evaluation, so $\bullet \wedge A\langle e_1, e_2\rangle$ will be reduced to just $A\langle e_1, e_2\rangle$ as the result for v_1.

In evaluation, conjunctions can be split because of the behavior of the assertion stack and the and-elimination property of $\wedge$. Disjunctions and negations cannot be split in this way because both cannot be performed if a child node has been lost to the solver, e.g., $\neg \bullet$. Thus, in accumulation, we construct symbolic terms to represent entire subtrees, which ensures information is not lost while still allowing for the subtree to be evaluated if it is sound to do so.

The right child, $v_2 = (p \wedge \neg q) \vee B\langle e_3, e_4\rangle$ requires accumulation. Evaluation will match on the root $\vee$ and send $(p \wedge \neg q) \vee B\langle e_3, e_4\rangle$ to accumulation via the $v_1 \vee v_2$ edge. Accumulation has two self-loops, one to create symbolic references (with labels $r, s, \ldots$), and one to recur to values. Accumulation matches the root $\vee$ and recurs on the self-loop with edge $v_1 \vee v_2$, where $v_1 = (p \wedge \neg q)$ and $v_2 = B\langle e_3, e_4\rangle$. Processing the left child first, accumulation will recur again with $v_1' = p$ and $v_2' = \neg q$. $v_1' = p$ is a base case for references, so a unique symbolic reference s_p is generated for p following the self-loop with label r and returned as the result for v_1'. v_2' will follow the self-loop with label $\neg v$ to recur through $\neg$ to q, where a symbolic term s_q will be generated and returned. This yields $\neg s_q$, which follows the $\neg s$ edge to be processed into a new symbolic term, yielding the result for v_2' as $s_{\neg q}$. With both results $v_1 = s_p \wedge s_{\neg q}$, accumulation will match on $\wedge$ *and* both s_p and $s_{\neg q}$ to accumulate the entire subtree to a single symbolic term, s_{pq}, which will be returned as the result for v_1. v_2 is a base case, so accumulation will return $s_{pq} \vee B\langle e_3, e_4\rangle$ to evaluation. Evaluation will conclude with $A\langle e_1, e_2\rangle$ as the result for the left child of $\wedge$ and $s_{pq} \vee B\langle e_3, e_4\rangle$ for the right child, yielding $A\langle e_1, e_2\rangle \wedge s_{pq} \vee B\langle e_3, e_4\rangle$ as the variational core of f.

A variational core is derived to save redundant work. If solved naively, plain sub-formulas of f, such as $a \wedge b$ and $p \wedge \neg q$, would be processed once for each variant even though they are unchanged. Evaluation moves sub-formulas into the solver state to be reused among different variants. Accumulation caches sub-formulas that cannot be immediately evaluated to be evaluated later.

Symbolic references are variables in the reduction engine's memory that represent a set of statements in the base solver.[1] For example, s_{pq} represents three declarations in the base solver:

```
(declare-const p Bool)
(declare-const q Bool)
(declare-fun s_pq () Bool (and p (not q)))
```

Similarly a variational core is a sequence of statements in the base solver with holes $\Diamond$. For example, the variational core of f would be encoded as:

```
(assert (and a b))                    ;; add a ∧ b to the assertion stack
(declare-const ◊)                     ;; choice A
  ⋮                                   ;; potentially many declarations and assertions
(declare-fun s_pq () Bool (and p q))  ;; get symbolic reference for s_pq
(declare-const ◊)                     ;; choice B
  ⋮                                   ;; potentially many declarations and assertions
(assert (or s_ab ◊))                  ;; assert waiting on ⟦B⟨e3, e4⟩⟧_C
```

[1]Note that while we use SMTLIB2 as an implementation target, any solver that exposes an incremental API as defined by minisat [100] can be used to implement variational satisfiability solving.

Each hole is filled by configuring a choice and may require multiple statements to process the alternative.

4.3 Solving the Variational Core

The reduction engine performs the work at each recursive step whereas the reification engine defines transitions between the recursive steps by manipulating the configuration. In Chapter 3, we formalized a configuration as a function $D \to \mathbb{B}$, which we encode in the solver as a set of tuples $\{D \times \mathbb{B}\}$. Fig. 4.1 shows two loops for the reification engine corresponding to the reification of choices. The edges from the reification engine to the reduction engine are transitions taken after a choice is removed, where new plain terms have been introduced and thus a new core is derived. If the user supplied a variation context, then it is used to check that the binding of a Boolean value to a dimension is valid in the variation context. For example, $vc = \neg A$ would prevent any configurations where $(A, \mathsf{true}) \in C$. Finally, a model is retrieved from the base solver when the reduction engine returns $\bullet$, indicating that a variant has been reached.

We show the edges of the reification engine relating to the $\wedge$ connective; the edges for the $\vee$ connective are similar. The left edge is taken when a choice is observed in the variational core: $v \wedge [\![D\langle e_1, e_2\rangle]\!]_C$ and $D \in C$. This edge reduces choices with dimension D to an alternative, which is then translated to IL. The right edge is dashed to indicate assertion stack manipulation and is taken when $D \notin C$. For this edge, the configuration is mutated for both alternatives: $C \cup \{(D, \mathsf{true})\}$ and $C \cup \{(D, \mathsf{false})\}$, and the recursive call is wrapped with a PUSH and POP command. To the base solver, this branching appears as a linear sequence of assertion stack manipulations that performs backtracking behavior. For example, the representation of f is:

```
  ⋮            ;; declarations and assertions from variational core
(push 1)       ;; a configuration on B has occurred
  ⋮            ;; new declarations for left alternative
(declare-fun s () Bool (or s_pq ◊[◊ → s_B_T]))  ;; fill
(assert s)
  ⋮            ;; recursive processing
(pop 1)        ;; return for the right alternative
(push 1)       ;; repeat for right alternative
```

Where the hole $\Diamond$, will be filled with a newly defined variable s_{D_T} that represents the left alternative's formula.

4.4 Variational Models

Classic SAT models map variables to Boolean values; variational models map variables to variation contexts that record the variants where the variable was assigned $\mathtt{T}$. The variational context for a variable r in a variational model is denoted as vc_r. A variational model reserves a special variable called $\_Sat$ to track the configurations that were found satisfiable. We use the notation M_v^f to mean the variational model produced by solving a variational formula f, and $M_v^f(C)$ to mean the plain model which results from substitution of a configuration C into the variational model M_v^f. As an example, consider an altered version of the query formula from the previous section $f = ((a \wedge \neg b) \wedge A\langle a \rightarrow \neg p, c\rangle) \wedge ((p \wedge \neg q) \vee B\langle q, p\rangle)$. We can easily see that one variant, with configuration $\{(A, \mathtt{T}), (B, \mathtt{F})\}$ is unsatisfiable. If the remaining variants are satisfiable, then three models would result, as illustrated in Fig. 4.3; with the

$$\begin{array}{ccc}
a \rightarrow \mathtt{T} & a \rightarrow \mathtt{T} & a \rightarrow \mathtt{T} \\
b \rightarrow \mathtt{F} & b \rightarrow \mathtt{F} & b \rightarrow \mathtt{F} \\
c \rightarrow \mathtt{T} & c \rightarrow \mathtt{T} & \\
p \rightarrow \mathtt{T} & p \rightarrow \mathtt{T} & p \rightarrow \mathtt{F} \\
q \rightarrow \mathtt{F} & q \rightarrow \mathtt{F} & q \rightarrow \mathtt{T} \\
C_{FF} = \{(A, \mathtt{F}), (B, \mathtt{F})\} & C_{FT} = \{(A, \mathtt{F}), (B, \mathtt{T})\} & C_{TT} = \{(A, \mathtt{T}), (B, \mathtt{T})\}
\end{array}$$

Figure 4.3: Possible plain models for variants of f.

$$\begin{array}{ll}
_Sat \rightarrow & (\neg A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge B) \\
a \rightarrow & (\neg A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge B) \\
b \rightarrow & \mathtt{F} \\
c \rightarrow & (\neg A \wedge \neg B) \vee (\neg A \wedge B) \\
p \rightarrow & (\neg A \wedge \neg B) \vee (\neg A \wedge B) \\
q \rightarrow & (A \wedge B)
\end{array}$$

Figure 4.4: Variational model corresponding to the plain models in Fig. 4.3.

corresponding variational model shown in Fig. 4.4.

We see that $vc_{\_Sat}$ consists of three disjuncted terms, one for each satisfiable variant. Variational models are flexible; a satisfiable assignment of the query formula can be found by calling SAT on $vc_{\_Sat}$. Assuming the model $C_{FT} = \{(A, \mathtt{F}), (B, \mathtt{T})\}$ is returned, one can find a variable's value through substitution with the configuration; for example, substituting the returned model on vc_c yields:

$$\begin{array}{lr}
c \rightarrow (\neg A \wedge \neg B) \vee (\neg A \wedge B) & vc \text{ for } c \\
c \rightarrow (\neg \mathtt{F} \wedge \neg \mathtt{T}) \vee (\neg \mathtt{F} \wedge \mathtt{T}) & \text{Substitute } \mathtt{F} \text{ for } A, \mathtt{T} \text{ for } B \\
c \rightarrow \mathtt{T} & \text{Result}
\end{array}$$

Furthermore, finding variants where a variable such as c is satisfiable is reduced to

$SAT(vc_c)$

Variational models are constructed incrementally by merging each new plain model returned by the solver into the variational model. A merge requires the current configuration, the plain model, and current vc of a variable. Variables are initialized to F. For each variable i in the model, if i's assignment is T in the plain model, then the configuration is translated to a variation context and disjuncted with vc_i. For example, to merge the C_{FT}'s plain model to the variational model in Fig. 4.4, C_{FT}'s configuration is converted to $\neg A \wedge B$. This clause is disjuncted for variables assigned T in the plain model: vc_a, vc_c, and vc_p, even if they are new (e.g., c). Variables assigned F are skipped, thus vc_q remains F. For example, in the next model C_{TT}, c is F thus vc_c remains unaltered, while vc_q flips to T hence vc_q records $A \wedge B$. Variables such as b, whose vc's stay F are called *constant*.

Variational models are constructed in disjunctive normal form (DNF), and form a monoid with $\vee$ as the semigroup operation, and F as the unit value. We note this for mathematically inclined readers and those looking to implement their own variational solver because it is an important property for asynchronous implementations of variational satisfiability solvers.

4.5 Formalization

In this section we formalize variational SAT solving by specifying the semantics of the *accumulation* and *evaluation* phases of the variational solver, as well as the semantics of processing the variational core, which we call *choice removal*. Variational SAT solving

$$\begin{array}{rcllll} \texttt{Not} & : & (\Delta, s) & \rightarrow & (\Delta, s) & \textit{Negate a symbolic value} \\ \texttt{And} & : & (\Delta, s, s) & \rightarrow & (\Delta, s) & \textit{Conjunction of symbolic values} \\ \texttt{Or} & : & (\Delta, s, s) & \rightarrow & (\Delta, s) & \textit{Disjunction of symbolic values} \\ \texttt{Var} & : & (\Delta, r) & \rightarrow & (\Delta, s) & \textit{Create symbolic value based on a variable} \\ \texttt{Assert} & : & (\Gamma, \Delta, s) & \rightarrow & \Gamma & \textit{Assert a symbolic value to the solver} \\ \texttt{GetModel} & : & (\Gamma, \Delta) & \rightarrow & m & \textit{Get a model for the current solver state} \end{array}$$

Figure 4.5: Assumed base solver primitive operations.

assumes the existence of an underlying incremental SAT solver, which we refer to as the *base solver*.

The variational solver interacts with the base solver via several primitive operations. In our semantics, we simulate the effects of these primitive operations by tracking their effects on two stores. The *accumulation store* Δ tracks values cached during accumulation by mapping IL terms to symbolic references. The *evaluation store* Γ tracks the symbolic references that have been sent to the base solver during evaluation.

4.5.1 Primitives

Fig. 4.5 lists a minimal set of primitive operations that the base solver is assumed to support. These primitive operations define the interface between the base solver and the variational solver.

The primitive operations can be roughly grouped into two categories: The first four operations, consisting of the logical operations `Not`, `And`, and `Or`, plus the `Var` operation, are used in the accumulation phase and are concerned with creating and maintaining symbolic references that may stand for arbitrarily complex subtrees of the original formula. These operations simulate caching information in the base solver. The final two operations, `Assert` and `GetModel`, are used in the evaluation phase and simulate pushing new assertions to the base solver and obtaining a satisfiability model based on the current solver state, respectively.

It's important to note that our primitive operations are pure functions and do not simulate interacting with the base solver via side effects. The effect of a primitive operation can be determined by observing its type. For example, the `Assert` operation pushes new assertions to the base solver. This is reflected in its type, which includes an evaluation store as input and produces a new evaluation store (with the assertion included) as output. Since evaluation stores are immutable, we do not need a primitive operation to simulate popping assertions from the base solver. Instead, we simulate this by directly reusing old evaluation stores.

Many of the primitive operations operate on references to symbolic values. Such symbolic references may stand for arbitrarily complex subtrees of the original formula, built up through successive calls to the corresponding primitive operations. For example, recall the example formula $p \wedge \neg q$ from Section 4.1, which was replaced by the symbolic value s_{pq} after the following sequence of smtlib declarations.

```
(declare-const p Bool)
(declare-const q Bool)
(declare-fun s_pq () Bool (and p (not q)))
```

In our formalization, we would represent this same transformation of the formula $p \wedge \neg q$ into a symbolic reference s_{pq} using the following sequence of primitive operations:

$$\texttt{Var}(\Delta_0, p) = (\Delta_1, s_p)$$
$$\texttt{Var}(\Delta_1, q) = (\Delta_2, s_q)$$
$$\texttt{Not}(\Delta_2, s_q) = (\Delta_3, s'_q)$$
$$\texttt{And}(\Delta_3, s_p, s'_q) = (\Delta_4, s_{pq})$$

The accumulation store tracks what information is associated with each symbolic reference. The store must therefore be threaded through the calls to each primitive operation so that subsequent operations have access to existing definitions and can produce a new, updated store. For example, the final store produced by the above example contains the following mappings from IL terms to symbolic references, $\Delta_4 = \{(p, s_p), (q, s_q), (\neg s_q, s'_q), (s_p \wedge s'_q, s_{pq})\}$.

When comparing the smtlib notation to our formalization, observe that each use of declare-const corresponds to a use of the `Var` primitive, while the declare-fun line in smtlib may potentially expand into several primitive operations in our formalization. For the evaluation primitives, the `Assert` operation corresponds to an smtlib assert call, while the `GetModel` operation corresponds roughly to an smtlib check-sat call, which retrieves a model for the current set of assertions on the stack. However, the exact semantics of check-sat depends on the base solver in use. For example, given the plain formula $p = a \vee b \vee c$, z3 returns only a minimal satisifiable model, such as $\{b = \texttt{T}\}$, providing no values for the other variables in the formula. To normalize this behavior across solvers, we instead consider `GetModel` equivalent to check-sat followed by a get-value call call for every variable in the query formula, yielding a complete model.

$$\underline{\texttt{Var}}(\Delta, r) = \begin{cases} (\Delta, s) & (r, s) \in \Delta \\ \texttt{Var}(\Delta, r) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Not}}(\Delta, s) = \begin{cases} (\Delta, s') & (\neg s, s') \in \Delta \\ \texttt{Not}(\Delta, s) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{And}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 \wedge s_2, s_3) \in \Delta \\ \texttt{And}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Or}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 \vee s_2, s_3) \in \Delta \\ \texttt{Or}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

Figure 4.6: Wrapped accumulation primitive operations.

For example, a complete model for p would be $\{a = \texttt{F}, b = \texttt{T}, c = \texttt{F}\}$.

Finally, in Fig. 4.6 we define wrapped versions of the primitive operations used in accumulation. These wrapper functions first check to see whether a symbolic reference for the given IL term exists already in the accumulation store, and if so, returns it without changing the store. Otherwise, it invokes the corresponding primitive operation to generate and return the new symbolic reference and updated store.

4.5.2 Accumulation

The accumulation phase is formally specified in Fig. 4.7 as a relation of the form $(\Delta, v) \mapsto (\Delta', v')$. Accumulation replaces plain subterms of the formula with references to symbolic values, wherever possible. The replacement of subterms by symbolic references is achieved by the first four rules in the figure. In the A-REF rule, a variable reference is replaced by a symbolic reference by invoking the wrapped version of

$$\frac{\underline{\texttt{Var}}(\Delta, r) = (\Delta', s)}{(\Delta, r) \mapsto (\Delta', s)} \text{ A-REF}$$

$$\frac{(\Delta, v) \mapsto (\Delta', s) \qquad \underline{\texttt{Not}}(\Delta', s) = (\Delta'', s')}{(\Delta, \neg v) \mapsto (\Delta'', s')} \text{ A-NOT-S}$$

$$\frac{\begin{array}{c}(\Delta, v_1) \mapsto (\Delta_1, s_1) \\ (\Delta_1, v_2) \mapsto (\Delta_2, s_2) \qquad \underline{\texttt{And}}(\Delta_2, s_1, s_2) = (\Delta_3, s_3)\end{array}}{(\Delta, v_1 \wedge v_2) \mapsto (\Delta_3, s_3)} \text{ A-AND-S}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, s_1) \qquad (\Delta_1, v_2) \mapsto (\Delta_2, s_2) \qquad \underline{\texttt{Or}}(\Delta_2, s_1, s_2) = (\Delta_3, s_3)}{(\Delta, v_1 \vee v_2) \mapsto (\Delta_3, s_3)} \text{ A-OR-S}$$

$$\frac{}{(\Delta, D\langle e_1, e_2\rangle) \mapsto (\Delta, D\langle e_1, e_2\rangle)} \text{ A-CHC} \qquad \frac{(\Delta, v) \mapsto (\Delta', v')}{(\Delta, \neg v) \mapsto (\Delta', \neg v')} \text{ A-NOT-V}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, v_1') \qquad (\Delta_1, v_2) \mapsto (\Delta_2, v_2')}{(\Delta, v_1 \wedge v_2) \mapsto (\Delta_2, v_1' \wedge v_2')} \text{ A-AND-V}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, v_1') \qquad (\Delta_1, v_2) \mapsto (\Delta_2, v_2')}{(\Delta, v_1 \vee v_2) \mapsto (\Delta_2, v_1' \vee v_2')} \text{ A-OR-V}$$

Figure 4.7: Accumulation inference rules.

the `Var` primitive, which returns the corresponding symbolic reference or generates a new one, if needed. The A-NOT-S, A-AND-S, and A-OR-S rules all similarly replace an IL term by a symbolic reference by invoking the corresponding wrapped primitive operation. These rules all require that their subterms completely reduce to symbolic references, as illustrated by the premise $(\Delta, v) \mapsto (\Delta', s)$ in the A-NOT-S rule, otherwise the primitive operation cannot be invoked.

However, not all subterms can be completely reduced to symbolic references. In particular, variational subterms—subterms that contain one or more choices within them—cannot be accumulated to a symbolic reference. The A-CHC rule prevents accumulation under a choice. The A-NOT-V, A-AND-V, and A-OR-V rules are congruence rules that recursively apply accumulation to subterms. Although not explicitly stated in the premises, it is assumed that these A-*-V rules apply only if the corresponding A-*-S rule does not apply, that is, when at least one of the subterms does not reduce completely to a symbolic reference.

We have omitted rules for processing the constant values `T` and `F`. These rules correspond closely to the A-REF rule, but use a predefined variable reference for the true and false constants.

To illustrate the semantics of accumulation, consider the plain formula $g = a \vee (a \wedge b)$ with an initial accumulation store $\Delta = \varnothing$. The A-OR-S rule matches the root $\vee$ connective with $v_1 = a$ and $v_2 = a \wedge b$. Since v_1 is a reference, the A-REF rule applies, generating a new symbolic reference s_a and returning the store $\Delta_1 = \{(a, s_a)\}$. Processing v_2 requires an application of the A-AND-S rule with $v'_1 = a$ and $v'_2 = b$, both of which require another application of the A-REF rule. For v'_1, the variable a is found in the store returning s_a, while for v'_2, a new symbolic reference s_b is generated and added to the resulting store $\Delta_2 = \{(a, s_a), (b, s_b)\}$. Since both the left and right sides of v_2 reduce to a symbolic reference, the `And` primitive is invoked, yielding a new symbolic reference s_{ab} and the store $\Delta_3 = \{(a, s_a), (b, s_b), (a \wedge b, s_{ab})\}$. Finally, since both the left and right sides of the original formula g reduce to symbolic references, the `Or` primitive is invoked yielding the final symbolic reference s_g and the final accumulation

$$\frac{(\Delta, v) \mapsto (\Delta', v') \qquad (\Gamma, \Delta', v') \rightarrowtail (\Gamma', \Delta'', v'')}{(\Gamma, \Delta, v) \rightarrowtail (\Gamma', \Delta'', v'')} \text{ E-ACC} \qquad \frac{\texttt{Assert}(\Gamma, \Delta, s) = \Gamma'}{(\Gamma, \Delta, s) \rightarrowtail (\Gamma', \Delta, \bullet)} \text{ E-SYM}$$

$$\frac{}{(\Gamma, \Delta, D\langle e_1, e_2\rangle) \rightarrowtail (\Gamma, \Delta, D\langle e_1, e_2\rangle)} \text{ E-CHC} \qquad \frac{}{(\Gamma, \Delta, v_1 \vee v_2) \rightarrowtail (\Gamma, \Delta, v_1 \vee v_2)} \text{ E-OR}$$

$$\frac{(\Gamma, \Delta, v_1) \rightarrowtail (\Gamma_1, \Delta_1, \bullet) \qquad (\Gamma_1, \Delta_1, v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_2')}{(\Gamma, \Delta, v_1 \wedge v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_2')} \text{ E-AND-L}$$

$$\frac{(\Gamma, \Delta, v_1) \rightarrowtail (\Gamma_1, \Delta_1, v_1') \qquad (\Gamma_1, \Delta_1, v_2) \rightarrowtail (\Gamma_2, \Delta_2, \bullet)}{(\Gamma, \Delta, v_1 \wedge v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_1')} \text{ E-AND-R}$$

$$\frac{(\Gamma, \Delta, v_1) \rightarrowtail (\Gamma_1, \Delta_1, v_1') \qquad (\Gamma_1, \Delta_1, v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_2')}{(\Gamma, \Delta, v_1 \wedge v_2) \rightarrowtail (\Gamma_2, \Delta_2, v_1' \wedge v_2')} \text{ E-AND}$$

Figure 4.8: Evaluation inference rules.

store $\Delta_4 = \{(a, s_a), (b, s_b), (s_a \wedge s_b, s_{ab})(s_a \vee s_{ab}, s_g)\}$.

When a formula contains choices, all of the plain subterms surrounding the choices are accumulated to symbolic references, but choices remain in place and their alternatives are not accumulated. For example, consider the variational formula $g' = (a \vee (a \wedge b)) \vee D\langle a, a \wedge b\rangle \wedge (a \vee (a \wedge b))$ which contains two instances of g as subterms. The formula g' accumulates to the variational core $s_g \vee D\langle a, a \wedge b\rangle \wedge s_g$ with the same final store Δ_4 produced when accumulating g alone. Note that the each instance of g in g' was reduced to the same symbolic reference s_g and the alternatives of the choice were not reduced.

4.5.3 Evaluation

The evaluation phase is formally specified in Fig. 4.8 as a relation of the form $(\Gamma, \Delta, v) \rightarrowtail (\Gamma', \Delta', v')$, where an evaluation store Γ represents the base solver's state. The E-ACC and E-SYM rules are the heart of evaluation: the E-ACC rule enables accumulating subterms during evaluation, while the E-SYM rule sends a fully accumulated subterm to the base solver. Evaluation cannot occur under choices or un-accumulated disjunctions (i.e. disjunctions that contain choices), as seen in the E-CHC and E-OR rules, but can occur under un-accumulated conjunctions, as reflected by the three E-AND$*$ rules. This will be explained in more detail below.

When a subterm is sent to the base solver by E-SYM, it is replaced by the unit value $\bullet$ and the evaluation store Γ is updated accordingly. Conceptually, the evaluation store represents the internal state of the underlying solver (e.g. z3's internal state), but we model it formally as the set of assertions that have been sent to the solver. For example, given the accumulation store $\Delta = \{(a, s_a), (b, s_b), (s_a \wedge s_b, s_{ab})\}$, the assertion $\texttt{Assert}(\{\}, \Delta, s_a)$ yields $\{s_a\}$ and subsequent assertions add more elements to this set, for example, $\texttt{Assert}(\{s_a\}, \Delta, s_{ab}) = \{s_a, s_{ab}\}$. The assertions sent to a SAT solver are implicitly conjuncted together, which is why partially accumulated conjunctions may still be evaluated, but partially accumulated disjunctions may not. Such disjunctions are instead handled during choice removal using back-tracking.

The three E-AND$*$ rules propagate accumulation over conjunctions. In all three rules, the subterms are evaluated left-to-right, propagating the resulting stores accordingly. The E-AND-L rule states that if the left side of a conjunction can be fully evaluated to $\bullet$, then the expression can be evaluated to the result of the right side; likewise, E-AND-R states that if the right side fully evaluates, the result of evaluating the expression is the result of the left side. If neither side fully evaluates to $\bullet$ (i.e. because both contain choices or disjunctions), then E-AND applies, which leaves the conjunction in place (with evaluated subterms) to be handled during choice removal.

Consider evaluating the formula $g = (a \vee b) \wedge D\langle a, c\rangle$ with initially empty stores. We start by applying accumulation using the E-ACC rule, yielding the intermediate term $g' = s_{ab} \wedge D\langle a, c\rangle$ with the accumulation store $\Delta = \{(a, s_a), (b, s_b), (s_a \vee s_b, s_{ab})\}$. We then apply E-AND-L to g', which sends the left subterm s_{ab} to the base solver via the E-SYM rule, and the right side will be unevaluated via the E-CHC rule. Ultimately, evaluation yields the expression $D\langle a, c\rangle$ with accumulation store Δ and evaluation store $\{s_{ab}\}$.

4.5.4 Choice removal

The main driver of variational solving is the choice removal phase, which is formally specified in Fig. 4.9 as a relation of the form $(C, \Gamma, \Delta, M, z, v) \Downarrow M'$. The main role of choice removal is to relate an IL term v to a variational model M'. However, to do this requires several pieces of context including a configuration C, an evaluation store Γ, an accumulation store Δ, an initial variational model M, and an evaluation context z. The two stores have been explained earlier in this chapter, and variational models are explained at the end of Section 4.1. We explain configurations and evaluation contexts in the context of the relevant rules below.

$$\frac{(\Gamma,\Delta,v) \rightarrowtail (\Gamma',\Delta',\bullet) \qquad \texttt{Combine}(M,\texttt{GetModel}(\Delta,\Gamma)) = M'}{(C,\Gamma,\Delta,M,\top,v) \Downarrow M'}\ \textsc{C-Eval}$$

$$\frac{(D,\textsf{true}) \in C \qquad (C,\Gamma,\Delta,M,z,e_1) \Downarrow M'}{(C,\Gamma,\Delta,M,z,D\langle e_1,e_2\rangle) \Downarrow M'}\ \textsc{C-Chc-T}$$

$$\frac{(D,\textsf{false}) \in C \qquad (C,\Gamma,\Delta,M,z,e_2) \Downarrow M'}{(C,\Gamma,\Delta,M,z,D\langle e_1,e_2\rangle) \Downarrow M'}\ \textsc{C-Chc-F}$$

$$\frac{\begin{array}{c} D \notin dom(C) \qquad (C \cup (D,\textsf{true}),\Gamma,\Delta,M,z,e_1) \Downarrow M_1 \\ (C \cup (D,\textsf{false}),\Gamma,\Delta,M',z,e_2) \Downarrow M_2 \end{array}}{(C,\Gamma,\Delta,M,z,D\langle e_1,e_2\rangle) \Downarrow M_2}\ \textsc{C-Chc}$$

$$\frac{(C,\Gamma,\Delta,M,\neg\cdot :: z,v) \Downarrow M'}{(C,\Gamma,\Delta,M,z,\neg v) \Downarrow M'}\ \textsc{C-Not}$$

$$\frac{(\Delta,\neg s) \mapsto (\Delta',s') \qquad (C,\Gamma,\Delta,M,z,s') \Downarrow M'}{(C,\Gamma,\Delta,M,\neg\cdot :: z,s) \Downarrow M'}\ \textsc{C-Not-In}$$

$$\frac{(C,\Gamma,\Delta,M,\cdot \wedge v_2 :: z,v_2) \Downarrow M'}{(C,\Gamma,\Delta,M,z,v_1 \wedge v_2) \Downarrow M'}\ \textsc{C-And}$$

$$\frac{(C,\Gamma,\Delta,M,s \wedge \cdot :: z,v) \Downarrow M'}{(C,\Gamma,\Delta,M,\cdot \wedge v :: z,s) \Downarrow M'}\ \textsc{C-And-InL}$$

$$\frac{(\Delta,s_1 \wedge s_2) \mapsto (\Delta',s_3) \qquad (C,\Gamma,\Delta,M,z,s_3) \Downarrow M'}{(C,\Gamma,\Delta,M,s_1 \wedge \cdot :: z,s_2) \Downarrow M'}\ \textsc{C-And-InR}$$

$$\frac{(C,\Gamma,\Delta,M,\cdot \vee v_2 :: z,v_2) \Downarrow M'}{(C,\Gamma,\Delta,M,z,v_1 \vee v_2) \Downarrow M'}\ \textsc{C-Or}$$

$$\frac{(C,\Gamma,\Delta,M,s \vee \cdot :: z,v) \Downarrow M'}{(C,\Gamma,\Delta,M,\cdot \vee v :: z,s) \Downarrow M'}\ \textsc{C-Or-InL}$$

$$\frac{(\Delta,s_1 \vee s_2) \mapsto (\Delta',s_3) \qquad (C,\Gamma,\Delta,M,z,s_3) \Downarrow M'}{(C,\Gamma,\Delta,M,s_1 \vee \cdot :: z,s_2) \Downarrow M'}\ \textsc{C-Or-InR}$$

Figure 4.9: Choice removal inference rules

The C-EVAL rule states that v fully evaluates to $\bullet$, then we can get the current model from the base solver using the `GetModel` primitive and update our variational model. We use the operation `Combine` to perform the variational model update operation described in Section 4.1. The rest of the choice removal rules are structured so that C-EVAL will be invoked once for every variant of the variational core so that the final output will be a variational model that encodes the solutions to every variant of the original formula.

The next three rules concern choices and are the heart of choice removal. These rules make use of a *configuration* C, which maps dimensions to Boolean values (encoded as a set of pairs). The configuration tracks which dimensions have been selected and how to ensure that all choices in the same dimension are synchronized. Whenever a choice $D\langle e_1, e_2\rangle$ is encountered during choice removal, we check C to determine what to do. In C-CHC-T, if $(D, \mathsf{true}) \in C$, then the first alternative of the dimension has already been selected, so choice removal proceeds on e_1. Similarly, in C-CHC-F, if $(D, \mathsf{false}) \in C$, the right alternative has been selected, so choice removal proceeds on e_2. In C-CHC, if $D \notin dom(C)$, then the dimension has not yet been selected, so we recursively apply choice removal to both e_1 and e_2, updating C accordingly in each case. Observe that we use the same accumulation store, evaluation store, and evaluation context for each alternative. This simulates a backtracking point in the solver, where we first solve e_1, then reset the state of the solver to the point where we encountered the choice and solve e_2. Only the variational model, which is threaded through the solution of both e_1 and e_2, is maintained to accumulate the results of solving each alternative.

The final eight rules apply choice removal to the logical operations. These rules make heavy use of an *evaluation context* z that keeps track of where we are in a partially evaluated IL term during choice removal. Evaluation contexts are defined as a zipper data structure [67] over IL terms, given by the following grammar.

$$z \;::=\; \top \mid \neg\cdot :: z \mid \;\cdot \wedge v :: z \mid s \wedge \cdot :: z \mid \;\cdot \vee v :: z \mid s \vee \cdot :: z$$

An evaluation context z is like a breadcrumb trail that enables focusing on a subterm within a partially evaluated IL term while also keeping track of work left to do. The empty context $\top$ indicates the root of the term. The other cases in the grammar prepend a "crumb" to the trail. The crumb $\cdot\neg$ focuses on the subterm within a negation, $\cdot \wedge v$ focuses on the left subterm within a conjunction whose right subterm is v, and $v \wedge \cdot$ focuses on the right subterm of a conjunction whose left subterm has already been reduced to s. The cases for disjunction are similar to conjunction.

As an example, consider the IL term $\neg(a \vee b) \wedge c$. When evaluation is focused on a, the evaluation context will be $\cdot \vee b :: \neg\cdot :: \cdot \wedge c :: \top$, which states that a exists as the left child of a disjunction whose right child is b, which is inside a negation, which is the left child of a conjunction whose right child is c. The b and c terms captured in the context are subterms of the original term that must still be evaluated. During choice removal, IL terms are evaluated according to a left-to-right, post-order traversal; as IL subterms are evaluated they are replaced by symbolic references via accumulation. When evaluation is focused on b, the context will be $s_a \vee \cdot :: \neg\cdot :: \cdot \wedge c :: \top$, where s_a is the symbolic reference produced by accumulating the variable a. When evaluation is eventually focused on c, the evaluation context will be simply $s_{ab} \wedge \cdot :: \top$ since the entire subtree $\neg(a \vee b)$ on the left side of the conjunction will have been accumulated to the symbolic reference s_{ab}.

The C-NOT, C-AND, and C-OR rules define what to do when encountering a logical operation for the first time. In C-NOT, we focus on the subterm of the negation, while in C-AND and C-OR, we focus on the left child while saving the right child in the context. The C-AND-INL and C-OR-INL rules define what to do when *finished* processing the left child of the corresponding operation. A fully processed child have been accumulated to a symbolic reference s. At this point, we move the s into the evaluation context and shift focus to the previously saved right child of the logical operation. Finally, the C-NOT-IN, C-AND-INR, and C-OR-INR rules define what to do when finished processing all children of a logical operation. At this point, all children will have been reduced to symbolic references so we can accumulate the entire subterm and apply choice removal to the result. For example, in C-AND-INR, we have just finished processing the right child to s_2 and we previously reduced the left child to s_1, so we now accumulate $s_1 \vee s_2$ to s_3 and proceed from there.

Evaluation contexts support a simple recursive approach to solving variational formulas by adding to the context as we move down the term and removing from the context as we move back up. The extra effort over a more direct recursive strategy is necessary to support the backtracking pattern implemented by the C-CHC rule. Whenever we encounter a choice in a new dimension, we can simply split the state of the solver to explore each alternative. Without evaluation contexts, this would be extremely difficult since choices may be deeply and arbitrarily nested within a variational formula. We would have to somehow remember all of the locations in the term that we must backtrack to later and the state of the solver at each of those locations.

4.6 Variation Preservation

$$
\begin{tikzcd}
f \arrow[r, Rightarrow, "\{f' \mid \exists C.\ \llbracket f \rrbracket_C = f',\ f' \in C_2\}"] \arrow[d, "VSAT(f)"'] & \{f'\} \arrow[d, Rightarrow, "SAT(f')"] \\
M_v^f \arrow[r, Rightarrow, "\{M_v^f(C) \mid \exists C.\ \llbracket f \rrbracket_C = f',\ f' \in C_2\}"] & \{\mathrm{SAT}\}
\end{tikzcd}
$$

Figure 4.10: Commuting diagram showing variation preservation for a VPL formula f.

We have formalized variational satisfiability solving. In this section we prove that our method is variation preserving. Variation preservation is a key property for variational or variation-aware systems. A system is variation preserving if and only if processing the variational artifact produces a variational result which can be used to recover semantically identical plain results for each variant. For this work, variation preservation means that the variational solver should find the same results as a plain solver that is solving every variant.

The property of variation preservation is presented in Fig. 4.10. The commuting diagram states that for any VPL formula f, if we configure f to find all possible plain variants, and then run a plain SAT solver on each variant, we should find the same number of SAT/UNSAT results we would find if we ran the variational solver on f, received a variational model, and then substituted every total configuration into the model. Note that the models may still differ, because a formula may have more than one satisfying model.

To show variation preservation we must show two properties: First, if a variant exists then it is found by the variational solver. Second, that if the variant is found by the variational solver it is communicated to the base solver without *any loss* and without *anything extra*. In other words if a variant is found then it is communicated to the base solver without losing any terms and without gaining any terms during communication.

We begin by showing that accumulation does not lose any information. We use mathematical sets as a formal representation of symbolic values.

Lemma 4.6.1 (Accumulation progress)**.** *Given a VPL formula f, accumulation visits all sub-terms $t \in f$.*

Proof. By construction, for any formula f all terms in t are reachable by one or repeated applications of the congruence rules A-OR-V, A-AND-V and A-NOT-V. $\square$

Lemma 4.6.2 (Accumulation preserves choices)**.** *For any VPL formula f, and any store Δ, accumulation on f with Δ preserves choices.*

Proof. By construction, for any formula f and any store, Δ, accumulation can only process choices with the A-CHC rule which preserves choices. $\square$

Lemma 4.6.3 (Accumulation preserves plain terms)**.** *For any VPL formula f, and any store Δ, if a term $t \in f$ and t is not a choice, then $\exists s \Delta' \, \forall f'. \; t \in s \in f'$ where s is a symbolic value, and f' is the accumulated result of f.*

Proof. By structural induction on t. The base case is t is a terminal. If t is terminal than a computation rule such as A-REF will convert it to s and store it in Δ yielding

Δ'. If t is not a terminal then it is a logical connective with one or two children and a root operator. There are three cases for the operator: $\neg$, $\vee$, and $\wedge$, each of which will have a similar proof. We show the remainder of the proof assuming the operator is $\vee$ because it is a more general case than $\neg$ and identical to $\wedge$ in the proof. The induction hypothesis is given the immediate sub-terms v_1 and v_2 of t, accumulation over the sub-terms preserves plain terms. There are two rules to consider: A-OR-S or A-OR-V. By assumption t is plain, and therefore so are v_1 and v_2. Thus by A-OR-S and the induction hypotheses we know that $(\Delta, v_1) \mapsto (\Delta_1, s_1)$ and $(\Delta_1, v_2) \mapsto (\Delta_2, s_2)$ are plain term preserving. Then application of `Or` we have s and Δ' under the assumption the solver is term preserving. □

Theorem 4.6.4 (Accumulation is term preserving)**.** *No terms are lost during accumulation of a formula f.*

Proof. By Lemma 4.6.1 all terms in a VPL formula are visited, by Lemma 4.6.3 all plain terms are preserved sure conversion to symbolic values, and by Lemma 4.6.2 all choices and thus variants are preserved. □

Similarly we must show that terms are not lost during evaluation. To not lose terms in evaluation means that any term reduced to a $\bullet$ is considered for future CHECK-SAT calls as long as the assertions are not removed from the assertion stack:

Theorem 4.6.5 ($\bullet$ values are shared)**.** *During evaluation, if a term is evaluated to $\bullet$ then the term is shared among variants in future assertion levels or the variant considered in the current assertion level.*

Proof. By definition if a term is *not* shared, then it is in some variants and not in others and therefore in a choice. By construction, evaluation can only produce $\bullet$ values from symbolic value, which can only be created by calling accumulation. There are two cases: either a symbolic value is returned by accumulation or an accumulated formula is returned. If a symbolic value is returned, then it can be evaluated to a $\bullet$ via E-SYM; placing it on the assertion stack of the base solver and therefore sharing it for all subsequent variants as it is deeper in the stack than future assertions. If an accumulated formula is returned, then there are three cases for the logical connective at the root of the formula: $\neg$, $\vee$ and $\wedge$. Only $\wedge$ allows evaluation to continue, in which case E-SYM and E-AND proceed until a connective other than $\wedge$ is at the root of the sub-term. By Theorem 4.6.4 accumulation preserves terms. Thus for each sub-term which is a symbolic value E-SYM applies and is therefore shared. For each sub-term which is not a symbolic value, the sub-term is either a choice or an accumulated formula which cannot be further reduced. $\square$

Lemma 4.6.6 (Evaluation progress)**.** *Given a VPL formula f, for any term $t \in f$ either evaluation processes t or calls accumulation on t.*

Proof. By structural induction on t. For every possible connective there is a case in the evaluation inference rules. From E-AND evaluation is propagated over $\wedge$, for $\neg$ and $\vee$ must switch to accumulation to process the term. $\square$

With these properties we can finally show that choice removal finds all variants, and communicates the variants to the base solver without losing or gaining terms which do not exist in the variant. For the proofs we do not assume an input vc.

Theorem 4.6.7 (Choice removal progress)**.** *Given a VPL formula f, choice removal processes all variants $v \in Variants(f)$.*

Proof. Given f, $\exists f'$ such that f' is the variational core of f. Then by structural induction on f', there are four cases: The base case is f' is $\bullet$ with evaluation context $\top$, and thus C-EVAL applies. The inductive cases have a connectives at the root of f'. If f' is a choice with evaluation context z then by structural induction on C either C-CHC, C-CHC-T or C-CHC-F applies. C-CHC is the important rule. If C-CHC applies then $D \notin Dom(C)$ and both variants are recursively processed with the same evaluation context and $\{D\} \cup Dom(C)$. Therefore, the evaluation context is shared between variants and no terms are lost. The C-CHC-T and C-CHC-F rules ensure variants do not cross communicate. Thus, if $D \in Dom(C)$ then either C-CHC-T or C-CHC-F applies and the current variant is recursively processed. If $\neg$, $\vee$ or $\wedge$ are at the root either C-AND, C-OR or C-NOT applies, forcing choice removal to recur into the left child. Since, variational cores are only composed of symbolic values and choices either the left child is a choice or a symbolic. If the left child is a choice then it is processed as described above. If the left child is a symbolic value then only C-AND-INL, C-OR-INL and C-NOT-IN apply, then by structural induction on z choice removal processes the right child. Once in the right child only C-AND-INR, C-OR-INR apply thereby folding symbolic values using accumulation. Thus, every choice will be reached, by manipulation of the configuration every variant is processed with C-CHC, and variants are never mixed due to C-CHC-T and C-CHC-F. $\square$

Theorem 4.6.8 (Choice removal preservation)**.** *Given a VPL formula f, choice removal does not lose terms during processing f and processed variants do not gain terms.*

Proof. The proof is nearly identical in structure to Theorem 4.6.7. By Theorem 4.6.4 and threading of Δ through choice removal terms are not lost. Terms are not gained in variants due to C-CHC-T and C-CHC-F as described in Theorem 4.6.7. $\square$

Now we can finally show variation preservation:

Theorem 4.6.9 (Variational satisfiability solving is variation preserving)**.** *Given a VPL formula f, there exists a set of total configurations CS such that* $\{r \mid M_v^f(C) = r, C \in CS, VSAT(f) = M_v^f\} = \{r \mid SAT(f') = r, C \in CS, [\![f]\!]_C = f'\}$.

Proof. We must show two directions: $\{r \mid M_v^f(C) = r, C \in CS, VSAT(f) = M_v^f\} \subseteq \{r \mid SAT(f') = r, C \in CS, [\![f]\!]_C = f'\}$ and $\{r \mid SAT(f') = r, C \in CS, [\![f]\!]_C = f'\} \subseteq \{r \mid M_v^f(C) = r, C \in CS, VSAT(f) = M_v^f\}$, we refer to these as *plain to variational* and *variational to plain*.

Subproof: plain to variational. Assume for contradiction there exists a variant f' of f such that $SAT(f') = r$ and $\nexists C.M_v^f(C) = r$. By Theorem 4.6.8, Theorem 4.6.4, and Theorem 4.6.5 terms are not lost nor gained when solving a variant using the variational solver, thus there must exist a configuration C' such that $M_v^f(C') = r$, hence a contradiction. $\blacksquare$

Subproof: variational to plain. Assume for contradiction there exists a variant f' determined by configuration C' such that $M_v^f(C') = r$ but $r \notin \{r' \mid SAT(v') = r', C \in CS, [\![f]\!]_C = v'\}$. Since f' is a variant of f determined by C', $C' \in CS$ and therefore $\exists r'.SAT(f') = r'$. We must show that $r' = r$. For $r' \neq r$ either terms were lost during $VSAT(f)$, or gained when f' was processed, but by Theorem 4.6.8, Theorem 4.6.4, and Theorem 4.6.5 this cannot be the case. Thus $r' = r$ and therefore $r \in \{r' \mid SAT(v') = r', C \in CS, [\![f]\!]_C = v'\}$ hence a contradiction. ■

We have shown both $\{r \mid M_v^f(C) = r, C \in CS, VSAT(f) = M_v^f\} \subseteq \{r \mid SAT(f') = r, C \in CS, [\![f]\!]_C = f'\}$ and $\{r \mid SAT(f') = r, C \in CS, [\![f]\!]_C = f'\} \subseteq \{r \mid M_v^f(C) = r, C \in CS, VSAT(f) = M_v^f\}$. Thus we have $\{r \mid M_v^f(C) = r, C \in CS, VSAT(f) = M_v^f\} = \{r \mid SAT(f') = r, C \in CS, [\![f]\!]_C = f'\}$. Therefore, regardless of the path in Fig. 4.10 the same variant will be considered by the base solver and therefore variational solving is variation preserving. □

Chapter 5: Variational Satisfiability-Modulo Theory Solving

We have covered the basics of variational satisfiability solving. In this chapter we generalize the variational solving procedure to variational SMT solving. SMT solvers generalize SAT solvers through the use of *background theories* that allow the solver to reason about values and constructs outside the Boolean domain. The SMTLIB2 standard defines seven such background theories: CORE (Boolean theory), ARRAYSEX, FIXEDSIZEBITVECTORS, FLOATINGPOINT, INTS, REALS, and REAL_INTS. In this chapter, we use integer arithmetic (INTS) as an example SMT extension for variational SMT solving. Extensions for other background theories are similar to the INTS extension with the exception of the array theory. The array theory presents unique challenges due to interactions with choices; we conclude the section by presenting the array extension thus recovering the most popular SMT background theories in the variational solver.

5.1 Variational propositional logic extensions and primitives

In order to construct a variational SMT solver we must first extend VPL to include non-Boolean values, we call the extended VPL VPL$^{\mathbb{Z}}$ since its values can range over integers. VPL included two kinds of relations: relations such as $\neg$ and $\vee$ which required accumulation in the presence of variation, and relations such as $\wedge$ which required no special handling. Unfortunately, in the presence of variation there are no relations such

$$
\begin{array}{rcll}
i & \in & \mathbb{Z} & \textit{Integers} \\
t_i & ::= & r_i \quad | \quad i & \textit{Integer variables and literals} \\[4pt]
ar & ::= & t_i & \textit{Terminal} \\
 & | & -\, ar & \textit{Arithmetic Negation} \\
 & | & ar \ - \ ar & \textit{Subtraction} \\
 & | & ar \ + \ ar & \textit{Addition} \\
 & | & ar \ * \ ar & \textit{Multiplication} \\
 & | & ar \ \div \ ar & \textit{Division} \\
 & | & D\langle ar, ar\rangle & \textit{Choice}
\end{array}
$$

Figure 5.1: Syntax of integer arithmetic extension.

$$
\begin{array}{rcll}
t & ::= & r \quad | \quad \mathtt{T} \quad | \quad \mathtt{F} & \textit{Variables and Boolean literals} \\[4pt]
\otimes & ::= & < \ | \ \leq \ | \ \geq \ | \ > \ | \ \equiv & \textit{Binary relations} \\[4pt]
f & ::= & t & \textit{Terminal} \\
 & | & \neg f & \textit{Boolean Negation} \\
 & | & f \vee f & \textit{Or} \\
 & | & f \wedge f & \textit{And} \\
 & | & ar \otimes ar & \textit{Integer comparisons} \\
 & | & D\langle f, f\rangle & \textit{Choice}
\end{array}
$$

Figure 5.2: Syntax of extended VPL (VPL$^{\mathbb{Z}}$).

$$
\begin{array}{rcll}
\texttt{Not} : (\Delta, s) & \to & (\Delta, s) & \textit{Negate a symbolic value} \\
\texttt{And} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Conjunction of symbolic values} \\
\texttt{Or} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Disjunction of symbolic values} \\
\texttt{Neg} : (\Delta, s) & \to & (\Delta, s) & \textit{Negate an arithmetic symbolic value} \\
\texttt{Add} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Add symbolic values} \\
\texttt{Sub} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Subtract symbolic values} \\
\texttt{Div} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Divide symbolic values} \\
\texttt{Mult} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Multiply symbolic values} \\
\texttt{Lt} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Less than over symbolic values} \\
\texttt{Lte} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Less than equals over symbolic values} \\
\texttt{Gt} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Greater than over symbolic values} \\
\texttt{Gte} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Greater than equals over symbolic values} \\
\texttt{Eqv} : (\Delta, s, s) & \to & (\Delta, s) & \textit{Arithmetic equivalence over symbolic values} \\
\texttt{Var} : (\Delta, r) & \to & (\Delta, s) & \textit{Create symbolic value based on a boolean variable} \\
\texttt{Var}_{\texttt{z}} : (\Delta, r_i) & \to & (\Delta, s) & \textit{Create symbolic value based on a arithmetic variable} \\
\texttt{Assert} : (\Gamma, \Delta, s) & \to & \Gamma & \textit{Assert a symbolic value to the solver} \\
\texttt{GetModel} : (\Gamma, \Delta) & \to & m & \textit{Get a model for the current solver state}
\end{array}
$$

Figure 5.3: Assumed base solver primitive operations for $\text{VPL}^{\mathbb{Z}}$

as $\wedge$ for the SMT theories. Thus we add support for each theory except arrays through accumulation. Our strategy to extend VPL to $\text{VPL}^{\mathbb{Z}}$ is to add the appropriate cases to the syntax of VPL, extend the intermediate language, add the requisite primitive operations, and then extend the inference rules of accumulation and choice removal.

The $\text{VPL}^{\mathbb{Z}}$ syntax is presented in Fig. 5.1. $\text{VPL}^{\mathbb{Z}}$ includes syntax of the integer arithmetic extension, which consists of integer variables, integer literals, a set of standard operators, and choices. The sets of Boolean and arithmetic variables are disjoint, thus an expression such as $(s < \mathit{10}) \wedge (s \vee p)$, where s occurs as both an integer and Boolean variable is disallowed. The syntax of the language prevents type errors and expressions that do not yield Boolean values. For example, $D\langle 1, 2\rangle \wedge p$ is syntactically invalid. Similarly, the language only allows arithmetic expressions as children of an inequality, for

$$\underline{\texttt{Var}_\texttt{z}}(\Delta, r_i) = \begin{cases} (\Delta, s) & (r_i, s) \in \Delta \\ \texttt{Var}_\texttt{z}(\Delta, r_i) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Neg}}(\Delta, s) = \begin{cases} (\Delta, s') & (-s, s') \in \Delta \\ \texttt{Neg}(\Delta, s) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Add}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 + s_2, s_3) \in \Delta \\ \texttt{Add}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Sub}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 - s_2, s_3) \in \Delta \\ \texttt{Sub}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Div}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 \div s_2, s_3) \in \Delta \\ \texttt{Div}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Mult}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 * s_2, s_3) \in \Delta \\ \texttt{Mult}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

(a) Wrapped arithmetic primitives.

$$\underline{\texttt{Lt}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 < s_2, s_3) \in \Delta \\ \texttt{Lt}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Lte}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 \leq s_2, s_3) \in \Delta \\ \texttt{Lte}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Gt}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 > s_2, s_3) \in \Delta \\ \texttt{Gt}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Gte}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 \geq s_2, s_3) \in \Delta \\ \texttt{Gte}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

$$\underline{\texttt{Eqv}}(\Delta, s_1, s_2) = \begin{cases} (\Delta, s_3) & (s_1 \equiv s_2, s_3) \in \Delta \\ \texttt{Eqv}(\Delta, s_1, s_2) & \textit{otherwise} \end{cases}$$

(b) Wrapped inequality primitives.

Figure 5.4: Wrapped SMT primitives.

$$
\begin{aligned}
\neg &::= \neg && \textit{Boolean negation} \\
\dagger &::= - && \textit{Negation} \\
\otimes &::= \wedge && \textit{Conjunction} \\
&\mid\quad \vee && \textit{Disjunction} \\
\bowtie &::= < && \textit{Less than} \\
&\mid\quad > && \textit{Greater than} \\
&\mid\quad \leq && \textit{Less than Equal} \\
&\mid\quad \geq && \textit{Greater than Equal} \\
&\mid\quad \equiv && \textit{Equivalency} \\
\oplus &::= + && \textit{Addition} \\
&\mid\quad - && \textit{Subtraction} \\
&\mid\quad * && \textit{Multiplication} \\
&\mid\quad \div && \textit{Division}
\end{aligned}
$$

Figure 5.5: Syntactic categories of primitive operations

example: $g = (A\langle 1, 2\rangle + j \geq 2) \vee a \wedge A\langle c, d\rangle$ is syntactically valid but $p \wedge (1+7+2+9)$ is not. Choices in the same dimension are synchronized across Boolean and arithmetic sub-expressions, for example, the expression $g = (A\langle 1, 2\rangle + j \geq 2) \vee (a \wedge A\langle c, d\rangle)$ represents two variants: $[\![g]\!]_{\{(A,\mathsf{true})\}} = (1+j \geq 2) \vee (a \wedge c)$ and $[\![g]\!]_{\{(A,\mathsf{false})\}} = (2+j \geq 2) \vee (a \wedge d)$.

Similarly to Chapter 4, we define the assumed primitive operations of the base solver in Fig. 5.3, and wrapped versions for new operators in Fig. 5.4a and Fig. 5.4b. The wrapped versions are defined identically as the wrapped primitives in Fig. 4.6 and serve the same purpose. From the perspective of the variational solver, operations such as addition, division, and subtraction only differ in the primitive operation emitted to the base solver. Thus, we define syntactic categories over like operations in Fig. 5.5. Notice that the categories correspond to the respective type of each operation. For example, the Boolean categories encapsulate operations which take two Boolean expressions and return a Boolean expression, similarly the inequality category encapsulate operators which take numeric expressions and return Boolean expressions. Further SMT extensions would directly copy this pattern, that is, defining a syntactic category of FIXEDSIZEBITVECTOR or REALS operators. Similarly, while we present only a single arithmetic unary function $-$, other arithmetic unary functions would be straightforward to add. For example, to include an absolute value operator abs, one would define the wrapped primitive, and add the operator to the appropriate syntactic category without requiring any modification to the inference rules or intermediate languages.

Just as VPL was extended, the intermediate language must be extended. First we must add cases for inequality operations, and second we must define an intermediate language for the arithmetic domain. Fig. 5.6 defines the intermediate arithmetic language $ar^{\mathbb{Z}}$, and the extended intermediate language $v^{\mathbb{Z}}$. The syntax of both intermediate languages follow directly from VPL$^{\mathbb{Z}}$ and should be unsurprising. The only important difference from IL is that $ar^{\mathbb{Z}}$ cannot express a $\bullet$ value. This is a purposeful design decision; recall that a $\bullet$ represents a term that has been sent to the base solver. Thus if $\bullet$ were in $ar^{\mathbb{Z}}$ then expressions such as $\bullet + 2$ would be expressible in $ar^{\mathbb{Z}}$, however because all arithmetic formula's require accumulation the only possible result of evaluation/accumulation on arithmetic expressions is either a choice or a symbolic term, not a $\bullet$. Hence, we syntactically avoid classes of bugs by omitting the $\bullet$ value in $ar^{\mathbb{Z}}$.

$$
\begin{aligned}
v^{\mathbb{Z}} \; &::= \; \bullet \mid t \mid r \mid s \mid \neg v^{\mathbb{Z}} \mid v^{\mathbb{Z}} \otimes v^{\mathbb{Z}} \mid ar^{\mathbb{Z}} \bowtie ar^{\mathbb{Z}} \mid D\langle e, e\rangle \\
ar^{\mathbb{Z}} \; &::= \; i \mid r_i \mid s \mid \dagger\, ar^{\mathbb{Z}} \mid ar^{\mathbb{Z}} \oplus ar^{\mathbb{Z}} \mid D\langle ar, ar\rangle
\end{aligned}
$$

Figure 5.6: Extended intermediate language syntax

5.2 Accumulation

The variational SMT version of accumulation is specified in Fig. 5.7 and is a generalized variational fold over the abstract syntax tree of $v^{\mathbb{Z}}$. Just as before, accumulation is split into congruence rules over the intermediate language, computation rules over symbolic values and computation rules for references and choices.

The variational SAT version of accumulation is a specialized form of this version of accumulation. The only semantic difference between operators is the code emitted to the base solver, hence we generalize the previous version by performing a lookup to retrieve the appropriate wrapped primitive. The primitive is indicated with an underline. For example, if $\otimes = \wedge$ then A-BOOL-S specializes to A-AND-S where $\underline{\otimes} = \underline{\texttt{And}}$, and thus the resulting call becomes $\underline{\texttt{And}}(\Delta_2, s_1, s_2)$. Hence, the rules A-AND-S, and A-OR-S are specialized forms of the general rule A-BOOL-S.

Similarly, we collapse the arithmetic and inequality computation rules to A-ARITH-S and A-INEQ-S. The semantics of each rule, besides the operator lookup, remains unchanged; the congruence rules recur into the abstract syntax tree to convert references to symbolic values, choices are skipped over due to A-CHC and A-CHC-I, and plain values are combined with the computation rules A-BOOL-S, A-ARITH-S, and A-INEQ-S. The only other substantial difference is two new computation rules to handle arithmetic choices and variables, A-CHC-I, and A-REF-I. Both serve the same function as their

$$\frac{\underline{\texttt{Var}}(\Delta, r) = (\Delta', s)}{(\Delta, r) \mapsto (\Delta', s)}\ \textsc{A-Ref} \qquad \frac{\underline{\texttt{Var}_{\texttt{z}}}(\Delta, r_i) = (\Delta', s)}{(\Delta, r_i) \mapsto (\Delta', s)}\ \textsc{A-Ref-I}$$

$$\frac{(\Delta, v^{\mathbb{Z}}) \mapsto (\Delta', s) \qquad \underline{\texttt{Not}}(\Delta', s) = (\Delta'', s')}{(\Delta, \neg v^{\mathbb{Z}}) \mapsto (\Delta'', s')}\ \textsc{A-Not-S}$$

$$\frac{(\Delta, ar^{\mathbb{Z}}) \mapsto (\Delta', s) \qquad \underline{\dagger}(\Delta', s) = (\Delta'', s')}{(\Delta, \dagger\ ar^{\mathbb{Z}}) \mapsto (\Delta'', s')}\ \textsc{A-Unary-S}$$

$$\frac{\begin{array}{c}(\Delta, v_1^{\mathbb{Z}}) \mapsto (\Delta_1, s_1)\\ (\Delta_1, v_2^{\mathbb{Z}}) \mapsto (\Delta_2, s_2) \qquad \underline{\otimes}(\Delta_2, s_1, s_2) = (\Delta_3, s_3)\end{array}}{(\Delta, v_1^{\mathbb{Z}}\ \otimes\ v_2^{\mathbb{Z}}) \mapsto (\Delta_3, s_3)}\ \textsc{A-Bool-S}$$

$$\frac{\begin{array}{c}(\Delta, ar_1^{\mathbb{Z}}) \mapsto (\Delta_1, s_1)\\ (\Delta_1, ar_2^{\mathbb{Z}}) \mapsto (\Delta_2, s_2) \qquad \underline{\oplus}(\Delta_2, s_1, s_2) = (\Delta_3, s_3)\end{array}}{(\Delta, ar_1^{\mathbb{Z}}\ \oplus\ ar_2^{\mathbb{Z}}) \mapsto (\Delta_3, s_3)}\ \textsc{A-Arith-S}$$

$$\frac{\begin{array}{c}(\Delta, ar_1^{\mathbb{Z}}) \mapsto (\Delta_1, s_1)\\ (\Delta_1, ar_2^{\mathbb{Z}}) \mapsto (\Delta_2, s_2) \qquad \underline{\bowtie}(\Delta_2, s_1, s_2) = (\Delta_3, s_3)\end{array}}{(\Delta, ar_1^{\mathbb{Z}}\ \bowtie\ ar_2^{\mathbb{Z}}) \mapsto (\Delta_3, s_3)}\ \textsc{A-InEq-S}$$

$$\frac{}{(\Delta,\ D\langle e_1, e_2\rangle) \mapsto (\Delta, D\langle e_1, e_2\rangle)}\ \textsc{A-Chc}$$

$$\frac{}{(\Delta,\ D\langle ar_1, ar_2\rangle) \mapsto (\Delta, D\langle ar_1, ar_2\rangle)}\ \textsc{A-Chc-I}$$

$$\frac{(\Delta, v^{\mathbb{Z}}) \mapsto (\Delta', v^{\mathbb{Z}\prime})}{(\Delta, \neg v^{\mathbb{Z}}) \mapsto (\Delta', \neg v^{\mathbb{Z}\prime})}\ \textsc{A-Not-V} \qquad \frac{(\Delta, v^{\mathbb{Z}}) \mapsto (\Delta', v^{\mathbb{Z}\prime})}{(\Delta, \dagger v^{\mathbb{Z}}) \mapsto (\Delta', \dagger v^{\mathbb{Z}\prime})}\ \textsc{A-Unary-V}$$

$$\frac{(\Delta, {v^{\mathbb{Z}}}_1) \mapsto (\Delta_1, v_1^{\mathbb{Z}\prime}) \qquad (\Delta_1, {v^{\mathbb{Z}}}_2) \mapsto (\Delta_2, v_2^{\mathbb{Z}\prime})}{(\Delta, {v^{\mathbb{Z}}}_1 \otimes {v^{\mathbb{Z}}}_2) \mapsto (\Delta_2, v_1^{\mathbb{Z}\prime} \otimes v_2^{\mathbb{Z}\prime})}\ \textsc{A-Bool-V}$$

$$\frac{(\Delta, ar_1^{\mathbb{Z}}) \mapsto (\Delta_1, ar_1^{\mathbb{Z}\prime}) \qquad (\Delta_1, ar_2^{\mathbb{Z}}) \mapsto (\Delta_2, ar_2^{\mathbb{Z}\prime})}{(\Delta, ar_1^{\mathbb{Z}} \oplus ar_2^{\mathbb{Z}}) \mapsto (\Delta_2, ar_1^{\mathbb{Z}\prime} \oplus ar_2^{\mathbb{Z}\prime})}\ \textsc{A-Arith-V}$$

$$\frac{(\Delta, ar_1^{\mathbb{Z}}) \mapsto (\Delta_1, ar_1^{\mathbb{Z}\prime}) \qquad (\Delta_1, ar_2^{\mathbb{Z}}) \mapsto (\Delta_2, ar_2^{\mathbb{Z}\prime})}{(\Delta, ar_1^{\mathbb{Z}} \bowtie ar_2^{\mathbb{Z}}) \mapsto (\Delta_2, ar_1^{\mathbb{Z}\prime} \bowtie ar_2^{\mathbb{Z}\prime})}\ \textsc{A-InEq-V}$$

Figure 5.7: Accumulation inference rules

Boolean counterparts A-CHC and A-REF.

In this form it should be plain to see the recipe to further extend accumulation to another background theory. One would add a new computation rules for the new kinds of references and choices, a new computation rule for symbolic references in the theory, and a new congruence rule over the new abstract syntax trees. Extending accumulation with new operators is similarly trivial. Recall the modulus example, to extend accumulation with a modulus operator, assuming the wrapped primitive has been defined, we would only need to add the operator to $\oplus$ syntactic category and create a case such that $mod \in \oplus$ succeeds.

5.3 Evaluation

Evaluation's purpose is to assert symbolic terms in the base solver if it is safe to do so. Thus, the extensions to evaluation are minimal as accumulation is performing the majority of the work in creating the symbolic terms.

Variational SMT evaluation is defined in Fig. 5.8. The only change is the addition of E-INEQ corresponding to the addition of inequalities to VPL. Just as E-OR skips over un-accumulated disjunctions, E-INEQ skips over un-accumulated inequalities since evaluating inside an inequality is unsound in the base solver. Since evaluation calls E-ACC to accumulate relations if E-AND-L, E-AND-R, and E-AND don't apply, variational SMT evaluation simply relies on accumulation to progress. The special cases for conjunctions are maintained in order to sequence evaluation from left to right to take advantage of the behavior of the assertion stack and to propagate accumulation across conjunctions, just

$$\frac{(\Delta, v^{\mathbb{Z}}) \mapsto (\Delta', v^{\mathbb{Z}\prime}) \qquad (\Gamma, \Delta', v^{\mathbb{Z}\prime}) \rightarrowtail (\Gamma', \Delta'', v^{\mathbb{Z}\prime\prime})}{(\Gamma, \Delta, v^{\mathbb{Z}}) \rightarrowtail (\Gamma', \Delta'', v^{\mathbb{Z}\prime\prime})} \text{ E-ACC}$$

$$\frac{\texttt{Assert}(\Gamma, \Delta, s) = \Gamma'}{(\Gamma, \Delta, s) \rightarrowtail (\Gamma', \Delta, \bullet)} \text{ E-SYM} \qquad \frac{}{(\Gamma, \Delta, D\langle e_1, e_2\rangle) \rightarrowtail (\Gamma, \Delta, D\langle e_1, e_2\rangle)} \text{ E-CHC}$$

$$\frac{}{(\Gamma, \Delta, v_1^{\mathbb{Z}} \vee v_2^{\mathbb{Z}}) \rightarrowtail (\Gamma, \Delta, v_1^{\mathbb{Z}} \vee v_2^{\mathbb{Z}})} \text{ E-OR}$$

$$\frac{}{(\Gamma, \Delta, v_1^{\mathbb{Z}} \bowtie v_2^{\mathbb{Z}}) \rightarrowtail (\Gamma, \Delta, v_1^{\mathbb{Z}} \bowtie v_2^{\mathbb{Z}})} \text{ E-INEQ}$$

$$\frac{(\Gamma, \Delta, v_1^{\mathbb{Z}}) \rightarrowtail (\Gamma_1, \Delta_1, \bullet) \qquad (\Gamma_1, \Delta_1, v_2^{\mathbb{Z}}) \rightarrowtail (\Gamma_2, \Delta_2, v_2^{\mathbb{Z}\prime})}{(\Gamma, \Delta, v_1^{\mathbb{Z}} \wedge v_2^{\mathbb{Z}}) \rightarrowtail (\Gamma_2, \Delta_2, v_2^{\mathbb{Z}\prime})} \text{ E-AND-L}$$

$$\frac{(\Gamma, \Delta, v_1^{\mathbb{Z}}) \rightarrowtail (\Gamma_1, \Delta_1, v_1^{\mathbb{Z}\prime}) \qquad (\Gamma_1, \Delta_1, v_2^{\mathbb{Z}}) \rightarrowtail (\Gamma_2, \Delta_2, \bullet)}{(\Gamma, \Delta, v_1^{\mathbb{Z}} \wedge v_2^{\mathbb{Z}}) \rightarrowtail (\Gamma_2, \Delta_2, v_1^{\mathbb{Z}\prime})} \text{ E-AND-R}$$

$$\frac{(\Gamma, \Delta, v_1^{\mathbb{Z}}) \rightarrowtail (\Gamma_1, \Delta_1, v_1^{\mathbb{Z}\prime}) \qquad (\Gamma_1, \Delta_1, v_1^{\mathbb{Z}}) \rightarrowtail (\Gamma_2, \Delta_2, v_2^{\mathbb{Z}\prime})}{(\Gamma, \Delta, v_1^{\mathbb{Z}} \wedge v_2^{\mathbb{Z}}) \rightarrowtail (\Gamma_2, \Delta_2, v_1^{\mathbb{Z}\prime} \wedge v_2^{\mathbb{Z}\prime})} \text{ E-AND}$$

Figure 5.8: Evaluation inference rules

$$
\begin{array}{rcl}
z^{\mathbb{Z}} & ::= & \top \\
& | & \neg \cdot :: z^{\mathbb{Z}} \\
& | & \dagger \cdot :: z^{\mathbb{Z}} \\
& | & \cdot \otimes v^{\mathbb{Z}} :: z^{\mathbb{Z}} \\
& | & s \otimes \cdot :: z^{\mathbb{Z}} \\
& | & \cdot \oplus v^{\mathbb{Z}} :: z^{\mathbb{Z}} \\
& | & s \oplus \cdot :: z^{\mathbb{Z}} \\
& | & \cdot \bowtie v^{\mathbb{Z}} :: z^{\mathbb{Z}} \\
& | & s \bowtie \cdot :: z^{\mathbb{Z}}
\end{array}
$$

Figure 5.9: Variational SMT zipper context

as in variational satisfiability evaluation.

5.4 Choice removal

With accumulation and evaluation complete we turn to choice removal. Our strategy is similar to accumulation; we generalize the zipper context over Boolean, arithmetic and inequality relations using the syntactic categories defined in Fig. 5.5. Conceptually, choice removal remains a variational left fold that builds the zipper context until a symbolic value is the focus. at which point rules of the form C-*-INL *switch* the fold to process the right child of the relations, and rules of the form C-*-INR call accumulation to reduce the relation over symbolic values. We formally specify generalized choice removal in Fig. 5.10. The heart of choice removal remains the same, the rules C-EVAL, C-CHC, C-CHC-T, and C-CHC-F are reproduced for the new zipper context $z^{\mathbb{Z}}$ but are semantically identical to the specialized versions. The remaining rules are generalized versions of the SAT rules to handle each syntactic category the variational solver can process.

$$\frac{(\Gamma, \Delta, v^{\mathbb{Z}}) \rightarrowtail (\Gamma', \Delta', \bullet) \qquad \texttt{Combine}(M, \texttt{GetModel}(\Delta, \Gamma)) = M'}{(C, \Gamma, \Delta, M, \top, v^{\mathbb{Z}}) \Downarrow M'} \text{ C-EVAL}$$

$$\frac{(D, \mathsf{true}) \in C \qquad (C, \Gamma, \Delta, M, z^{\mathbb{Z}}, v_1^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, z^{\mathbb{Z}}, D\langle v_1^{\mathbb{Z}}, v_2^{\mathbb{Z}}\rangle) \Downarrow M'} \text{ C-CHC-T}$$

$$\frac{(D, \mathsf{false}) \in C \qquad (C, \Gamma, \Delta, M, z^{\mathbb{Z}}, v_2^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, z^{\mathbb{Z}}, D\langle v_1^{\mathbb{Z}}, v_2^{\mathbb{Z}}\rangle) \Downarrow M'} \text{ C-CHC-F}$$

$$\frac{\begin{array}{c} D \notin dom(C) \qquad (C \cup (D, \mathsf{true}), \Gamma, \Delta, M, z^{\mathbb{Z}}, v_1^{\mathbb{Z}}) \Downarrow M_1 \\ (C \cup (D, \mathsf{false}), \Gamma, \Delta, M', z^{\mathbb{Z}}, v_2^{\mathbb{Z}}) \Downarrow M_2 \end{array}}{(C, \Gamma, \Delta, M, z^{\mathbb{Z}}, D\langle v_1^{\mathbb{Z}}, v_2^{\mathbb{Z}}\rangle \Downarrow M_2} \text{ C-CHC}$$

$$\frac{(C, \Gamma, \Delta, M, \neg \cdot :: z^{\mathbb{Z}}, v^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, z^{\mathbb{Z}}, \neg v^{\mathbb{Z}}) \Downarrow M'} \text{ C-NOT}$$

$$\frac{(\Delta, \neg s) \mapsto (\Delta', s') \qquad (C, \Gamma, \Delta, M, z^{\mathbb{Z}}, s') \Downarrow M'}{(C, \Gamma, \Delta, M, \neg \cdot :: z^{\mathbb{Z}}, s) \Downarrow M'} \text{ C-NOT-IN}$$

$$\frac{(C, \Gamma, \Delta, M, \cdot \otimes v_2^{\mathbb{Z}} :: z, v_1^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, z^{\mathbb{Z}}, v_1^{\mathbb{Z}} \otimes v_2^{\mathbb{Z}}) \Downarrow M'} \text{ C-BOOL}$$

$$\frac{(C, \Gamma, \Delta, M, s \otimes \cdot :: z^{\mathbb{Z}}, v^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, \cdot \otimes v^{\mathbb{Z}} :: z^{\mathbb{Z}}, s) \Downarrow M'} \text{ C-BOOL-INL}$$

$$\frac{(\Delta, s_1 \otimes s_2) \mapsto (\Delta', s_3) \qquad (C, \Gamma, \Delta, M, z^{\mathbb{Z}}, s_3) \Downarrow M'}{(C, \Gamma, \Delta, M, s_1 \otimes \cdot :: z^{\mathbb{Z}}, s_2) \Downarrow M'} \text{ C-BOOL-INR}$$

Figure 5.10: Variational SMT choice removal inference rules

$$\frac{(C, \Gamma, \Delta, M, \dagger \cdot :: z^{\mathbb{Z}}, v^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, z^{\mathbb{Z}}, \dagger v^{\mathbb{Z}}) \Downarrow M'} \text{ C-U}\textsc{nary}$$

$$\frac{(\Delta, \dagger s) \mapsto (\Delta', s') \qquad (C, \Gamma, \Delta, M, z^{\mathbb{Z}}, s') \Downarrow M'}{(C, \Gamma, \Delta, M, \dagger \cdot :: z^{\mathbb{Z}}, s) \Downarrow M'} \text{ C-U}\textsc{nary}\text{-I}\textsc{n}$$

$$\frac{(C, \Gamma, \Delta, M, \cdot \bowtie v_2^{\mathbb{Z}} :: z, v_1^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, z^{\mathbb{Z}}, v_1^{\mathbb{Z}} \bowtie v_2^{\mathbb{Z}}) \Downarrow M'} \text{ C-I}\textsc{n}\text{E}\textsc{q}$$

$$\frac{(C, \Gamma, \Delta, M, s \bowtie \cdot :: z^{\mathbb{Z}}, v^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, \cdot \bowtie v^{\mathbb{Z}} :: z^{\mathbb{Z}}, s) \Downarrow M'} \text{ C-I}\textsc{n}\text{E}\textsc{q}\text{-I}\textsc{n}\text{L}$$

$$\frac{(\Delta, s_1 \bowtie s_2) \mapsto (\Delta', s_3) \qquad (C, \Gamma, \Delta, M, z^{\mathbb{Z}}, s_3) \Downarrow M'}{(C, \Gamma, \Delta, M, s_1 \bowtie \cdot :: z^{\mathbb{Z}}, s_2) \Downarrow M'} \text{ C-I}\textsc{n}\text{E}\textsc{q}\text{-I}\textsc{n}\text{R}$$

$$\frac{(C, \Gamma, \Delta, M, \cdot \oplus v_2^{\mathbb{Z}} :: z, v_1^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, z^{\mathbb{Z}}, v_1^{\mathbb{Z}} \oplus v_2^{\mathbb{Z}}) \Downarrow M'} \text{ C-A}\textsc{rith}$$

$$\frac{(C, \Gamma, \Delta, M, s \oplus \cdot :: z^{\mathbb{Z}}, v^{\mathbb{Z}}) \Downarrow M'}{(C, \Gamma, \Delta, M, \cdot \oplus v^{\mathbb{Z}} :: z^{\mathbb{Z}}, s) \Downarrow M'} \text{ C-A}\textsc{rith}\text{-I}\textsc{n}\text{L}$$

$$\frac{(\Delta, s_1 \oplus s_2) \mapsto (\Delta', s_3) \qquad (C, \Gamma, \Delta, M, z^{\mathbb{Z}}, s_3) \Downarrow M'}{(C, \Gamma, \Delta, M, s_1 \oplus \cdot :: z^{\mathbb{Z}}, s_2) \Downarrow M'} \text{ C-A}\textsc{rith}\text{-I}\textsc{n}\text{R}$$

Figure 5.10: Variational SMT choice removal inference rules

For each syntactic category we define three kinds of rules which form a template to extend choice removal to new background theories: First, we have rules which determine what to do when encountering a binary or unary relation for the first time. As a design choice this is defined to proceed into the left child. For example C-NOT, C-BOOL, and C-INEQ initiate the left fold by storing the relation in the zipper and focusing the left child $v_1^{\mathbb{Z}}$. Second, we define rules which recur down the left child of the relations until a symbolic value results from accumulation. For example, C-INEQ-INL, C-BOOL-INL, and C-ARITH-INL all move the focused symbol value s to the zipper context allowing choice removal to proceed to the right children of the same relation. Lastly, we have computation rules which perform the fold on the relation by calling accumulation. For example, C-UNARY-IN, C-ARITH-INR, and C-INEQ-INR call accumulation to process the symbolic value and reduce the given relation. In effect, accumulation *encapsulates* the semantics of the relations, evaluation propagates accumulation and performs code generation in the base solver, and choice removal alters the configuration, maintains evaluation contexts, and removes choices introducing new plain terms to the formula.

5.5 Variational SMT models

We have thus far covered accumulation, evaluation, and choice removal. However, to support SMT theories, variational models must be abstract enough to handle values other than Booleans. Functionally, variational SMT models must satisfy several constraints: the variational SMT model must be more memory efficient than storing all models returned by the solver naively. The variational model must allow users to find satisfying values for a variant. The model must allow users to find all variants in which a variable has a particular value or range of values.

Furthermore, several useful properties of variational models should be maintained: The model is non-variational; the user should not need to understand the choice calculus to understand their results. The model produces results that can be fed into a plain SAT or SMT solver. The model can be built incrementally and without regard to the ordering of results. Variational SAT models guaranteed this last constraint by forming commutative monoid under $\vee$; a technique which we cannot replicate for variational SMT models.

To maintain these properties and satisfy the functional requirements, our strategy for variational SMT models is to create a mapping of variables to SMT expressions. A variable's type is syntactically ensured to not change as variable sets are disjoint. Thus variables are disallowed from changing type as the result of a choice. For any variable in the model, we assume the type returned by the base solver is correct, and store the satisfying value in a linked list constructed of *if-statements*. Specifically, we utilize the function $ite : \mathbb{B} \rightarrow T \rightarrow T$ from the SMTLIB2 standard to construct the list. All variables are initialized as undefined (Und) until a value is returned from the base solver for a variant. To ensure the correct value of a variable corresponds to the appropriate variant, we translate the configuration which determines the variant to a variation context, and place the appropriate value in the *then* branch.

Consider the following VPL$^{\mathbb{Z}}$ formula: $f = ((A\langle i, 13\rangle - c) < (b + 10)) \rightarrow B\langle a, c > i\rangle$. f contains two unique choices, A, B, and thus represents four variants. In this case, the expression is under-constrained and so each variant will be found satisfiable.

$$
\begin{array}{cc}
\begin{array}{c}
i \rightarrow \text{-1} \\
c \rightarrow 0 \\
\\
b \rightarrow 4 \\
C_{FF} = \{(A, \textsf{false}), (B, \textsf{false})\}
\end{array}
&
\begin{array}{c}
\\
c \rightarrow 0 \\
a \rightarrow \texttt{F} \\
b \rightarrow 0 \\
C_{FF} = \{(A, \textsf{false}), (B, \textsf{true})\}
\end{array}
\\
\\
\begin{array}{c}
i \rightarrow 0 \\
c \rightarrow 0 \\
\\
b \rightarrow \text{-10} \\
C_{FT} = \{(A, \textsf{true}), (B, \textsf{false})\}
\end{array}
&
\begin{array}{c}
i \rightarrow \text{-1} \\
c \rightarrow 0 \\
a \rightarrow \texttt{T} \\
b \rightarrow 3 \\
C_{TT} = \{(A, \textsf{true}), (B, \textsf{true})\}
\end{array}
\end{array}
$$

Figure 5.11: Possible plain models for variants of f.

Fig. 5.11 shows possible plain models for f with the corresponding variational SMT model presented in Fig. 5.12. We've added line breaks to emphasize the $then$ and $else$ branches of the ite SMTLIB2 primitive.

This formulation maintains the functional requirements and desirable properties of the variational SAT models. The variable $\_Sat$ is used to track the variants that were found satisfiable, just as in the variational SAT solver. In this case, all variants are satisfiable and thus we have four clauses over dimensions in disjunctive normal form. If a user has a configuration then they only need to perform substitution to determine the value of a variable under that configuration. For example, if the user were interested in the value of i in the $\{(A, \texttt{T}), (B, \texttt{T})\}$ variant they would substitute the configuration into the result for i and recover 2 from the first ite case. To find the variants at which a variable has a value, a user may employ an SMT solver, add the entry for i as a constraint, and query for a model. This specification of variational SMT models does not require knowledge of choice calculus or variation, it is still monoidal—although not

$$
\begin{array}{lll}
\_Sat & \rightarrow & (\neg A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge \neg B) \vee (A \wedge B) \\
i & \rightarrow & (ite\ (A \wedge B)\ \text{-1} \\
& & \quad (ite\ \ (A \wedge \neg B)\ \ 0 \\
& & \quad\quad (ite\ \ (\neg A \wedge \neg B)\ \ \text{-1}\ Und))) \\
c & \rightarrow & (ite\ (A \wedge B)\ \ 0 \\
& & \quad (ite\ \ (A \wedge \neg B)\ \ 0 \\
& & \quad\quad (ite\ \ (\neg A \wedge B)\ \ 0 \\
& & \quad\quad\quad (ite\ \ (\neg A \wedge \neg B)\ \ 0\ Und))) \\
a & \rightarrow & (ite\ (A \wedge B)\ \ \texttt{T} \\
& & \quad (ite\ (\neg A \wedge B)\ \texttt{F}\ \ Und)) \\
b & \rightarrow & (ite\ (A \wedge B)\ \ 3 \\
& & \quad (ite\ \ (A \wedge \neg B)\ \ \text{-10} \\
& & \quad\quad (ite\ \ (\neg A \wedge B)\ \ 0 \\
& & \quad\quad\quad (ite\ \ (\neg A \wedge \neg B)\ \ 4\ Und)))
\end{array}
$$

Figure 5.12: Variational model corresponding to the plain models in Fig. 5.11.

a commutative monoid—and can be built in any order as long as there are no duplicate variants; a scenario that is impossible by the property of synchronization on choices.

However, there are some notable differences. Where variational SAT models clearly compressed results by preventing duplicate values with constant variables, the variational SMT model allows for duplicate values, if those values are produced out of order. For example, both models for i and c contain duplicate values. The i model has duplicate -1 and the c model contains duplicate 0. However only one: c is easy to check in $\mathcal{O}(1)$ time; each call to COMBINE could check the last immediate value to prevent duplicate branches. In contrast, the duplicate -1's for i occur in variants that are likely to occur with several other plain models between them, namely the models for the $C_{\texttt{TF}}$ and $C_{\texttt{FF}}$ variants. Hence, a check during COMBINE would require $\mathcal{O}(n)$ time, where n is the number of satisfiable variants that i occurs in. While such a case is easily avoided in an implementation by tracking the values a variable has been previously assigned, we provide only a minimum specification and thus leave the details to an implementation. Lastly, the use of Und may seem unattractive. While all bindings in the model end with an Und, a binding cannot result in an Und as that would imply a variant that was found to be satisfiable but was not satisfiable, and hence would be indicative of a bug in the variational solver implementation. Mathematically inclined readers may observe that the monoid variational SMT models form corresponds to the free monoid formed by lists.

5.6 A complete variational SMT example

With variational SMT solving formally specified. We present a complete example of solving a variational SMT problem. Consider the query formula $h = ((1 + 2 < (i - A\langle k, l\rangle)) \wedge a) \wedge (B\langle c, \neg b\rangle \vee b)$ with two choices parameterized by the dimensions A and B. Derivation of the variational core for h begins with all evaluation contexts and all stores Δ, Γ initialized to $\emptyset$.

The root of h is $\wedge$ and thus E-AND is the only applicable rule. From E-AND we have $v_1^{\mathbb{Z}} = ((1 + 2 < (i - A\langle k, l\rangle)) \wedge a)$, and $v_2^{\mathbb{Z}} = (B\langle c, \neg b\rangle \vee b)$. We traverse $v_1^{\mathbb{Z}}$ first, leading to a recursive application of E-AND. We denote recursive levels with a tick mark: $'$, thus $v_1^{\mathbb{Z}'} = (1 + 2 < (i - A\langle k, l\rangle))$ is the left child of $v_1^{\mathbb{Z}}$, with $v_2^{\mathbb{Z}'} = a$ as the right child.

The root of $v_1^{\mathbb{Z}'}$ is an inequality, so the only way to progress is to try to accumulate $v_1^{\mathbb{Z}'}$. The accumulation will succeed; in accumulation, only A-INEQ-V can apply as accumulation will be unable to transform the right child of $v_{\mathit{1}}^{\mathbb{Z}\prime}$ to a symbolic value due to the presence of a choice. A-INEQ-V will further destruct $v_{\mathit{1}}^{\mathbb{Z}\prime}$ to $ar_{\mathit{1}}^{\mathbb{Z}} = 1 + 2$ and $ar_{\mathit{2}}^{\mathbb{Z}} = i - A\langle k, l\rangle$. $ar_{\mathit{1}}^{\mathbb{Z}}$ will be accumulated to a single symbolic value by application of A-ARITH-S and A-REF on the literals 1 and 2 yielding $ar_{\mathit{1}}^{\mathbb{Z}} = s_{12}$, with store $\Delta = \{(s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$.

Using the resultant store from accumulating $ar_{\mathit{1}}^{\mathbb{Z}}$, accumulation on $ar_{\mathit{2}}^{\mathbb{Z}}$ will yield the term $s_i - A\langle k, l\rangle$. The variable i will be accumulated to a symbolic value with A-REF and the choice will be passed over by A-CHC. Thus we have the accumulated result for $v_{\mathit{1}}^{\mathbb{Z}\prime}$ as the intermediate term $v_{\mathit{1}\ acc}^{\mathbb{Z}} = s_{12} < (s_i - A\langle k, l\rangle)$ with store $\Delta = \{(i, s_i), (s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$.

With the left child of $v_{\mathit{1}}^{\mathbb{Z}\prime}$ accumulated, E-AND attempts to continue evaluation on the right child and will succeed. Notice that this is a special case as the root of $v_{\mathit{1}}^{\mathbb{Z}}$ is $\wedge$ and so is the root of h. Thus, $v_{\mathit{2}}^{\mathbb{Z}}$ will transform a to a symbolic value through accumulation using the previous store and assert the symbolic value in the base solver with E-SYM. The resulting intermediate term will be $s_{12} < (s_i - A\langle k, l\rangle) \wedge \bullet$, with stores $\Gamma = \{s_a\}$, $\Delta = \{(a, s_a), (i, s_i), (s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$ and will be reduce to the intermediate result $v_{\mathit{1}\ core}^{\mathbb{Z}} = s_{12} < (s_i - A\langle k, l\rangle)$ with the same stores via application of E-AND-R.

We have now returned back to the top level call to E-AND with a result for the left child and populated stores. Evaluation will proceed on the right child $v_{\mathit{2}}^{\mathbb{Z}}$. $v_{\mathit{2}}^{\mathbb{Z}}$'s root is a disjunction, and thus to proceed evaluation switched to accumulation by applying E-ACC. The accumulation is straightforward; the left child is the choice $B\langle c, \neg b\rangle$ and is returned by A-CHC. The right child is a single variable, and thus is translated to the symbolic value s_b. Thus we have the final result for $v_{\mathit{2}}^{\mathbb{Z}}$, $v_{\mathit{2}\ core}^{\mathbb{Z}} = B\langle c, \neg b\rangle \vee s_b$ and the variational core of h, $h_{core} = s_{12} < (s_i - A\langle k, l\rangle) \wedge (B\langle c, \neg b\rangle \vee s_b)$ with stores $\Gamma = \{s_a\}$, $\Delta = \{(b, s_b), (a, s_a), (i, s_i), (s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$.

With the variational core derived we can begin choice removal. We assume an empty configuration for the remainder of the example. The exact semantics of a vc is implementation specific. For example, our prototype variational SAT solver pre-populates the configuration with a generated configuration based on the user vc. In contrast, the prototype variational SMT solver checks the dimensions assignments of true or false in C-CHC are valid with respect to the vc, if not then the variant is skipped.

Choice removal begins with the variational core in the focus and an evaluation context $z^{\mathbb{Z}} = \top$, because h_{core}'s root is $\wedge$ only C-BOOL applies moving $s_{12} < (s_i - A\langle k, l\rangle)$ into the focus and storing the right child in the context: $z^{\mathbb{Z}} = \cdot \wedge (B\langle c, \neg b\rangle \vee s_b) :: \top$. With $s_{12} < (s_i - A\langle k, l\rangle)$ as the focus, the only applicable rule is C-INEQ due to $<$ at the root of the focus. C-INEQ again recurs left, focusing on the sub-term s_{12} with context $z^{\mathbb{Z}} = \cdot < (s_i - A\langle k, l\rangle) :: \cdot \wedge (B\langle c, \neg b\rangle \vee s_b) :: \top$ which states that s_{12} exists in the left child of an inequality which also exists in the left child of a conjunction.

We have arrived at the base case with a symbolic value in focus, and the immediate parent in the evaluation context is an inequality. To proceed we need to *switch* to begin processing the right child of the inequality; thus we must apply C-INEQ-INL. C-INEQ-INL swaps the symbolic with the un-processed right child held in the context, hence we have $(s_i - A\langle k, l\rangle)$ in focus with context $z^{\mathbb{Z}} = s_{12} < \cdot :: \cdot \wedge (B\langle c, \neg b\rangle \vee s_b) :: \top$. Subtraction, $-$, is a previously unseen relation. When a new relation is found, choice removal will recur into the left child. In this case $- \in \oplus$ and so C-ARITH applies. C-ARITH moves s_i into the focus and extend the evaluation context to $z^{\mathbb{Z}} = \cdot - A\langle k, l\rangle ::$

$s_{12} < \cdot :: \cdot \wedge (B\langle c, \neg b\rangle \vee s_b) :: \top$.

With s_i in the focus, we've arrived at another base case, only this time when the switch occurs a choice will be in focus. The switch is performed by C-ARITH-INL and yields $A\langle k, l\rangle$ as the focus with context $z^{\mathbb{Z}} = s_i - \cdot :: s_{12} < \cdot :: \cdot \wedge (B\langle c, \neg b\rangle \vee s_b) :: \top$. Now the heart of choice removal applies, because we have $C = \emptyset$, the only applicable rule with a choice in the focus is C-CHC. C-CHC creates two recursive calls, one for each alternative using *the same context*, thus we'll have $C = \{(A, \mathsf{true})\}$, with focus k, and context $z^{\mathbb{Z}} = s_i - \cdot :: s_{12} < \cdot :: \cdot \wedge (B\langle c, \neg b\rangle \vee s_b) :: \top$.

For the remainder of the example we'll continue with only true alternatives; the other variants follow similar paths. Accumulation is called on the introduced plain terms, converting k to s_k and extending the accumulation store to
$\Delta = \{(k, s_k), (b, s_b), (a, s_a), (i, s_i), (s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$.

With a symbolic value in focus, and with the context already switched to the right child, we have come to a sequence of base cases which perform the folds, in this case C-ARITH-INR applies. C-ARITH-INR calls accumulation on $s_i - s_k$. $s_i - s_k$ has not yet been observed in accumulation and thus the new symbolic value s_{ik} will be generated. This yields s_{ik} in the focus, with $\Delta = \{(s_i - s_k, s_{ik}), (k, s_k), (b, s_b), (a, s_a), (i, s_i), (s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$, and $z^{\mathbb{Z}} = s_{12} < \cdot :: \cdot \wedge (B\langle c, \neg b\rangle \vee s_b) :: \top$.

With s_{ik} in focus, we have yet another base case which consumes some context, only this time we consume the inequality using C-INEQ-INR. C-INEQ-INR calls accumulation on $s_{12} < s_{ik}$, similar to the previous call over $-$, this call produces a new symbolic value and extends the accumulation store. Hence, we'll have s_{12ik} in the focus, with $\Delta = \{(s_{12} < s_{ik}, s_{12ik}), (s_i - s_k, s_{ik}), (k, s_k), (b, s_b), (a, s_a), (i, s_i), (s_1 +$

$s_2, s_{12}), (2, s_2), (1, s_1)\}$, and $z^{\mathbb{Z}} = \cdot \wedge (B\langle c, \neg b\rangle \vee s_b) :: \top$ as the evaluation context. With a symbolic value in the focus, and a context indicating the left child of a relation, choice removal switches to process the right alternative. The relation in this case is $\wedge$ and so C-BOOL-INL applies to execute the switch yielding $B\langle c, \neg b\rangle \vee s_b$ in the focus, and $z^{\mathbb{Z}} = s_{12ik} \wedge \cdot :: \top$. $\vee$ is a relation that is previously unseen, and thus C-BOOL recurs into its left child yielding the choice $B\langle c, \neg b\rangle$ in the focus and
$z^{\mathbb{Z}} = \cdot \vee s_b :: s_{12ik} \wedge \cdot :: \top$ as the context.

We have arrived at the second choice. $B \notin dom(C)$ and thus only C-CHCapplies. Following the true alternative for B, accumulation is called on c yielding s_c in the focus, with $C = \{(B, \mathsf{true}), (A, \mathsf{true})\}$, $\Delta = \{(c, s_c), (s_{12} < s_{ik}, s_{12ik}), (s_i - s_k, s_{ik}), (k, s_k)$ $, (b, s_b), (a, s_a), (i, s_i), (s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$, and $z^{\mathbb{Z}} = \cdot \vee s_b :: s_{12ik} \wedge \cdot :: \top$. All that is left is a switch and then to complete the fold with the symbolic values. C-BOOL-INL switches the context placing s_b in the focus and yielding $z^{\mathbb{Z}} = s_c \vee \cdot ::$ $s_{12ik} \wedge \cdot :: \top$, which will be followed by C-BOOL-INR to disjunct s_c and s_b using accumulation. The resulting term will have s_{bc} in the focus, $\Delta = \{(s_c \vee s_b, s_{cb}), (c, s_c), (s_{12} < s_{ik}, s_{12ik}), (s_i - s_k, s_{ik}), (k, s_k)$ $, (b, s_b), (a, s_a), (i, s_i), (s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$, and $z^{\mathbb{Z}} = s_{12ik} \wedge \cdot :: \top$, which leaves only one more reduction until model generation. C-BOOL-INR applies again to conjunct the last two symbolic values, yielding $z^{\mathbb{Z}} = \top$, s_{12ikbc} in the focus and $\Delta = \{(s_{12ik} \wedge s_{bc}, s_{12ikbc}), (s_c \vee s_b, s_{cb}), (c, s_c), (s_{12} < s_{ik}, s_{12ik}), (s_i - s_k, s_{ik}), (k, s_k)$ $, (b, s_b), (a, s_a), (i, s_i), (s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$.

Thus we have reached the variant parameterized by $C = \{(B, \mathsf{true}), (A, \mathsf{true})\}$ C-EVAL applies due to $z^{\mathbb{Z}} = \top$ and the symbolic value in the focus, E-SYM will yields $\bullet$ with $z^{\mathbb{Z}} = \top$, indicating that it is safe to query a model for this variant from the base solver. Due to the two application of C-CHC three other variants will be found during backtracking beginning with the dimension used in the most recent application. In this case that is the dimension B, and thus the next variant that will be found is parameterized by $C = \{(B, \mathsf{false}), (A, \mathsf{true})\}$ with context $z^{\mathbb{Z}} = \cdot \vee s_b :: s_{12ik} \wedge \cdot :: \top$ and $\Delta = \{(c, s_c), (s_{12} < s_{ik}, s_{12ik}), (s_i - s_k, s_{ik}), (k, s_k)\ , (b, s_b), (a, s_a), (i, s_i), (s_1 + s_2, s_{12}), (2, s_2), (1, s_1)\}$.

5.7 Variational SMT arrays

With variational SMT solving fully specified we can reflect on the generalization recipe from the previous sections. Say we desired to add the SMT background for REALS. Doing so would follow the straightforward recipe demonstrated with INTS: From the SMTLIB2 standard we have a set of primitive operators, we would define wrapped primitive versions for each operator. Using these wrapped operators we would define new cases for accumulation and a base case for evaluation indicating that the new operator requires accumulation. Then we would add the new domain to the syntactic categories in Fig. 5.5. Choice removal would be extended with three new rules, a rule to begin the processing of the left child of the relation, a rule to switch from the left child to the right only when a symbolic value is in the focus, and a rule that performs the fold by combining two symbolic values and thus consuming some of the context.

In essence, we have a recipe for a generalized variational folding algorithm over binary relations, that forces reuse of shared terms and is applied to the domain of SAT and SMT solvers. Recall that a symbolic value is a sequence of statements in the base solver. Thus, another way to view our generalized folding algorithm is as a compiler from the language of variational SAT or SMT to plain SMTLIB2 script. The stages of the compiler in this interpretation are straightforward: We parse a variational SAT or SMT problem to an abstract syntax tree in an intermediate language. The intermediate language enables optimization passes and is easier to work with than the object language. Accumulation and evaluation produce a variational core, which can be seen as another, further reduced core language, or as a syntax object that encapsulates the variational aspects of the input. The core language is then operated by choice removal which deterministically produces the variant syntax objects. Code generation is spread across generation of symbolic values in accumulation, assertion of constraints in evaluation, and calls to PUSH or POP during choice removal, specifically during C-CHC.

The exact ordering of the operations in the base solver, or the ordering of code generation, is implementation specific. In the prototype solvers that we have produced and will discuss further in the next chapter, code generation that corresponds to generating symbolic values occurs when the symbolic value becomes known to Δ. When a configuration occurs, the PUSH/POP calls encapsulate the operator the choice was nested in and any new symbolic values which result from the configuration. This ensures sharing of terms that are in the same assertion levels. For example, consider the case of s_{12} from Section 5.6, s_{12} will be shared once for each variant because it is plain, thus the code which defines it must occur *before* a PUSH and POP call:

```
(declare-const s1 Int)          ;; literal declarations
(declare-const s2 Int)
(declare-fun s12 () Int
  (+ s1 s2))
(push)                          ;; push for true alternative of A
```

⋮

Similarly for terms such as s_{ik}, which will be shared twice but are not plain, because i was transformed into a symbolic before the choice was configured its declaration still occurs outside the PUSH/POP block. In contrast, k is parameterized by a choice and thus its declaration occurs inside a PUSH/POP block:

```
(declare-const s_1 Int)          ;; literal declarations
(declare-const s_2 Int)
(declare-fun s_12 () Int
  (+ s_1 s_2))
⋮
(declare-const s_i Int)           ;; i is declared
(push)                            ;; push for true alternative of A
(declare-const s_k Int)
(declare-fun s_ik () Int
  (< s_i s_k))
⋮
```

In this case, the ordering of symbolic value generation forced s_k to be inside the PUSH call so that it is removed from the local scope of the solver after a POP and thus the boundaries between alternatives do not leak information. Notice that the inference rules in Section 5.2, Section 5.3, and Section 5.4 guarantee this behavior because symbolic value creation is ordered according to levels of variation. For example, plain terms are level 0 since no configuration has happened. In a sense they are globally scoped and thus become symbolic values or •'s first. It is only after a configuration occurs from C-CHC that more plain terms are introduced. When a configuration occurs, a new PUSH/POP block is entered, and thus any calls to accumulation which occur inside it occur inside that block in the base solver and correspond to level 1. Furthermore, the level is propagated by the $D \in C$ check in C-CHC-T and C-CHC-F.

To demonstrate the generality of this design we now consider the case of adding SMT arrays. To add SMT arrays we treat arrays as a new kind of relation. By treating them like any other relation, we take advantage of the aforementioned ordering behavior and offload the hard work to choice removal. SMT arrays are defined by two operations $(\mathit{store}\ a\ i\ e)$ and $(\mathit{select}\ a\ i)$, where a is a variable representing the array, i is an index into the array, and e is an element of the array. Each operation must exist inside a boolean constraint to propagate information about the array, for example $(\mathit{store}\ a\ 2\ b)$ leaves a unconstrained, while $a \equiv (\mathit{store}\ a\ 2\ b)$ adds a constraint that forces position 2 to store b in a. Similarly, select must exist in a constraint, e.g., $x \equiv (\mathit{select}\ a\ 2)$ will add a constraint which sets x to b.

Assume that we restrict i to only INT, using the recipe above we would wrap these operations, add new rules to accumulation to accumulate anything in the a, i, or e positions and then extend $z^{\mathbb{Z}}$ such that a context could be captured wherever choices may occur. For example we might have:

$$
\begin{array}{rcl}
z^{\mathbb{Z}} & ::= & \top \\
& | & \neg \cdot :: z^{\mathbb{Z}} \\
& | & \dagger \cdot :: z^{\mathbb{Z}} \\
& | & \cdot \otimes\ v^{\mathbb{Z}} :: z^{\mathbb{Z}} \\
& | & s \otimes \cdot :: z^{\mathbb{Z}} \\
& | & \cdot \oplus\ v^{\mathbb{Z}} :: z^{\mathbb{Z}} \\
& | & s \oplus \cdot :: z^{\mathbb{Z}} \\
& | & \cdot \bowtie\ v^{\mathbb{Z}} :: z^{\mathbb{Z}} \\
& | & s \bowtie \cdot :: z^{\mathbb{Z}} \\
& | & (\mathit{store}\ \cdot\ s\ s) :: z^{\mathbb{Z}} \\
& | & (\mathit{store}\ s\ \cdot\ s) :: z^{\mathbb{Z}} \\
& | & (\mathit{store}\ s\ s\ \cdot) :: z^{\mathbb{Z}} \\
& | & (\mathit{select}\ \cdot\ s) :: z^{\mathbb{Z}} \\
& | & (\mathit{select}\ s\ \cdot) :: z^{\mathbb{Z}}
\end{array}
$$

Which is verbose but would work. Now consider a formula which contains a nested choice in an arithmetic expression in the element slot of select but not store: $f = (a \equiv (\mathit{store}\ a\ 2\ (i - A\langle k, l\rangle))) \wedge (i \equiv \mathit{select} a 2) \wedge (i \equiv l)$. The conjunctions indicate separate statements in SMTLIB2 due to the behavior of the assertion stack. The formula is noteworthy because both the select and $i \equiv l$ call are plain and thus will be processed by evaluation/accumulation *before* choice removal via the E-AND-L and E-AND-R. So the formula may seem problematic because calling a select before a store in other paradigms would lead to an error. However this will not be the case, consider the compiled SMTLIB2 script for f in Fig. 5.13. From the compiled output, we see that evaluation/accumulation did find the plain select and $i \equiv l$ constraints and assert them before the choice is processed. However, because constraints in the base solver can be unordered due to the implicit conjunction of all assertions in an assertion level, the out

```
(declare-const s_a (Array Int Int))        ;; variable declarations
(declare-const s_i Int)
(declare-const s_l Int)
(declare-const s_2 Int)
(declare-fun s_sel () Int                  ;; select
  (= s_i (select s_a s_2)))
(declare-fun s_il () Int                   ;; equivalency constraint
  (= s_i s_l))
(assert s_sel)
(assert s_il)
(push)                                     ;; push for true alternative of A
(declare-const s_k Int)
(declare-fun s_ik () Int
  (- s_i s_k))
(assert (= s_a (store s_a (s_2) (s_ik))))
(check-sat)
(get-model)                                ;; plain model for true alternative
(pop)
(push)                                     ;; push for false alternative of A
(declare-fun s_il () Int
  (- s_i s_l))
(assert (= s_a (store s_a (s_2) (s_il))))
(check-sat)
(get-model)                                ;; plain model for false alternative
(pop)
```

Figure 5.13: The SMTLIB2 output from compiling f.

of order $select$ and constraint $i \equiv l$ are not problematic. Furthermore, we see that the SMTLIB2 snippet has desirable properties: every plain or variational term, such as l and i, can be shared. When a PUSH/POP block is entered the block is as small as possible, thus sharing is maximized as much as possible. Due to the symbolic values generated by accumulation/evaluation, variation does not spread past the immediate relation and thus other relations do not suffer from *variational infection* [132].

We have demonstrated the generality of our approach by extensions with the differing domains of arithmetic over integers and arrays. Key to the approach is the indirection with symbolic values, and the use of a zipper to construct a generalized variational folding algorithm over any unary, binary, or ternary relation. Thus, with just what has been presented here we can support the rest of the core theories in SMTLIB2 using the aforementioned extension recipe. We return to this point in Chapter 7 when we discuss the implications for a variational logic programming language.

Chapter 6: Case Studies

We have formalized variational SMT and SAT solving. However, we have yet to investigate the performance of our methods. Recall from Chapter 1 that a motivating reasons for a variational solver was that if we only compute shared terms once, then we should observe a speedup in runtime performance when solving sets of related SAT problems because information is reused. In this chapter, we investigate and verify these claims.

Assessing the performance of SAT and SMT solvers is notoriously difficult [59] because it depends on the input problem to the SAT or SMT solver. The issue is related to the computational hardness of the input. Hardness, in this domain is estimated by the ratio of clauses in the SAT or SMT problem to the number of variables. Conceptually, if there are many clauses and not many variables then the problem is over-constrained and it is easy to decide UNSAT, however if there are few clauses but many variables then the problem is under-constrained and it is easy to decide SAT. Thus, hard problems rest in a *phase transition* zone where the ratio of clauses to variables is neither over-constrained nor under-constrained [59].

To investigate the performance of our methods, we construct a prototype variational solver, VSAT in the Haskell programming language [66] and quantitatively compare it to incremental and non-incremental SAT solving. Assessing the prototype in realistic conditions is difficult as there does not exist a corpus of accepted representative variational SAT problems. Thus, to test the prototype in as realistic conditions as possible we utilize real-world data from a previous study by Nieke et al. [102] from the SPL community.

Before we describe the datasets and resulting variational SAT problems we first introduce some terminology from the SPL community. A software product-line is an instance of variational software, more formally it is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [6, 40, 105].

A good example of a software product-line is the Linux kernel [126]. The Linux kernel is a set of core assets which devise an operating system, but the assets are parameterized by *features* which, in this case, are the Boolean conditions of conditional-compilation statements such as `#ifdef`s. To select the particular kind of kernel to build, the Linux kernel uses the `KConfig` [37] tool to enable or disable features and thus specify the exact kernel to build. The set of features and their dependencies which determine the product, or in this case determine the kernel that is built is call a *feature model*[68].

It is common to express feature models as a SAT formula where variables are features, and dependencies are expressed using logical connectives. Thus, reasoning about feature models with a SAT solver is an active sub-field in software product-lines [20, 55, 19, 123]. For example, a *void analysis* uses a SAT solver to determine that a product is possible, and a *core analysis* manipulates the feature model to check that a given feature must be `T` (or enabled in the SPL terminology) for every viable product. Conceptually in this domain, if a SAT is returned from the solver then the resulting model is an assignment of features which specifies a viable product. If an UNSAT is returned than no viable product exists given the constraints on the feature model for the software product-line. Further analysis on the product line are

6.1 Experimental Methodology

For the remainder of the chapter, we must distinguish between concepts in the application domain, such as a void or core analysis, and concepts in the solver domain, such as a query or choice. In general, we focus on the solver domain as it is our primary concern.

Nieke et al. provides two datasets[1], *automotive02* and *financialServices1* which encode the evolution histories of two feature models as propositional formulas. We refer to these as the *auto* dataset, and *fin* dataset for the remainder of this chapter. Since these datasets encode evolution histories, variants in our analysis correspond to snapshots of feature models over time, and a plain model of a variant corresponds to a void analysis over that feature model. For example, a variant of the *auto* dataset is a C_2 formula which encodes a feature model at time 0, and another variant encodes *the same* feature model at time 2, where $0 < 2$. Recall the possible existence of extra variants from Section 3.4, since extra variants may exist given a VPL encoding of the datasets we use the phrase *version variant* to refer to variants that are snapshots of a feature model in the application domain. For example, the variant which corresponds to a feature model at time 0 is a version variant, but the variant which corresponds to a feature model at time 0 *and* and time 2 are non-version variants.

[1] see https://gitlab.com/evolutionexplanation/evolutionexplanation

We assess the performance characteristics VSAT by attempting to answer the following research questions.

RQ1 How does variational solving scale as variation increases?

RQ2 What is the impact of base solvers on performance?

RQ3 What is the impact of sharing on performance?

RQ4 What is the cost of solving a plain formula on VSAT?

To investigate **RQ1** and **RQ2**, we consider all variants of the VPL formulas constructed for each dataset, rather than just the version variants that are of interest in the application domain. This allows us to better evaluate how VSAT scales to accommodate variability. For **RQ3**, we hypothesize that VSAT will show observable speedups as sharing increases, which would validate our method of deriving a variational core. To investigate this, we restrict the analysis to consecutive version variants (i.e., consecutive monthly snapshots of a feature model), and observe performance as sharing is left uncontrolled. Finally, **RQ4** provides insight on the overhead incurred by variational solving, which we investigate by inputting each version variant as a propositional logic formula rather than a single variational formula for each solver used in **RQ2** and **RQ1**.

6.1.1 Data Description and Encoding

Nieke et al.'s formulas collapse sets of C_2 formulas to a single formula using implications on an SMT variable that represents a moment in time. A two-pass process was used to translate Nieke et al.'s formulas into VPL—one pass to parse to an internal representation and another to detect and convert Nieke et al.'s temporal ranges to choices, nesting the implied clauses into the true alternative.

The two datasets differ in important ways. The *auto* dataset encodes four monthly snapshots while the *fin* dataset encodes ten. Hence, the *auto*'s query formula represents 16 variants, while the *fin* query formula represents 1,024 variants. For **RQ3** and **RQ4**, we construct several vc's to restrict the analysis to version variants. The vcs range from ones that enable only one version variant (for **RQ4**): $vc_{auto\_V_1} = (V_1 \wedge \neg V_2 \wedge \neg V_3 \wedge \neg V_4)$ to vcs that enable only consecutive version variants (for **RQ3**): $vc_{auto\_V12} = V_1 \veebar V_2$.

For **RQ4** we decouple performance from the number of variants by performing an initial pass over the query formula to replace choices representing non-consecutive versions variants (e.g. a variational formula which represents V_1 and V_2 but not V_1 and V_3) with their false alternatives (which contain the value `T`). Then we construct a vc to forbid non-version variants. As an example, the *auto* dataset, yields three data points by this process, the change from versions V_1 to V_2, V_2 to V_3, and V_3 to V_4. All results presented for **RQ3** were calculated using the z3 [43] SAT solver.

Measuring Performance To answer our research questions, we construct four different solving algorithms using our prototype tool. We use the notation *<formula>→<solver>* to describe, for each algorithm, whether the query formulas and solver are plain (p) or variational (v), respectively. The algorithms are: the baseline, *p→p*, which runs plain formulas on a plain solver; the variational case, *v→v*, which runs a variational formula on the variational solver; the overhead case, *p→v*, which runs plain formulas on the variational solver; and the typical case, $v{\rightarrow}p$, which runs the variational formula, variant by variant, on a plain solver. Inputs for each algorithm are constructed by configuring the query formula, thus ensuring that the same variation context is used across algorithms.

We construct the $p{\rightarrow}p$ algorithm by configuring the query formula to its variants *before* benchmarking begins. These formulas are then sent to the base solver one-by-one, with the solver begin shut down and initialized between runs, thus preventing the solver form maintaining any learned information. The $p{\rightarrow}v$ case corresponds to **RQ4** and elucidates the potential overhead of solving a plain query on a variational solver. We perform the same pre-processing as the $p{\rightarrow}p$ case but send each plain formula to VSAT instead. This provides insight into the cost incurred by the reduction engine. For $v{\rightarrow}p$, we configure the query formula to retrieve variants *during* benchmarking. Each formula is sent to the base solver *with* the solver maintaining information between queries. This gives insight into the overhead incurred by configuring a variational formula, and the benefits of the internal caching in the base solver. Notable, this case keeps the base solver running, performing each call in incremental mode, thus this case corresponds to the typical use of an incremental solver in applications that utilize incremental SAT solvers.

We construct a variational model for all algorithms since it is unclear how to combine plain models, and since the storage of plain models is an orthogonal concern to performance, we sought to keep convolved variables constant.

Unless specified, all results are a bootstrapped statistical average representing numerous raw measurements.[2] For **RQ2** we repeat **RQ1** with four different base solvers: z3 [43], cvc4 [14], yices [44] and boolector [23], each of which called through the widely used Haskell library [48]. To assess **RQ2** we perform a Kruskall-Wallis test [103] followed by a pairwise Wilcox test [135] with Holm-Bonferroni p-value correction [62] in the R programming language [106] v4.0.3 and assume a 5% significance level. For **RQ3**, we similarly normalize the data to the baseline ($v \rightarrow p$), fit a linear model, and statistically assess differences by repeating the aforementioned statistical tests. For **RQ4**, we retrieve the raw measurements from the bootstrapped average and assess statistical differences identically to **RQ3** but do not fit any models to the data. Furthermore, the variational input is nuanced for **RQ4** as each data point is on a variant, which is necessarily plain. Thus, **RQ4** is a special case; for **RQ4** $v \rightarrow v$ inputs the variational formula but utilizes a variant context to restrict the solver to the version variant. $v \rightarrow p$ performs configuration to configure for the version variant and then runs the variant on a base solver during benchmarking. All results, including variational models and statistical analysis scripts, are available online.[3]

6.2 Results and Discussion

The datasets yielded dissimilar query formulas: the *auto* query formula consisted of 4,212 choice terms (not including terms in a choice's alternatives), and 26,808 plain

[2] Using v0.2.5 of the gauge [104] library and v8.6 of the sbv [48] library with solver seeds set to `1729`. All data was collected on a desktop running NixOS 20.09, with an AMD Ryzen 7 2700X CPU @ 4.0GHz, 32GB RAM.We used stack lts-15.7 (GHC 8.8.3) and tested with RTS options "-qg" which enables parallel garbage collection in the Haskell runtime.

[3] https://github.com/lambda-land/VSat-Papers/tree/master/EMSE2021

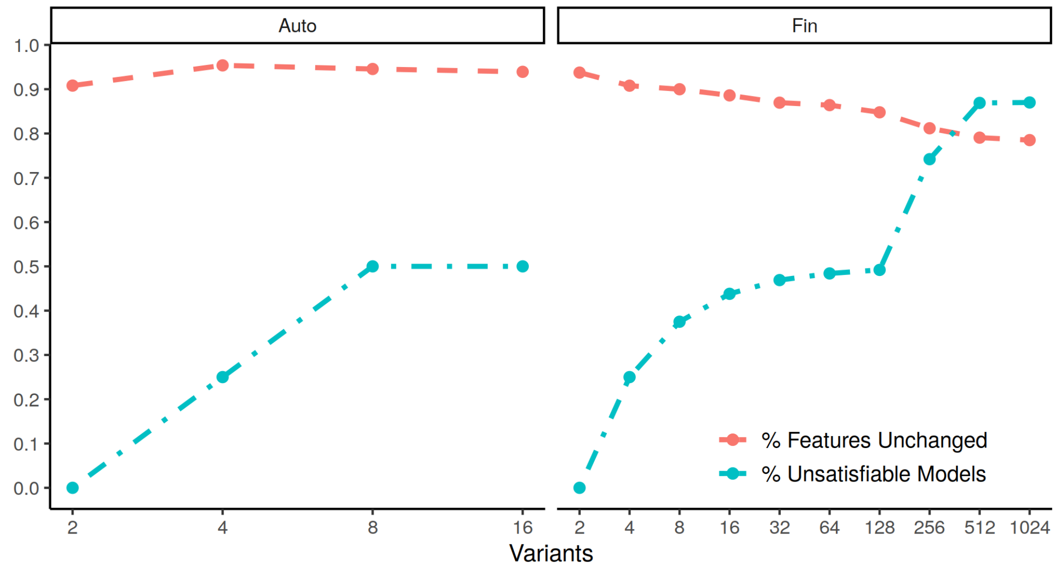


Figure 6.1: Most models found to be unsatisfiable. Only a small portion of features ever flipped to `T`.

terms. In contrast, *fin* had 3,809 choice terms, and 1,441 plain terms. Thus *fin* had larger changes between product line versions. Fig. 6.1 shows the ratio of unsatisfiable models to total plain models, and the ratio of constant features for each product line version (represented by variant count). For both datasets the number of satisfiable models decreased as new versions were considered, and the majority of features in each model never flipped from their initialized value `F` to `T`. Thus, the variational model is likely a compressed version of the set of plain models. Compression metrics were not calculated as this is an orthogonal concern to the performance of variational satisfiability solving.

Variational models permit product analyses without a SAT solver. Fig. 6.1 shows such a purely syntactic analysis: counting disjuncted clauses in the variational model as a representation of satisfiable plain models. We believe post-hoc analyses such as

DataSet	Boolector	CVC4	Yices	Z3
auto	3.29	3.51	3.20	2.62
fin	2.44	2.51	2.50	2.16

(a) Speedup by solver for the most variational case; 16 variants for *auto*, 1024 for *fin*.

Boolector	CVC4	Yices	Z3
623.0	738.6	623.7	862.0
788.8	1026.6	729.2	884.2

(b) Time [s] to solve with $v \rightarrow v$ by solver.

Table 6.1: Time to solve and speedup of most variational case by solver.

this may be useful to feature modelers as they direct attention to impactful versions of the feature model. For example, the change from V_7 to V_8 (128 to 256 Variants) of *fin* clearly constrained the feature model as the number of unsatisfiable models increased from 50% to 80%.

The experiment required 7 days, 6 hours and 21 minutes to complete. Due to the amount of time required to generate the data, we limited the number of raw measurements to 3. Thus, each data point presented in our results is a bootstrapped average of 3 raw measurements. In contrast, the conference version of this work used 56 raw measurements per data point. This experiment quadruples the run time of the conference version, because we repeat the experiment with four base solvers. Thus we restrict the sampling as the experiment was not feasible with the previous rate of sampling. Our results agree with our former work even with the reduced sampling.

6.2.1 RQ1: Performance of Variational Solving as Variation Scales

The VSAT tool outperforms other algorithms as the count of variants to solve increases for every base solver. Fig. 6.2 shows the time to solve the query formula as variants

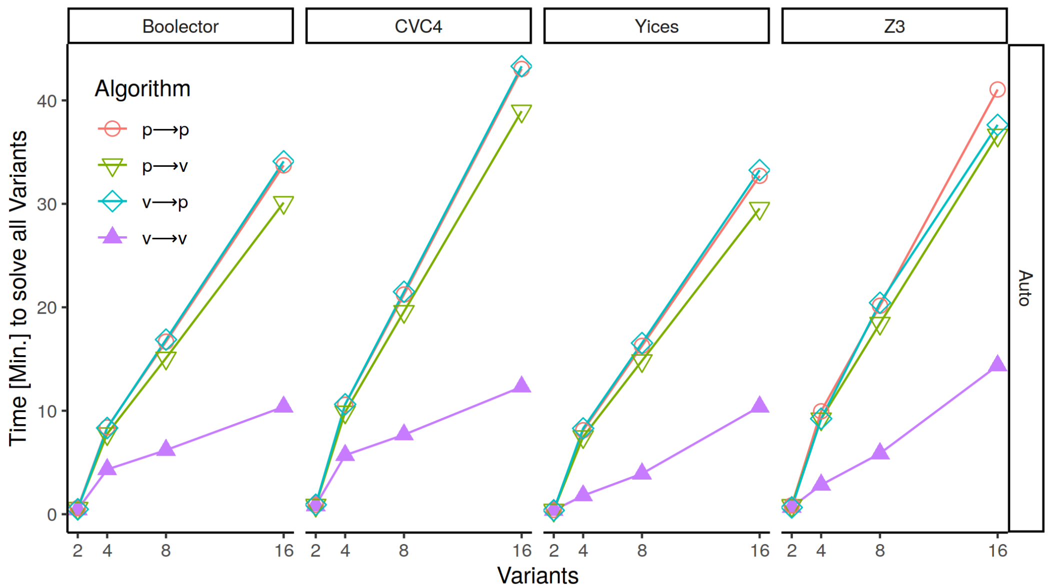


Figure 6.2: (Auto) RQ1: performance as variants increase per base solver. $v{\rightarrow}v$ shows a speedup of 2.8–3.5x for the *auto* dataset depending on base solver.

increase from 2 to 16 for the *auto* dataset for each solver. Similarly Fig. 6.3 shows time to solve by base solver for the *fin* dataset.

For the *auto* dataset, variational solving is faster with an average speedup of 2.60x. For the most variational case (16 variants) the greatest speedup was found to be 3.5x with cvc4. The *fin* dataset shows an average speedup of 4.70x [4]. For the most variational case (1024 variants), cvc4 also showed the greatest speedup at 2.51x. We find that $v{\rightarrow}v$ is statistically different from every other algorithm with p-values of 2.77×10^{-4} ($v{\rightarrow}p$), 1.06×10^{-2} ($p{\rightarrow}p$), and 1.92×10^{-2} ($p{\rightarrow}v$) for *auto* and 1.62×10^{-5} ($v{\rightarrow}p$), 1.92×10^{-5} ($p{\rightarrow}p$), and 1.70×10^{-4} ($p{\rightarrow}v$) for *fin*.

VSAT outperforms the other algorithms because the variational core caches plain

[4] Due to extreme outliers (10x–15.1x speedup) from yices when solving 2–32 variants.

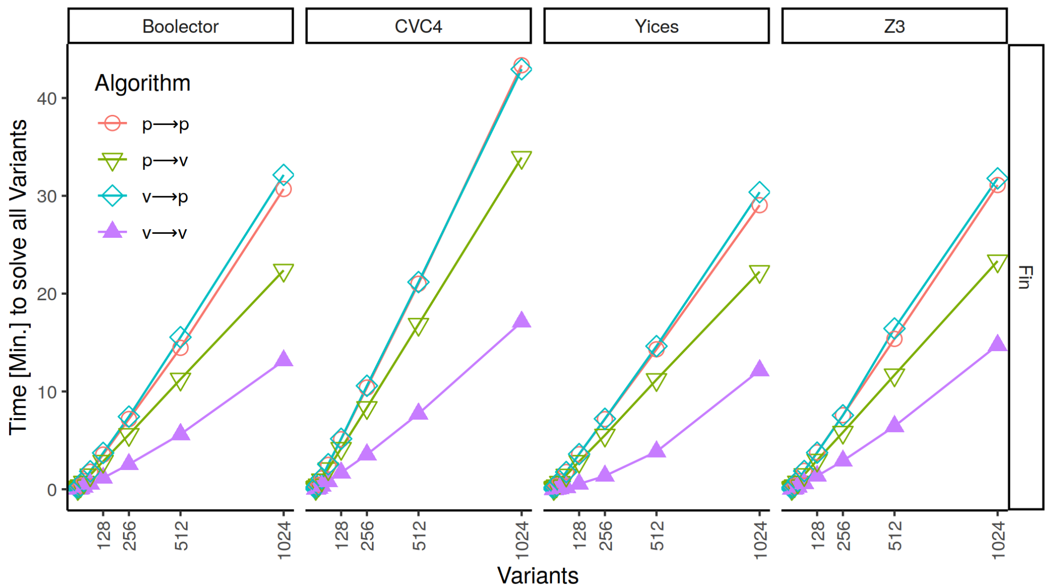


Figure 6.3: (Financial) RQ1: performance as variants increase per base solver. $v{\rightarrow}v$ shows a speedup of 2.4–3.2x for the *fin* dataset depending on the base solver. Overlapping x-axis labels elided.

terms, thereby preventing the re-evaluation of these terms for each variant. By this data, we observed a constant factor speedup. Thus, variational solving still grows linearly in the number of variants being solved.

6.2.2 RQ2: Performance Impact of Base Solver

From **RQ1** we determined that $v{\rightarrow}v$ is faster than the baseline algorithms and that the difference is statistically significant. We observe from Fig. 6.2 and Fig. 6.3 that the $v{\rightarrow}v$ algorithm is robust across every tested base solver and $v{\rightarrow}v$ produced reasonable results with each base solver. We summarize our results in Table 6.1. Notable yices was consistently the most performent base solver for all algorithms and all test cases. For $v{\rightarrow}v$ yices demonstrates not only a high degree of speedup but also a reduction of 238.3 seconds, and 132.8 seconds, in run time from z3 for the most variational case of *auto* and *fin*. Thus, yices is an attractive target for the base solver of future prototype variational SAT solvers.

Cvc4 is also noteworthy; cvc4 benefited the most from $v{\rightarrow}v$ for both datasets with a speedup 3.51x (*auto*) and 2.51 (*fin*). The cvc4 case is interesting as it implies that a base solver which shows poor performance with in the typical use case ($v{\rightarrow}p$; SAT calls occur in an incremental constext, and solver is kept alive) may greatly benefit from the variational solving algorithm we've presented. Although the exact reasons behind this behavior will be particular to the base solver, these results imply that our use case (i.e., heavily exercising the incremental code paths) is peculiar and thus selecting a solver based on only its typical performance may not be representative of its performance in this use case.

6.2.3 RQ3: Performance Impact of Plain Terms

We hypothesize that the ratio of plain terms to total terms should increase the variational solver's performance. Specifically, we hypothesize that as sharing grows, the query formula's variational core is further reduced. We observe this behavior using the z3 data in Fig. 6.4. Both $v \rightarrow v$ and $p \rightarrow p$ showed a statistically significant fit to a linear model. Furthermore, only $v \rightarrow v$ was found to be statistically different from $p \rightarrow p$ and $p \rightarrow v$ with p-values of 6.95×10^{-3} and 4.44×10^{-6} thus confirming that sharing positively correlates to speedups for variational solving in these datasets.

This result is evidence that a dataset's sharing ratio is an important factor in the performance of a variational SAT solver, as we hypothesized. When the sharing ratio is high, the reduction engine produces a smaller variational core. With a smaller variational core, more reuse of plain terms occurs and thus computational time is saved in the base solver. Hence, an avenue of future work is to leverage the laws of the variational logic to automatically refactor input formulas to increase sharing. The consequences of this observation will be particular to the application domain. For software product lines this means that any method to increase sharing between product line versions or the representative SAT problems is desirable; this may be smaller changes with respect to the entire feature model, more frequent snapshots of the feature model, or syntactic manipulations to mitigate the occurrence of new features.

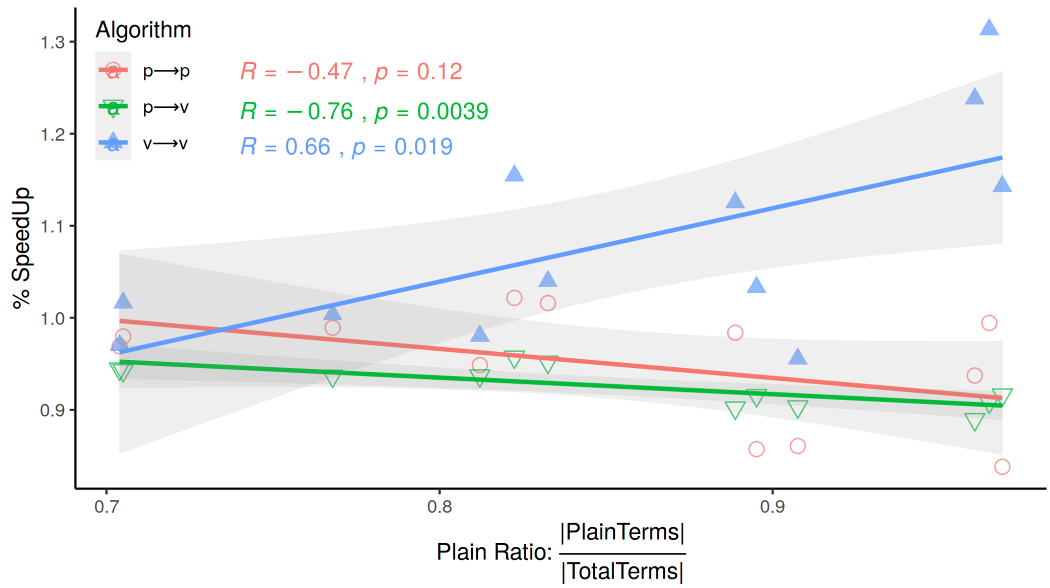
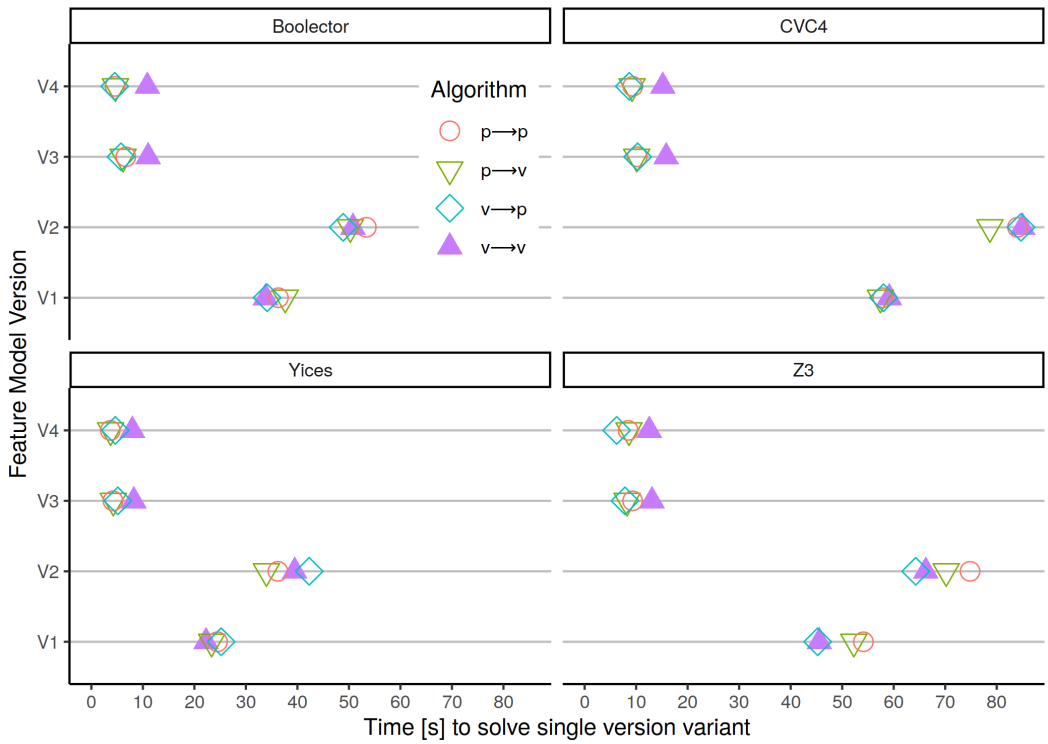


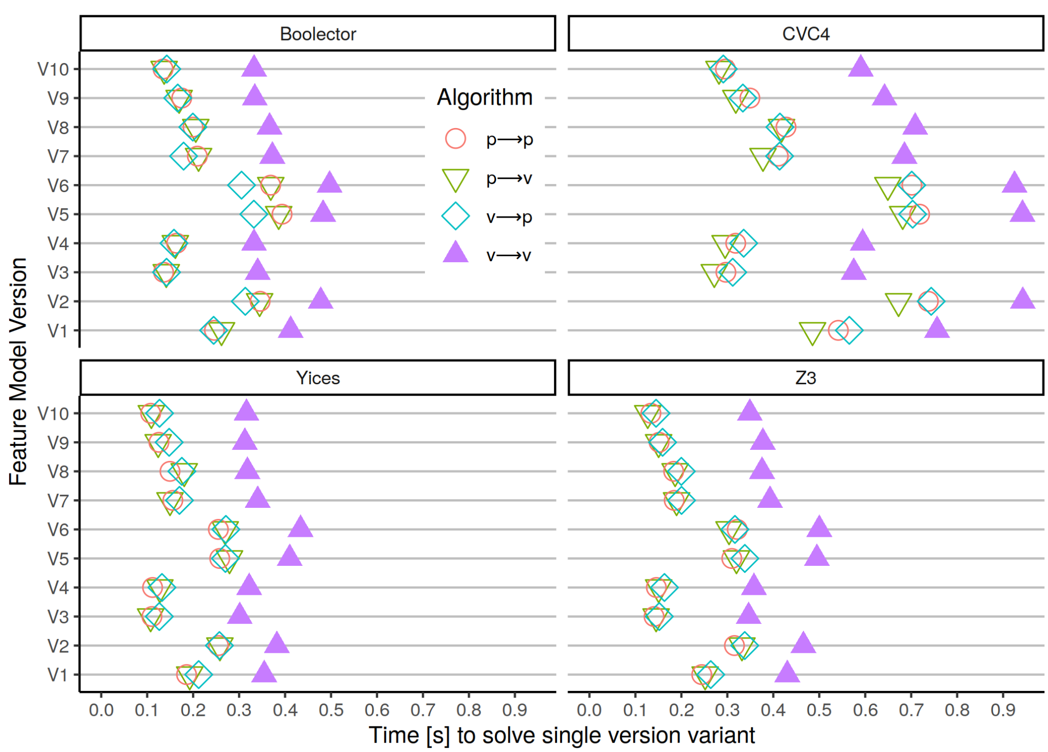
Figure 6.4: RQ3: Performance as a function of plain ratio. We observe that sharing positively correlates to speedup only for $v{\rightarrow}v$, where `% SpeedUp` $= \frac{\text{Algorithm}}{v \rightarrow p}$.

6.2.4 RQ4: Overhead of a Plain Query on VSAT

Fig. 6.5a and Fig. 6.5b displays the bootstrapped averages of each version variant, for each algorithm, and base solver for the *auto*, and *fin* datasets, respectively. Given **RQ2**, and the composition of *fin*, we expect VSAT to show slowdowns for *fin*. This is observed in Fig. 6.5b and is statistically significant for all versions. For *auto*, only the V_1 version variant showed a significant difference between the overhead case, $p{\rightarrow}v$, and $v{\rightarrow}v$, and between the overhead case $p{\rightarrow}v$, and the typical case $v{\rightarrow}p$. Notably, $v{\rightarrow}v$ did not differ from the typical case, $v{\rightarrow}p$. Fig. 6.5a suggests statistically significant differences for other versions but omits variance, hence the discrepancy between the plot and statistical tests. That $p{\rightarrow}v$ was statistically different for V_1 suggests particular formulas may not



(a) (Auto) RQ4: Overhead of $v{\rightarrow}v$ on plain formulas. We observe that $v{\rightarrow}v$ incurs an average slowdown of 9% for *auto*, when solving a version variant.



(b) (Financial) RQ4: Overhead of $v{\rightarrow}v$ on plain formulas. We observe that $v{\rightarrow}v$ incurs an average slowdown of 75% for the *fin* dataset, when solving a version variant.

respond well to the reduction engine, although the exact slowdown will be dependent on the SAT problem.

6.2.5 Threats to Validity

Our results are subject to several threats to validity. Notably, we are unable to make absolute performance claims because our study, with only two product lines, may not be representative. To mitigate this we reused real-world data from Nieke et al.'s previous study [102] and chose dissimilar product lines. We inherit encoding-based threats to validity by reusing Nieke et al.'s formulas but ensured each algorithm experienced identical ordering of plain terms as described in Section 6.1. Furthermore our results, and our prototype solver are based on the widely used Haskell library sbv. However, we believe this is a likely to be a common implementation strategy for a variational solver (i.e., a solver built using a library rather than a foreign function interface, similar to tools built on top of sat4j [83]) it is nonetheless a threat to validity as our prototype directly depends on this library. To mitigate this threat we maintained the same version of sbv throughout the experiment, employed it's interface to interoperate for each base solver, and enforced the same code paths through the library.

We have evinced the scalability claim with RQ1, and shown the translation and automation of incremental solving in Chapter 4. However, our results depend on a VPL formula as input. We believe that VPL formulas can be incrementally and automatically constructed in practice, as described in Chapter 3, as new variants occur or become known. However, assessing the challenges of VPL construction is left to future work.

We do not provide a proof of the soundness of the prototype solver. We mitigate this threat in several ways: we performed property-based testing [34] on our prototype and verified that a satisfiable variant was found to be satisfiable across all algorithms. In addition, we define a property that ensures that for each plain model p, found with $p{\to}v$, $v{\to}p$, and $p{\to}p$, an identical model p' was found by substituting p on the variational model returned from VSAT. We performed the property-based tests with 3,000 generated VPL formulas, finding no counter-examples.

6.3 Variational SMT Results and Discussion

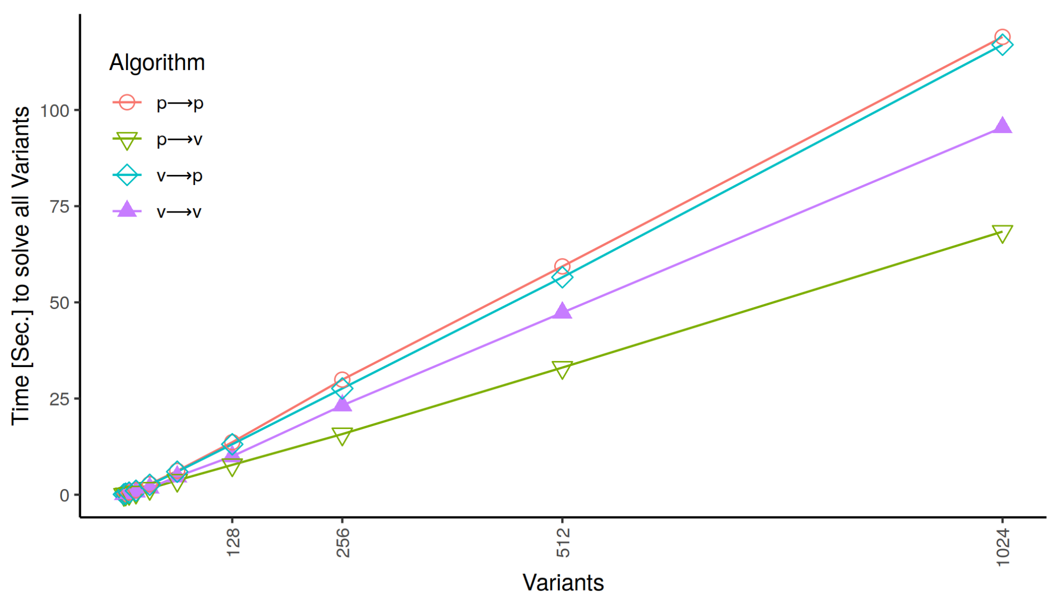


Figure 6.6: (Financial) Performance as variants increase for the variational SMT solver.

We have shown that the variational SAT solver exhibits speedup for two real-world datasets and that the sharing ration of a VPL formula is a significant factor in that speedup. However, we have yet to show that the same is true for the prototype variational SMT solver, VSMT.

To test VSMT we use an SMT version of the *fin* dataset from Nieke et al.'s study. Unfortunately, only the *fin* dataset has an SMT version and so our evaluation of the prototype SMT solver is limited. Furthermore, in the course of encoding the dataset to a VPL$^{\mathbb{Z}}$ formula we discovered type errors in 1,514 formulas out of a total of 4,621 formulas. To utilize the dataset we detected and corrected the type errors during parsing. The type errors were essentially identical and revolved around the encoding of a ONE-OF constraint; where only one constraint out of a sequence of constraints can be true. For example, an incorrect version would be: $(f_i \equiv 1) \equiv (f_0 + f_1 \ldots + f_n)$ for some i and n. Thus, the error is that $f_i \equiv 1$ yields a Boolean constraint (i.e. $\equiv$ has type $\equiv : ar^{\mathbb{Z}} \to ar^{\mathbb{Z}} \to v^{\mathbb{Z}}$) but it is the left child of another $\equiv$ which expects an arithmetic expression as its left child not a $v^{\mathbb{Z}}$ expression. The correction is to repeat f_i and handle the boolean constraint correctly. For example the corrected version of the above formula is $(f_i \equiv 1) \wedge (f_i \equiv (f_0 + f_1 \ldots + f_n))$, where we separate out the left child from the summation but preserve the semantics of the ONE-OF constraint.

Fig. 6.6 displays the performance of VSMT as variants to solve increases. The resulting VPL$^{\mathbb{Z}}$ formula matched the number of choice and plain terms from the VSAT version. Similarly, the number of satisfiable variants matched the results for the *fin* dataset in Fig. 6.1. VSMT displays a speedup of 1.22x over the baseline *v→p* at 1,024 variants. There are two significant differences between the VSMT and VSAT results. First, the prototype SMT solver *does not* depend on the Haskell library `sbv`. Instead, VSMT utilizes a foreign-function interface to the C API of z3. Consequently, where `sbv` utilizes strings over `stdout` to communicate to the base solver, the ffi VSMT uses utilizes bytecode and hence has higher throughput. This has several implications, first, the range of results Fig. 6.6 are measured in seconds rather than minutes such as Fig. 6.2 and Fig. 6.3. Secondly, the overhead case $p{\rightarrow}v$ shows a speedup of 1.71x over the same baseline and is consistently faster than the variational case $v{\rightarrow}v$. There are several possible explanations but the exact reason is unclear.

We speculate on possible causes; the complete $p{\rightarrow}v$ algorithm is given below:

```
-- | Plain propositions on the variational solver testing the overhead of
-- accumulate/evaluate
pOnV :: [Plain Proposition] -> IO R.Result
pOnV = fmap mconcat . mapM (solve Nothing defSettings) . fmap unPlain
```

The algorithm is in the form for a classic optimization technique in functional programming called *map fusion*[5] and thus is likely to be optimized by the Haskell compiler while the other algorithms were not.

cite map fusion and deforesting and average speedup

$p{\rightarrow}v$ computes the variant by accumulation/evaluation, reducing the entire variant to a single symbolic and then issuing a CHECK-SAT call. Thus, any difference between $v{\rightarrow}v$ and $p{\rightarrow}v$ must come from choice removal. This is significant because this result may be a case where the extra work induced by the evaluation context in choice removal does not yield performance increases. To be specific, $v{\rightarrow}v$ constructs a context to efficiently reuse symbolic values, but if the SMT problem is very simple or if there exists a tautology or contradiction that is plain, then $v{\rightarrow}v$ will still construct and operate on this context even each variant does not need to be computed. Thus, it could be the case that for the majority of variants a contradiction or tautology occurred and was found by z3 before

[5] in practice fusion can lead to significant speedups ranging from 2x to an order of magnitude

the variation terms were considered (before the PUSH/POP calls) and thus any extra work to compute the result for the variant was redundant.

The exact reason for this result is unclear and more data is required to understand the prototype SMT implementation. Discovering the definitive reason is left to future work. If an early tautology or contradiction is the culprit then this could be detected in preliminary check. For example, one could replace each choice in the variational core with `T` and issue a CHECK-SAT to check that the core is satisfiable before computing the satisfiability of the variants. Similarly, if the Haskell compiler is optimizing $p{\rightarrow}v$ and not optimizing the other treatment conditions then this is an implementation detail which does not invalidate our methods for variational SAT and SMT solving. It also could be the case that particular sets SMT and SAT problems do not gain as much speedup from variational solving. For example, they might trigger heuristics in the base solver that simplify the problem space, and thus do not benefit from variational solving. Detecting such sets would require more data to understand the interaction between the variational SAT and SMT problems and the variational solver. However, the overall result: that $v{\rightarrow}v$ outperforms the baseline case $v{\rightarrow}p$, is further demonstration that our methods are effective for the three datasets we tested.

The last significant result is the magnitude of difference in the runtime between the variational SMT prototype and variational SAT prototype. Such a difference is to be expected given their implementation differences, but this difference indicates *other* domains where applying variation as a computational concept might be useful. As we have shown (perhaps unsurprisingly) the performance benefits from using concepts of variation are greater when the cost of a single transaction to the plain object language is high. For example, when the cost of emitting a single string to the base solver is high, e.g. in the prototype variational SAT solver, we observe a greater performance speedup (3.51x) than we the cost is low, e.g. in the prototype variational SMT solver (1.22x). Thus, other domains where a transaction cost is high are likely to benefit from research on variation. These domains might include network communication, where throughput and response times can be significant, or file systems and databases, where disk accesses are the limiting performance factor. In either case, this project successfully demonstrates performance speedups for two real-world cases in SAT and SMT solving by employing variational concepts.

Chapter 7: Related Work

We have succeeded in creating a variation-aware SAT and SMT solver by utilizing plain SAT and SMT solvers. This chapter situates this work in the larger research context. Section 7.1 discusses other satisfiability solvers that reuse information and provides a small history of incremental SAT solving. Section 7.2 discusses other methods to reason about variability in software product-lines and situates this work in that domain. Lastly, Section 7.3 compares our approach to other variation-aware systems that have been invented over the last decade.

7.1 Comparison to other solvers and execution models

This work is most similar to the Green solver by Visser et al. [129], which also constructs a SAT solver that exploits shared terms and prevents redundant computation. However, the projects differ in important ways. Visser et al.'s solver is oriented for program analysis and does not use incremental SAT solving. Rather, it employs heuristics to find canonical forms of sliced programs, and caches solver results on these canonical forms in a key-value store [82]. In contrast, variational SAT solving is domain agnostic, solves SAT problems expressed in VPL, returns a variational model, and uses incremental SAT solving.

It is also possible to view incremental SAT and SMT solvers and the incremental

SAT problem as variational systems and as a variational problem. Both are concerned with efficiently solving instances of problems which by definition share terms and are therefore related, and thus variational. We provide a small history of incremental SAT here as it is related work by being the target language of our compiler.

The incremental SAT problem was first defined by Hooker [63], with successive refinements of techniques by Hachemi Bennaceur [60], and with the assertion stack idea developed in Kim et al. [74]. The incremental SAT problem was devised as a solution to verification and optimization problems in electronic design automation such as covering problems [39], detecting delay faults[73], and model checking[35]. The first incremental solver to gain traction was `SATIRE` created by Whittemore et al. [133].

double check

Just two years later, Eén and Sörensson [46] made a major advance in incremental SAT with `MiniSat` by defining, documenting, and popularizing the implementation techniques required for an incremental SAT solver. `MiniSat` was the result of work on two other solver's called `SATZOO` and `SATNIK`, and seemed to hit a sweet spot in the design space. It simplified the existing notions of incrementality from the state of the art incremental solvers `SATIRE` and `PBS` [3] and combined propagation strategies from the `Chaff` [99] solver such as conflict-driven backtracking[138] and dynamic variable ordering [99]. These combinations lead to a solver that was performent, and whose implementation was small and communicative. That same year, the first SMTLIB standard would be proposed by Tinelli [125] although incremental SAT commands would not be incorporated until the 2.0 version[13] in 2010.

The use of choices in the variational solvers is similar to the concept of *facets* by [11] and *faceted execution* by [112, 98, 12], in that both choices and facets syntactically demarcate terms in an object language that must be specially handled, and yet must also operate with terms outside of the choice or facet. Facets are very similar to choices, facets use a label to determine branches (or alternatives in our language), facets are synchronized by these labels, facets are treated as tree-data structures, and facets are similarly treated as undetermined until they are reified.

Schmitz et al. [111] define the faceted secure execution framework `Multef`, which tries to avoid repeated execution of non-faceted values just as this work tries to gain performance through avoiding repeated execution of plain values. `Multref` does this by forking executions threads when a novel facet is encountered. This strategy avoids redundant execution before the facet is found but still has redundant or repeated computations inside the fork. In contrast, our methods of accumulation, evaluation, and utilization's of a zipper succeeds in only evaluating plain terms a single time and reusing that information across variants. The facets have been employed to policy-agnostic programming models and information flow control [109], thus our methods might leak too much information to be useful in that domain.

However, there are other striking similarities, Algehed et al. [2] improves the performance of `Multref` by defining rewrite rules which manipulate facets similarly to the equivalence laws presented for choices in Fig. 3.1c. For example, Algehed et al. removes redundant facets through a rewrite rule called `Choice Irrelevance`, which is isomorphic to the IDEMP rule in Fig. 3.1c. Another case is definition of *Squashes* which finds dead branches in nested facets. Squashes are similarly isomorphic to our discussion of *dominating choices* in Section 3.2.

7.2 Reasoning about Variability in SPL

Since SAT solving is so common in software variability applications, many strategies have been developed to reduce effort in this domain.

Similar to variational formulas, Nieke et al. [102] encode several versions of a feature model in a single formula. We reuse their benchmark as part of our evaluation as described in Section 6.1; a direct comparison with their approach is nuanced and discussed in Section 6.2. While their work focuses on feature-model analysis only, variational formulas and variational solving can be applied to many application areas.

In the context of family-based type checking [122], others have discussed merging multiple SAT problems into one. Most work in this area use a *local* approach where SAT problems are solved as they are encountered during typing; in contrast, *global* approaches collect SAT checks into a single problem that is solved at the end of the analysis. While the global approach improves efficiency by increasing reuse of learned clauses in the solver, it loses the ability to identify *which* variants contain type errors [5, 64]. Variational solving can achieve the reuse benefits of the global approach without sacrificing the precision of the local approach.

Since the size of SAT problems in software variability applications is often dominated by the feature model, researchers tried to reduce the size of satisfiability problems by delaying consideration of the feature model until after the analysis and only using it rule out false positives [22, 36, 86], a technique known as late feature-model consideration [122]. Bodden et al. [22] found that this technique increases the overall efficiency of static analysis [22], while Classen et al. [36] found that it actually decreases efficiency of family-based model checking. Variational solving is orthogonal to these approaches since the feature model can be excluded from a variational formula and then used later to rule out false positives.

Feature models can also be reduced in size to speed up analyses, for example, by slicing [1, 79] or decomposition [114]. It is largely unexplored how much such reductions can improve efficiency, but the analysis will still involve multiple similar SAT problems, which can benefit from variational solving.

A final approach is to avoid SAT problems by using modal implications graphs [80], which support faster reasoning. The idea is to encode as many software variability constraints as possible in such graphs, then use a SAT solver only for the remaining constraints. The construction of modal implication graphs already requires solving SAT problems, but this cost is amortized if many SAT queries will be solved during the analysis, as Krieter et al. [80] found for configuration processes.

Lastly, our idea of representing variation in a non-traditional formula (a VPL formula in our case) is similar to the approach by [90], which uses quantified boolean formulas to encode variation, and quantified boolean SAT solvers to detect anomalies in context-aware feature models. Notably, this approach has the benefit of avoiding incremental SAT solving altogether.

7.3 Variational or Variation-Aware Systems

Variational SAT solving is the latest in a line of work that uses the choice calculus to investigate variation as a computational phenomena. This body of work ranges from data structures, to graphics, to full fledged systems such as the system presented in this thesis. Due to the nature of variational problems, many variational or variation-aware systems employ SAT and SMT solvers. We collect and discuss these contributions here beginning with variational data structures.

There is relatively little work on variational data structures. Erwig et al. [52] describes a general strategy for constructing a variational data structure. Walkingshaw et al. [132] expands on this strategy and attempts to formalize ad-hoc implementations used in variational systems such as TypeChef [72] and SuperC [58]. For this section we focus on recent advancements implementing performent variational stacks and lists. The goal for variational data structures is to construct a data structure which describes a set of non-variational data structures efficiently. The variational artifact is the implementation of the variational data structure, and the variants in this domain are the plain versions of the data structure or plain values that result from operating on the data structure. The challenge then is to devise a variational data structure that describes and contains the variation, and provides a set of operations to manipulate the data structure that are as close to the performance of their plain counterparts as possible.

A fundamental tension in this domain is exemplified by work on variational stacks by Meng et al. [97]. Meng et al. identify two kinds of possible variational stacks depending on the location of variation on may have either: a stack of choices, or a choice of stacks. However, their analysis on a general implementation strategy was inconclusive, rather they found that depending on the implementation strategy runtime performance could be affected by as much as 20%. Furthermore, the variation in their experiment is coarse grained, i.e., the sharing ratio is high. Thus, Meng et al. utilized heuristics (optimizations in their paper) which further improved performance for both implementations by 43%. Utilizing heuristics was also found to be a successful strategy in Meinicke's PhD dissertation which we address below.

The work on variational stacks yields an alternative implementation strategy for variational SAT solvers. We have carefully designed our variational SAT and SMT solvers with a goal to utilize a plain base solver. We could have done otherwise and implemented a variational solver directly. With variational stacks the variational solver could utilize a variational assertion stack and we would avoid the need for a zipper in choice removal. Such an implementation is worth considering although by developing an independent solver we lose any benefits brought by the SAT/SMT communities and lose the general recipe for constructing a variation-aware system *using* its plain counterpart.

Similar to variational stacks, Smeltzer and Erwig [119] successfully implemented variational lists. Smeltzer and Erwig devise six implementations of variational lists with one implementation, the *suffix list* coming from previous work [51]. Smeltzer and Erwig's study leads to some surprising results. Out of their six implementations they found that for some implementations, simple functions such as `head` (which returns the first element of the list) are slower than the brute force counterpart, because the implementation may be required to traverse the whole list to resolve the variation. However, they do conclude that one implementation a *segment list* yields reasonable performance given the data in their study. The segment list is an interesting result as the idea behind the design is to encode variation as a *sequence of segments*, where a segment is either a choice or a sequence of plain elements. This idea should sound familiar as accumulation and symbolic values are essentially pointers to sequences of plain terms. Smeltzer and Erwig also observe that the sharing ratio has a measurable impact on performance (a finding we also observed) and thus minimizing or manipulating choices to increase the ratio is important; a result that has also been observed in software product-lines by Apel et al. [7] and Kästner et al. [71].

In addition to data structures there has been research on applications of the choice calculus to graphics [49],type systems [25, 26, 32, 31], and error messages [30, 28, 31, 29]. For the remainder of this section we focus on variational or variation-aware systems.

This work is not the first to construct a variational or variation-aware system. Notably, Kenner et al. [72] used the choice calculus and variational data structures to type check every possible Linux kernel. Thus constructing a variational parsing [69], a variational lexer [81], type system[86, 70] and control-flow and data-flow analyses[86]. Similarly, Gazzillo and Grimm [58] variationally parse the Linux kernel by utilizing variational data structures and *choice nodes* in the abstract syntax tree. TypeChef is notable for several reasons: its implementation is a direct inspiration for our baseline algorithm $v{\rightarrow}p$ which uses an incremental SAT solver but only exhibits sharing *before* a choice is discovered. This kind of sharing, called *prefix sharing* by Smeltzer and Erwig [119] is the de-facto standard in software product-line applications which employ incremental SAT solvers. Given the results of this thesis, large performance gains are possible if our results are Representative with the use of a variational SAT or SMT solver. TypeChef is also notable for its two step approach, first it parses source code to find `#if-def` annotations, and stores these in files called *presence conditions*. Presence conditions are isomorphic to variation contexts, both are C_2 formula's over dimensions (or conditions of the `#if-def`) which determine a variant. Using the presence conditions, TypeChef annotates choice nodes to determine which variant the leaves of the choice node belong. Then TypeChef extends the symbol table of a C program to contain types which are conditional based on the presence conditions. This allows a variable's type to change from one variant to another. Each type checking operation is then lifted to handle the variational cases and then type checking checks the variation-aware types to ensure every each variant type checks. Similar to our use of variation contexts, TypeChef allows a *variability model* which specifies variants that should be type checked by conjoining the model with the presence conditions.

In his PhD dissertation, Meinicke [94, 95] constructs a variational interpreter called VarexJ and a variational bytecode transformer called VarexC, to achieve a variational execution and debugging framework. The framework tries to maximize sharing in two ways: It directly utilizes the choice calculus to represent local points of variation and achieve a *fine-grained approach*, this allows the framework to share program states and keep a unified heap. Additionally, the framework achieves instruction-level sharing among control-flows between variants. It achieves this by implementing a variational scheduler, which seeks to order the execution of program statements to optimize sharing. We achieve this same effect through the interaction between accumulation, evaluation and choice removal with the wrapped primitive operations. Interestingly, Meinicke identifies redundant SAT calls as a major bottleneck in the variational execution framework. Specifically they determine redundant CHECK-SAT calls as the most expensive. To reduce the redundant calls the variational execution framework caches calls to the solver thus only employing the solver for new queries. Chu-Pan Wong used VarexJ to do speculative mutation testing and automated program repair [136].

Lastly, choice calculus has been successfully applied to databases to construct a complete approach for variational databases including a variational database management system, a variational query language, and variational tables. Ataei et al. [9, 10] constructs a variational database in an interesting way. Ataei et al. add choices to relational algebra to define a variational query language for a variational relational database. The variational query language serves as the variational artifact similar to the role of VPL in the variational SAT and SMT solvers. Ataei et al. specifically choose to avoid adding choices to variational tables, instead opting to apply *annotations* to the table schema, table attributes, and table tuples. The annotations are C_2 formulas and are derived from the dimensions of choices in the variational query language. Annotations that are attached to an aspect of the database, such as a schema, attribute or tuple are called *presence conditions* following the work on TypeChef. Annotations which are not attached but describe possible variants are called *feature expressions*.

This careful design has several desirable properties: the separation between the variational aspects of the system and the database engine allows the database engine to remain mostly unchanged. Thus, Ataei et al. avoid implementing low level details such as a variational `B+-Tree` or file system. However the system is still memory efficient: Elements which are shared between variants are represented a single time in the database. To realize an element is shared, a SAT solver is called on the presence conditions for that element. For example, imagine an attribute that belongs to two variants A and B, to encode that this attribute belongs to these two variants is expressed in the presence condition as a disjunction, $A \vee B$. Thus, Ataei et al.'s system is a mixed approach; the query language embeds choices to explicitly represent local points of variation. The underlying object language (the database in this case) lacks a primitive operation to handle variation such as the PUSH/POP commands in our work. Hence, Ataei et al. choose to *realize* variation in the database through indirection based on annotations and SAT solving. Thereby enabling a full fledged variational database without requiring substantive changes to the entire database implementation. Therefore, Ataei et al.'s system is more expressive than the variational SAT and SMT solver's presented in this thesis because it can express dependencies between variants through presence conditions, while our approach is limited to express dependencies by nesting choices.

Notably, our method for variational SAT and SMT solving *requires* that all points of variation are known *before* the run-time of the solver. This is a direct consequence of VPL, by construction one can only make a VPL formula if a point of variation is known. If one does not know, or needs to discover the points of variation during runtime then the VPL formula cannot be constructed. This limitation is a significant difference from the aforementioned variational systems. We return to this point in Section 8.2, but using variational SAT solvers effectively in these domains is an open research question.

7.4 Possible Applications of Variational SAT solving

Variational SAT and SMT solving provides an improved user interface and possible performance gains for variational SAT and SMT problems. However, the space of variational SAT and SMT problems is largely unexplored, as viewing problems as inherently *variational* is only just beginning to gain awareness outside of the software product-line community. In this section we describe areas for possible applications.

Thüm et al. [124] define two fundamental dimensions of variation: variation in *time*, where software is revised over some unit of time with the intent that the new version will replace the old version; and variation in *space* where variants are meant to co-exist simultaneously. Our approach to variational SAT and SMT solving is able to express both kinds of variation with the caveat that all points of variation are known *before* running the solver. Thus, applications that utilize a plain SAT solver, do not need to discover variation during run-time and that must negotiate variation in time or space are possible applications for variational SAT or SMT solver.

Problems in this domain include scheduling problems [21] which need to account for a counterfactual event; for example, one might need to schedule a set of jobs on a number of machines but also account for one or several machines being unable to take jobs. Such a problem is directly expressible in VPL where each dimensions would correspond to a machine being online, or a machine being disabled. Another classic SAT application is circuit layout and hardware verification problems [21]. In this domain, SAT solvers are used as the back-end engine to answer safety and live-ness questions. Questions such as a given system can never reach a certain state or a system will always reach some given state after a certain state is reached [21]. This work could be directly applied to such problems; for example one might have two or more circuits which share significant regions and yet are distinct products with distinct behavior. Performing hardware verification on each circuit would produce two related SAT problems, where the shared portions are redundantly calculated. Thus, one can imagine translating the set of SAT problems to a VPL formula and solving them with a variational solver. Another direct application would be performing hardware verification in the presence of patches, one might encode speculative analyses to ensure desirable properties in the hardware if regions or elements in the circuit are completely removed, significantly patched, or stop operating. The particulars in this domain are open research questions, however given the findings in this thesis large performance gains are possible through the use of a variational SAT or SMT solver.

Software variability is a natural application domain for this work. The variability of SPLs or configurable software is often reduced to propositional logic [16, 42, 96] for analysis purposes [20, 122, 55]. Many analyses have been implemented using SAT solving such as [122], including feature-model analysis [20, 55], parsing [69], dead-code analysis [120], code simplification [130], type checking [121], consistency checking [41], dataflow analysis [86], model checking [36], variability-aware execution [101], testing [27], product sampling [93, 128], product configuration [110], optimization of non-functional properties [116], and variant-preserving refactoring [53]. While each of these analyses gives rise to multiple SAT problems for even a single analysis run, the authors typically do not discuss how they are solved. We argue that many could benefit from variational solving.

More generally, any scenario that involves solving many related SAT problems, and where all of these problems are known or can be generated in advance, is a potential application for variational SAT solving. Such situations arise in program analysis [129], and especially in *speculative* program analyses that involve generating and exploring huge numbers of variations of a program, for example, as in counterfactual [28] and migrational [26, 25] typing. Furthermore, we believe that variational solving could provide a basis for similar speculative analyses on feature models.

Chapter 8: Conclusion

This thesis has presented variational satisfiability and satisfiability-modulo theory solving. In Chapter 1 we defined the success of this thesis as applying the concept of variation in the domain of satisfiability solving to create a variational satisfiability solver. The solver must explicitly express the concept of variation in a user-facing language and must be performent with respect to the performance of plain satisfiability solvers. By this definition we have succeeded in demonstrating these ideas work in practice in the domain of satisfiability solving. We have not only shown that through the application of the choice calculus variation can be directly expressed by the end-user, but also that runtime performance may be improved because local points of variation are made explicit. To conclude the thesis we review the important contributions in Section 8.1. Section 8.2 provides immediate directions for future work.

make sure this was defined in intro

8.1 Summary of Contributions

The main contribution of this work is the formalization of variational satisfiability solver.

In Chapter 3 we formalized a many-valued logic to express variational SAT problems, demonstrated an application of the choice calculus with propositional logic as the object language. We defined the denotational semantics of configuration over the logic, and fundamental concepts such as variants and synchronization.

In Chapter 4 we formalized our approach to variational satisfiability solving based on this logic. Our approach is to variationalize non-variational solvers by constructing a compiler to a standardized input format. We saw that this approach has many desirable properties: The stages of accumulation, evaluation, and choice removal cleanly separate concerns. The sharing of plain terms is guaranteed between variants because we use a zipper to capture evaluation contexts. Since our design integrates plain base solvers, our variational solver can take advantage of advances made by the SAT and SMT communities. Lastly, we proved that our design is confluent, thereby showing that the variational solver is variation-preserving and thus sound.

In Chapter 5 we extended the architecture to handle non-Boolean constraints. We saw that extensions over the term language follow a pattern: One wraps the primitive base solver operations to handle symbolic values, then defines a congruence rule to process the recur on the left child of the relation, and finally defines a computation rule that calls the wrapped primitive to combine two symbolic values thereby producing a fold over the relation. We presented two extensions; one over integer constraints, and one over array based constraints. Since, symbolic values are untyped, we carefully constructed the extended logic to make type errors inexpressible, we could have otherwise chose to employ a simple type system as the SMTLIB2 standard does. Lastly, we saw that this extension pattern works even for background theories that seem difficult, because our architecture processes plain terms before variational terms due to the ordering between accumulation and choice removal.

In Chapter 6, we built two prototype variational solvers called VSAT and VSMT. We evaluated the solvers over two real-world datasets. We observed that variational solving does produce speedups over standard use of an incremental solver when solving many variants. The variational solvers produce this speedup by reusing shared terms and avoiding redundant computation. We observed that the base solver does have an impact on runtime performance. Therefore an advantage of our architecture is that it is base solver agnostic, and thus implementations may choose whichever solver is performent for its problem domain as long as the solver accepts the SMTLIB2 standard. However, we found that outside of its use case—when solving only a single variant—variational solving does show a performance overhead that was statistically significant for one dataset. Lastly, our finding that the sharing ratio is positively correlated to runtime performance repeats similar findings in the variational literature.

8.2 Future Work

There are numerous avenues of future work ranging from novel applications, to refining the implementations, to extended the solvers with new features. In this section we collect and discuss the most promising future work beginning with tool extensions and ending with abstracting this work to domains other than satisfiability solving.

8.2.1 Utilization of variational cores

Variational cores are an important and foundational concept for the variational solver's and consequently for the variationalization recipe. Recall that the purpose of variational cores was twofold: First, to condense the query formula such that the variational terms were the majority of terms core. Second, to simplify the choice removal process by reducing the amount of traversal required to process the choices. Third, to enforce sharing between variants as the contexts captured by the core were are reused during choice removal.

This last point is key, because variational cores in combination with the accumulation and evaluation stores, completely capture the context of a formula they can be reused in novel ways. For example, one might serialize a variational core and associated stores to disk, thus effectively caching the core for future use. Such a feature would enable desirable user facing effects: the solver could restart without losing information and thus might be useful for debugging or exploration, if the variational cores require a lot of processing time to generate this time be amortized, or if the application domain only builds on previous versions of the same formulas, then the variational core could be reused.

For example, consider the case of a feature model which evolves every month for several months, similarly to the *fin* and *auto* datasets. Since the feature model, and consequently the VPL formula evolves over time, the previous variational core could be modified to reflect the changes for the new formula. Adding new constraints is straightforward; one would simply nest the previous variational core in a conjunction context with the new core and reuse the previous stores when generating the new core to ensure sharing. A more difficult problem is removing constraints or variables in the previous core. Both removing constraints and removing variables is problematic as the variable or constraint could have been accumulated into a symbol value or several symbolic values. One could traverse a dependency graph to find all references of the variable and symbolic value, and then seek to replace those references with a unit value, such as $\mathtt{T}$ for $\wedge$ or $\mathtt{F}$ for $\vee$. However this immediately leads to the problematic case where the variable or symbolic to be removed is in a $\neg$ context. There is no unit value where $\neg$ does not have meaning and thus we cannot remove arbitrary variables from a variational core.

In addition to manipulating or storing variational cores, future variational solvers might utilize them as a convenient messaging format. Throughout this thesis, we have assumed and have only considered systems which process all variants in a single base solver instance, however this need not be the case. Instead, when a choice in is the focus of a the evaluation context (and thus the variational core) one might choose to solve the true alternative variants in a different solver and all the false alternatives in the same solver. For example, a user might know that all true alternative variants have particularly good performance characteristics for boolector, while all false variants have good characteristics for yices. Since we compile to SMTLIB2 script such a feature is possible with few changes to our method of variational solving. To add such a feature a future variational solver would allow the user to select particular solvers over the input vc or the configuration for a query formula.

8.2.2 Further SMT background theories and tool extensions

SAT and SMT solvers are attractive targets for research on variational languages. As of this writing, designing a language with variational side-effects is an open research question. The essential problem is tracking effects for particular variants across the interface between a variational-aware system and a plain system. For example, imagine writing a file to disk in one variant and deleting a different file in another variant. Since the file system has no concept of variation or variant the variational system is not able to guarantee variants are isolated and therefore variants may interact in undesirable and difficult to predict ways. SAT and SMT solvers side step this limitation as they are side-effect free systems. There is simply no way to read a file disc in an SMTLIB2 script. Similarly, classes of run-time errors are not possible. For example, consider an SMTLIB2 script which divides by zero, in this case the script simplify will not unify and an UNSAT will be returned.

Due to the attractive properties of SAT and SMT solvers for variational research, a straightforward avenue of future work is to continue to investigate efficient variational folds by further extending the variational solvers. Modern SAT and SMT solvers allow quantified constraints following first-order logic. In this thesis, we have only considered unquantified constraints, and thus the interaction of between quantified constraints and choices is an open research problem.

Similarly, we have demonstrated extensions for core background theories, but there are many features of plain solvers that would be desirable additions to variational solvers. Such features include generation of variational unsatisfiable-cores. An unsatisfiable core is a subset of constraints that prevent the SAT or SMT from unifying. Unsatisfiable cores are thus desirable for many problems where discovering this information is desirable. For example, one might desire to find the clique in a SAT encoded weighted graph which prevents a traversal under some cost limit. Or one might desire to find the sub-set of features in a feature model that prevent classes of products from being built.

Enabling variational unsatisfiable cores is possible with our approach of accumulation, evaluation and choice removal. The key requirement would be to ensure that the plain, GET-UNSAT-CORE command occurs inside the PUSH/POP block for a given variant. Thus far we have only seen the GET-MODEL command have this property. Thus a straightforward approach would be to create a syntactic category that contains arbitrary plain commands, such as GET-MODEL or GET-UNSAT-CORE to be sent to the base solver once a variant has been reduced to $\bullet$. Another approach is to create full fledged variational SMTLIB2 language instead of expressions of constraints as we have presented here. Constructing such a variational SMTLIB2 language is likely to save work for future extensions. The language would be identical to SMTLIB2 except that PUSH/POP would not be exposed to the user (or would only be enabled with an option), and choices would be included in the language just as we have included the for VPL and VPL$^{\mathbb{Z}}$.

Lastly, a promising area of future work is constructing an asynchronous variational SAT and SMT solver. During our experience bench marking the variational prototype solvers we found that the majority of the time spent in the base solver is spent querying for a model. Furthermore, each variant waits until they can be processed by the base solver. For example, consider the formula $A\langle a, b\rangle \wedge B\langle c, d\rangle$, solving this formula results in solving all four variants. Our prototype solvers choose true alternatives first, thus the ordering of the variants in the base solver will be $\{(A, \texttt{T}), B, \texttt{T}\}$, $\{(A, \texttt{T}), (B, \texttt{F})\}$, $\{(A, \texttt{F}), (B, \texttt{T})\}$, $\{(A, \texttt{F}), (B, \texttt{F})\}$. Notice that all $(A, \texttt{F})$ variants wait for $(A, \texttt{T})$ variants before being considered. Thus, instead of using PUSH and POP to represent variation we could instead fork a new solver thread and solve all $(A, \texttt{F})$ variants on that solver thread.

We have created three versions of asynchronous prototype solvers but have not succeeded in constructing a general sound asynchronous variational solver. Constructing an asynchronous solver is relatively straightforward. Since variational models form monoids, the order in which models are added to the variational model isn't important. Similarly, since variational cores capture the evaluation context at a given time transmitting variational cores to other solver instances is also straightforward. The problem for asynchronous solvers is ensuring that the ordering of alternatives is maintained. For example, a simple model might be to have a pool of producers, which derive variational cores, and a pool of consumers, which take a variational core and a configuration and find the next choice that is not in the configuration or generate a model. The two pool model is sufficient in most cases, however a subtle bug is now possible. Assume we have a formula with three unique dimensions A, B, and C which will be processed in that order. Since the order of alternatives is no longer deterministic we might encounter a case where we are either stuck or have mixed variants. For example, assume we have an unbalanced number of consumer's and producers. Now consider the case where a consumer thread has consumed $\{(A, \mathtt{T}), (B, \mathtt{F})\}$, then finds a choice with a C dimension and waits for a request from a producer thread. The consumer observes a request to consume $\{(C, \mathtt{T})\}$, does so, and produces a model for that variant. Now, the consumer will backtrack with a POP call and wait for another request from a producer for another C. However, this thread may have out paced other threads, and so the only request from the producer pool is to consume $\{(B, \mathtt{T})\}$, and now we are stuck. If the consumer accepts the request we will have mixed two variants on this thread yielding incorrect results, if the consumer does not take the request then we may end in a deadlock. Such an example is contrived but occurs with asynchronous communication and must be properly handled. The correct method is for each thread to track which variant it has solved and track the ordering of choices. We must ensure that the choices are solved in order such that if a request comes to solve a $\{(A, \mathtt{T})\}$ variant, and the thread has consumed the variational core with $\{(A, \mathtt{F})\}$ then the thread must issue as many POPS as needed to backtrack. By tracking this information we can avoid deadlocks, and malformed variants and still gain the benefits of concurrent solving which could be substantial especially for large variational formulas.

8.2.3 Automated VPL formulas

Thus far we have only considered a VPL or VPL$^{\mathbb{Z}}$ formula as input to a variational solver. This format is likely to be inconvienient as end-users consider sets of SAT problems. Thus, a useful extension for these users is to change the input from a VPL formula to a set of SAT problems the user is interested in. With the set of SAT problems, one could synthesize a VPL formula with a sharing ratio that is *good enough* and then run the solver on that VPL formula. For the rest of this section, we'll refer to the problem of synthesizing a *good* VPL formula from a set of SAT formulas the *synthesis problem*.

There are several considerations to highlight. First, we found that the sharing ratio of a formula positively correlates to run-time performance in Chapter 6, echoing results from previous research on variation. Therefore, the synthesis algorithm should try to maximize the sharing ratio as it chooses which variants to combine in a choice. Second, minimizing the number of choices is high priority for the algorithm. Our results indicate that the run-time of the variational solver grows linearly in the number of variants to solve (hence exponentially in the number of unique dimensions), thus adding a single new choice doubles the number of variants and the expected run-time. Rather than provide an algorithm that find the *best* VPL formula, we instead describe a greedy algorithm that tries to find a reasonable VPL formula. An algorithm that finds the *best* VPL formula, e.g. one which maximizes the sharing ratio while minimizing the number of choices is an open research problem. We suspect it is at least NP-hard (likely by demonstrating that the Binary Decision Diagram variable ordering problem karp reduces to the VPL synthesis problem), although we have not begun to investigate the problem space.

The synthesis problem is a search problem over a total un-directed graph of possible formula combinations. Each node in the graph is a SAT formula or VPL formula that can be combined and is connected to every other node. Edges represent the possible combinations of two nodes and is weighted with a *fitness metric* indicating a good match (and thus high amount of sharing between two nodes) or a bad match. Our approach is to traverse the graph and greedily select the best combination between two nodes. Combinations mutate the graph. The old nodes are replaced with the combined node, all old edges are relaxed and new edges connect the combined node to every other node in the graph. The algorithm then repeats until only a single node remains in the graph.

There are two sub-procedures in the algorithm: A procedure to combine nodes, and a procedure to calculate the fitness metric between two nodes. To combine two nodes we generate a unique dimension, nest one node in the true alternative, and the other in the false alternative. This simple combination procedure results in poor sharing as the choice is always be at the root of the abstract syntax tree. Thus to increase the sharing ratio, we attempt to drive the choice towards the leaves of the abstract syntax tree of the VPL formula using the equivalency laws in in Fig. 3.1c.

Next we need a procedure that inputs two SAT or VPL formulas, and returns a fitness metric. There are several possible algorithms; ranging from string edit distance, to a tree edit distance over the abstract syntax trees of the SAT or VPL formulas. String comparison algorithms such Levenshtein distance[84] or Hamming distance [61] are promising as both have implementations which run in polynomial time, assuming an encoding from the SAT problems to strings is computationally feasible. Graph edit distance is a more direct approach but is NP-Complete with an approximate solution that is APX-hard [87]. However, most edit distance algorithms work well in practice, and it is likely that the graph comparisons in this domain are simpler than comparisons which occur in the worst case, e.g., over enormous graphs such as those found in social networks. Furthermore there are many heuristics such as longest-common sub-string which might produce metrics that are good enough for reasonable sharing ratios.

8.2.4 Abstracting the variationalization recipe to other domains

Our approach to creating a variation-aware system by using the plain version of that system is not specific to satisfiability solvers. The only portion of our work that is particular to satisfiability solvers is code generation in the base solver. In essence, our method is a variational left-fold over a variational language. Thus, one might reuse the ideas of accumulation, evaluation, choice removal and variational cores in other domains. In particular, the recipe for variationalization of other domains is clear: To variationalize a plain system one needs to define the variational artifact for the domain and a method to express variation in that system; our variational artifact was VPL and we chose to use scopes from the SMTLIB2 standard, although forking solver instances is also feasible. One needs a method to express segments of plain terms and preserve sharing between variants in the plain system, our approach was to define symbolic values and utilize the internal cache of the plain solvers to preserve sharing. Lastly, one needs a way to retrieve results from the plain system and combine those results in any order, hence we defined monoidal variational models.

Using this recipe one can might imagine a variational prolog which reuses the work presented in this thesis. For such a language, the variational artifact would be a prolog-like programming language with choices. Expressing segments of plain terms with symbolic values could be directly reused from this thesis. Similarly, the variational result would be nearly identical to the variational models presented in Section 4.4. Embedding variation in prolog is the difficult part although there are several possibilities. SWI-prolog [134] defines a special kind of predicate called *dynamic predicates*. Dynamic predicates indicate to the prolog interpreter that the predicate may change during execution. Changing the predicate during execution is performed using two primitives, *assertz* and *retract*. Thus prolog defines a way to assert a constraint in the interpreter and then refine the constraint as needed, and so dynamic predicates may serve as a viable primitive for variation in the prolog interpreter. Another promising embedding is using delimited continuations. In Chapter 4 we hypothesized that because a Heut zipper is used for choice removal, using delimited continuations is also feasible as Huet zippers have been shown to be isomorphic to delimited continuation [75]. Fortunately prolog has first class support for delimited continuations [113] and thus choice removal could be done in the base prolog interpreter rather than at the variational level. Using delimited continuations could greatly reducing the complexity of creating a variational prolog, and it might be possible to define variational prolog as a library rather than a separate entity. The exact details for a variational implementation are not clear but creating a variational prolog is certainly feasible.

Bibliography

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing Feature Models. pages 424–427, 2011.

[2] Maximilian Algehed, Alejandro Russo, and Cormac Flanagan. Optimising faceted secure multi-execution. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 1–115, 2019. doi: 10.1109/CSF.2019.00008.

[3] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ilp versus specialized 0-1 ilp: An update. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '02, page 450–457, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 0780376072. doi: 10.1145/774572.774638. URL https://doi.org/10.1145/774572.774638.

[4] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. Implicit Constraints in Partial Feature Models. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 18–27, 2016.

[5] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Language-Independent Reference Checking in Software Product Lines. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 65–71, 2010.

[6] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. 2013.

[7] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. pages 482–491, 2013.

[8] Joe Armstrong. Making reliable distributed system in the presence of software errors. Website, 2003. Available online at http://www.erlang.org/download/armstrong_thesis_2003.pdf; visited on March 16th, 2021.

[9] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Variational Databases. In *Int. Symp. on Database Programming Languages (DBPL)*, pages 11:1–11:4. ACM, 2017.

[10] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Managing Structurally Heterogeneous Databases in Software Product Lines. In *VLDB Workshop: Polystores and Other Systems for Heterogeneous Data (Poly)*, 2018.

[11] Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 165–178, 2012.

[12] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321440. doi: 10.1145/2465106.2465121. URL https://doi.org/10.1145/2465106.2465121.

[13] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at www.SMT-LIB.org.

[14] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.

[15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[16] Don Batory. Feature Models, Grammars, and Propositional Formulas. pages 7–20, 2005.

[17] Roberto J. Bayardo and Robert C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 203–208. AAAI Press, 1997. ISBN 0262510952.

[18] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Automated Reasoning on Feature Models. pages 491–503, 2005.

[19] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. A First Step Towards a Framework for the Automated Analysis of Feature Models. pages 39–47, 2006.

[20] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35 (6):615–708, 2010.

[21] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[22] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. pages 355–364, 2013.

[23] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009. doi: 10.1007/978-3-642-00768-2_16. URL https://doi.org/10.1007/978-3-642-00768-2_16.

[24] Gianpiero Cabodi, Luciano Lavagno, Marco Murciano, Alex Kondratyev, and Yosinori Watanabe. Speeding-up heuristic allocation, scheduling and binding with sat-based abstraction/refinement techniques. *ACM Trans. Des. Autom. Electron. Syst.*, 15(2), March 2010. ISSN 1084-4309. doi: 10.1145/1698759.1698762. URL https://doi.org/10.1145/1698759.1698762.

[25] John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating Gradual Types. *Proc. of the ACM on Programming Languages (PACMPL)* issue *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*, 2:15:1–15:29, 2018.

[26] John Peter Campora III, Sheng Chen, and Eric Walkingshaw. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *Proc. of the ACM on Programming Languages (PACMPL)* issue *ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, 2:98:1–98:30, 2018.

[27] Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. 56(10):1183–1199, 2014.

[28] S. Chen and M. Erwig. Counter-Factual Typing for Debugging Type Errors. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 583–594, 2014.

[29] S. Chen, M. Erwig, and K. Smeltzer. Let's Hear Both Sides: On Combining Type-Error Reporting Tools. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 145–152, 2014.

[30] S. Chen, M. Erwig, and K. Smeltzer. Exploiting Diversity in Type Checkers for Better Error Messages. *Journal of Visual Languages and Computing*, 39:10–21, 2017.

[31] Sheng Chen, Martin Erwig, and Eric Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 29–40, 2012.

[32] Sheng Chen, Martin Erwig, and Eric Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems*, 36 (1):1:1–1:54, 2014.

[33] Sheng Chen, Martin Erwig, and Eric Walkingshaw. A Calculus for Variational Programming. In *European Conf. on Object-Oriented Programming (ECOOP)*, volume 56 of *LIPIcs*, pages 6:1–6:26, 2016.

[34] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132026. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266.

[35] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986. ISSN 0164-0925. doi: 10.1145/5397.5399. URL http://doi.acm.org/10.1145/5397.5399.

[36] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. on Software Engineering*, 39(8):1069–1089, 2013. ISSN 0098-5589.

[37] The Kernel Development Community. KConfig Language. Website, 2018. Available online at https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html; visited on March 13th, 2020.

[38] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. ISBN 9781450374644. doi: 10.1145/800157.805047. URL https://doi.org/10.1145/800157.805047.

[39] Olivier Coudert and Jean Chritophe Madre. New ideas for solving covering problems. In *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*, DAC '95, page 641–646, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917251. doi: 10.1145/217474.217603. URL https://doi.org/10.1145/217474.217603.

[40] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. 2000.

[41] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*, pages 211–220, 2006.

[42] Krzysztof Czarnecki and Andrzej Wąsowski. Feature Diagrams and Logics: There and Back Again. pages 23–34, 2007.

[43] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[44] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[45] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[46] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24605-3.

[47] Niklas Een, Alan Mishchenko, and Nina Amla. A single-instance incremental sat formulation of proof- and counterexample-based abstraction.

[48] Levent Erkok. SBV: SMT Based Verification: Symbolic Haskell theorem prover using SMT solving. Website, 2011. Available online at https://hackage.haskell.org/package/sbv-8.10; visited on Feb 14th, 2020.

[49] M. Erwig and K. Smeltzer. Variational Pictures. In *Int. Conf. on the Theory and Application of Diagrams*, LNAI 10871, pages 55–70, 2018.

[50] Martin Erwig and Eric Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011.

[51] Martin Erwig and Eric Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering IV (GTTSE 2011), Revised and Extended Papers*, volume 7680 of *LNCS*, pages 55–99, 2013.

[52] Martin Erwig, Eric Walkingshaw, and Sheng Chen. An Abstract Representation of Variational Graphs. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2013.

[53] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. pages 316–326, 2017.

[54] Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying smt in symbolic execution of microcode. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, page 121–128, Austin, Texas, 2010. FMCAD Inc.

[55] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models: Quo Vadis? *Computing*, 101(5):387–433, 2019.

[56] Vijay Ganesh, Charles W. O'Donnell, Mate Soos, Srinivas Devadas, Martin C. Rinard, and Armando Solar-Lezama. Lynx: A programmatic sat solver for the rna-folding problem. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, page 143–156, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 9783642316111. doi: 10.1007/978-3-642-31612-8_12. URL https://doi.org/10.1007/978-3-642-31612-8_12.

[57] James Garson. Modal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2018 edition, 2018.

[58] Paul Gazzillo and Robert Grimm. SuperC: Parsing All of C by Taming the Preprocessor. pages 323–334, 2012.

[59] Ian P. Gent and Toby Walsh. The sat phase transition. In *In Proc. ECAI-94*, pages 105–109, 1994.

[60] Gerard Plateau Hachemi Bennaceur, Idir Gouachi. An incremental branch-and-bound method for the satisfiability problem. volume 10, pages 301–208, 08 1998.

[61] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[62] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. ISSN 03036898, 14679469. URL http://www.jstor.org/stable/4615733.

[63] John N. Hooker. Solving the incremental satisfiability problem, Jan 1993. URL https://kilthub.cmu.edu/articles/journal_contribution/Solving_the_Incremental_Satisfiability_Problem/6708044/1.

[64] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically Safe Program Generation with SafeGen. *Science of Computer Programming*, 76(5):376–391, 2011.

[65] Spencer Hubbard and Eric Walkingshaw. Formula Choice Calculus. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 49–57. ACM, 2016.

[66] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992. ISSN 0362-1340. doi: 10.1145/130697.130699. URL http://doi.acm.org/10.1145/130697.130699.

[67] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997. doi: 10.1017/S0956796897002864.

[68] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[69] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. pages 805–824, 2011.

[70] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A Variability-Aware Module System. pages 773–792, 2012.

[71] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward Variability-Aware Testing. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 1–8, 2012.

[72] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 25–32, 2010.

[73] Joonyoung Kim, Jesse Whittemore, João P. Marques-Silva, and Karem Sakallah. On applying incremental satisfiability to delay fault testing. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE ’00, page 380–384, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132441. doi: 10.1145/343647.343801. URL https://doi.org/10.1145/343647.343801.

[74] Joonyoung Kim, Jesse P. Whittemore, Jõao Marques-Silva, and Karem A. Sakallah. On Solving Stack-Based Incremental Satisfiability Problems. In *Proc. of IEEE International Conference on Computer Design (ICCD)*, pages 379–382, Austin, Texas, September 2000.

[75] Oleg Kiselyov. Generic Zipper: the context of a traversal. http://okmij.org/ftp/continuations/zipper.html. Accessed at May 19, 2021.

[76] Stephen Cole Kleene. *Introduction to metamathematics*. Ishi Press, 1968.

[77] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. Technical Report 2016-01, TU Braunschweig, 2016.

[78] Robert A. Kowalski. The early years of logic programming. *Commun. ACM*, 31 (1):38–43, January 1988. ISSN 0001-0782. doi: 10.1145/35043.35046. URL https://doi.org/10.1145/35043.35046.

[79] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. Comparing Algorithms for Efficient Feature-Model Slicing. pages 60–64, 2016.

[80] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating Configuration Decisions with Modal Implication Graphs. pages 898–909, 2018.

[81] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. Partial preprocessing c code for variability analysis. In *In Proc. 5th ACM Workshop on Variability Modeling of Software-Intensive Systems*, pages 127–136, 2011.

[82] Redis Labs. Redis. https://redis.io/, 2020. Accessed at May 4th, 2020.

[83] Daniel Le Berre and Anne Parrain. The Sat4j Library, Release 2.2. 7(2-3):59–64, 2010.

[84] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[85] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Int. Conf. on Aspect-Oriented Software Development*, pages 191–202, 2011.

[86] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. pages 81–91, 2013.

[87] Chih-Long Lin. Hardness of approximating graph transformation problem. In Ding-Zhu Du and Xiang-Sun Zhang, editors, *Algorithms and Computation*, pages 74–82, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48653-4.

[88] Inês Lynce and João Marques-Silva. Sat in bioinformatics: Making the case with haplotype inference. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, page 136–141, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540372067. doi: 10.1007/11814948_16. URL https://doi.org/10.1007/11814948_16.

[89] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, May 1999. ISSN 0018-9340. doi: 10.1109/12.769433. URL https://doi.org/10.1109/12.769433.

[90] Jacopo Mauro. Anomaly detection in context-aware feature models. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450388245. doi: 10.1145/3442391.3442405. URL https://doi.org/10.1145/3442391.3442405.

[91] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. Anomaly Detection and Explanation in Context-Aware Software Product Lines. pages 18–21, 2017.

[92] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL https://doi.org/10.1145/367177.367199.

[93] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. pages 643–654, 2016.

[94] Jens Meinicke. VarexJ: A Variability-Aware Interpreter for Java Applications. Master's thesis, University of Magdeburg, 2014.

[95] Jens Meinicke. *Variational Debugging: Understanding Differences among Executions*. PhD dissertation, University of Magdeburg, 2019.

[96] Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald Cowan. Efficient Compilation Techniques for Large Scale Feature Models. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*, pages 13–22, 2008.

[97] Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Int. Work. on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 28–35. ACM, 2017.

[98] K. Micinski, D. Darais, and T. Gilray. Abstracting faceted execution. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 184–198, 2020. doi: 10.1109/CSF49147.2020.00021.

[99] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2. doi: 10.1145/378239.379017. URL http://doi.acm.org/10.1145/378239.379017.

[100] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately incremental sat. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 206–218, Cham, 2014. Springer International Publishing. ISBN 978-3-319-09284-3.

[101] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. pages 907–918, 2014.

[102] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. Anomaly Analyses for Feature-Model Evolution. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*, pages 188–201, 2018.

[103] National Institute of Standards and Technology. NIST e-Handbook of Statistical Methods. https://www.itl.nist.gov/div898/handbook/index.htm, 2020. Accessed at May 7th, 2020.

[104] Bryan O'Sullivan. Criterian: A Haskell microbenchmarking library. Website, 2009. Available online at https://hackage.haskell.org/package/gauge-0.2.5; visited on May 7th, 2020.

[105] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. 2005.

[106] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020. URL https://www.R-project.org/.

[107] Nicholas Rescher. *Many-Valued Logic*. New York: Mcgraw-Hill, 1969.

[108] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009. ISBN 0136042597.

[109] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006. ISSN 0733-8716. doi: 10.1109/JSAC.2002.806121. URL https://doi.org/10.1109/JSAC.2002.806121.

[110] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. Scalable product line configuration: A straw to break the camel's back. pages 465–474, 2013.

[111] Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted secure multi execution. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1617–1634, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243806. URL https://doi.org/10.1145/3243734.3243806.

[112] Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted secure multi execution. In *CCS '18*, 2018.

[113] Tom Schrijvers, Bart Demoen, Benoit Desouter, and Jan Wielemaker. Delimited continuations for prolog. *TPLP*, 13(4-5):533–546, 2013. doi: 10.1017/S1471068413000331. URL http://dx.doi.org/10.1017/S1471068413000331.

[114] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. pages 667–678, 2016.

[115] Ofer Shtrichman. Pruning techniques for the sat-based bounded model checking problem. In Tiziana Margaria and Tom Melham, editors, *Correct Hardware Design and Verification Methods*, pages 58–70, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44798-6.

[116] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. 20(3-4):487–517, 2012.

[117] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7597-7. URL http://dl.acm.org/citation.cfm?id=244522.244560.

[118] JOM Silva and Karem A Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 152–161. IEEE, 1997.

[119] K. Smeltzer and M. Erwig. Variational Lists: Comparisons and Design Guidelines. In *ACM SIGPLAN Int. Workshop on Feature-Oriented Software Development*, pages 31–40, 2017.

[120] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. pages 47–60, 2011.

[121] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe Composition of Product Lines. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*, pages 95–104, 2007.

[122] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. 47 (1):6:1–6:45, 2014.

[123] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. Analysis Strategies for Software Product Lines: A Classification and Survey. pages 57–58, 2015.

[124] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. Towards Efficient Analysis of Variation in Time and Space. pages 57–64, 2019.

[125] Cesare Tinelli. The smt-lib format: an initial proposal. 01 2003.

[126] Linus Torvalds. Linux Operating System. `www.kernel.org`. Accessed at December 02, 2019.

[127] Frank van Harmelen, Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of Knowledge Representation*. Elsevier Science, San Diego, CA, USA, 2007. ISBN 0444522115.

[128] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. pages 1–13, 2018.

[129] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. pages 58:1–58:11, 2012.

[130] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. Presence-Condition Simplification in Highly Configurable Systems. pages 178–188, 2015.

[131] Eric Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, 2013. `http://hdl.handle.net/1957/40652`.

[132] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226, 2014.

[133] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. Satire: A new incremental satisfiability engine. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 542–545, New York, NY, USA, 2001. Asso-

ciation for Computing Machinery. ISBN 1581132972. doi: 10.1145/378239.379019. URL https://doi.org/10.1145/378239.379019.

[134] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. ISSN 1471-0684.

[135] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. ISSN 00994987. URL http://www.jstor.org/stable/3001968.

[136] Chu-Pan Wong. *Beyond Configurable Systems: Applying Variational Execution to Tackle Large Search Spaces*. PhD dissertation, Carnegie Mellon University, 2021.

[137] Jeffrey Young, Eric Walkingshaw, and Thomas Thüm. Variational Satisfiability Solving. 2020. To appear.

[138] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press. ISBN 0-7803-7249-2. URL http://dl.acm.org/citation.cfm?id=603095.603153.

APPENDICES

Appendix A: Redundancy

This appendix is inoperable.