Variational Satisfiability Solving

Thesis proposal

Jeffrey M. Young
Department of Electrical Engineering and Computer Science
Oregon State University
youngjef@oregonstate.edu

November 27, 2020

Abstract

Over the last two decades, satisfiability and satisfiability-modulo theory (SAT/SMT) solvers have grown powerful enough to be general purpose reasoning engines employed in various areas of 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 terms in a set that are shared between problems and terms which are unique. By placing the onus on end-users to write a program, 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. In this thesis, I apply results from research on *variational* programming languages to the domain of SAT/SMT solvers to automate this interaction, creating the first variational SAT/SMT solver. I demonstrate numerous benefits to this approach: End-users need only identify the set of SAT/SMT problems to solve rather than identify the set *and* provide a program. Otherwise difficult optimizations can now be automatically detected and applied. Through use of variational constructs, the variational SAT/SMT can be made asynchronous and both single threaded and multi-threaded versions of variational SAT/SMT solvers are more performent in their expected use case.

1 Introduction

Classic satisfiability solving (SAT), which solves the boolean satisfiability problem [1] 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 [2], SAT solvers based on conflict-driven clause learning (CDCL) [3–5] have been able to solve boolean formulae with millions variables quickly enough for use in real-world applications [6]. Leading to their proliferation into several fields of scientific inquiry ranging from software engineering to Bioinformatics [7, 8].

However, 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 [9–15] require solving a set of related SAT problems [9, 10, 14]. To take just one example, software product-lines (SPL) utilizes SAT solvers for a diverse range of analyses including: automated feature model analysis [16–18], feature model sampling [19, 20], anomaly detection [21–23], and dead code analysis [24].

This gap between the SAT research community and the practical use cases of SAT solvers is well known. To address the gap, 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) [10, 11] where learned clauses from one problem in the problem set are propagated forward to future problems. Modern solvers are based on a major breakthrough that occurred with *incremental SAT under assumptions*, introduced in Minisat [25].

Incremental SAT under assumptions, made two major contributions: a performance contribution, where several pieces of information including learned clauses, restart and clause-detection heuristics are carried forward. A usability contribution; Minisat exposed an interface which 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 in the problem set.

Despite the success of incremental SAT, the incremental interface can be improved in two ways: First, 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. Second, 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.

In this thesis, I hypothesize usability and performance improvements to incremental SAT are possible by applying recent work on *variation*, and *variational programming* [26–30], which defines a theory of variation and formalizes a language to expresses variation called the *choice calculus*. With the choice calculus, the aforementioned set of problems is able to be expressed statically as a *variational artifact*. With this representation, the interface to incremental solving can be automated through a *variational interpreter*, furthermore by identifying and isolating the variational nature in incremental solvers, optimizations derived from the choice calculus become possible.

The goal of my research is to explore the design space and architecture of a *variational satisfiability solver* that uses research on variation in the context of incremental SAT solving. The rest of this section expands on these claims. Sec. 1.1 lists the specific contributions this thesis will make and outlines the rest of this document.

1.1 Proposed Contributions

The high-level goal of my research is to formalize and construct a variational satisfiability solver that understands and can solve SAT problems that contain *variational values* in addition to boolean values. In pursuit of this goal, my thesis will make the following contributions, items which are complete at time of this writing are indicated with a ✓:

1. *Variational propositional logic*: SAT solvers input and operate on sentences in propositional logic [31]. Variational satisfiability solvers, in order to reason about variation, must input sentences in a propositional logic that is *variational*, i.e. a many-valued logic [32] which contains variational values, called *choices* in addition to boolean values.

   The formulation of variational propositional logic (VPL) is requisite and central to the high level goal of designing a variational satisfiability solver. Furthermore, VPL serves two other functions: It provides an avenue for future work through the formalization of variation in the domain of propositional logic for variational satisfiability solvers. It provides a foundation for research on variation in propositional logic outside of the considerations of satisfiability solvers.

   This work is nearly complete. The logic has been formalized and successfully used in a prototype variational solver [33]. Sec. 3 introduces VPL and describes the following contributions which are directly enabled by it:

   (a) ✓ A set of variation preserving equivalences. Similar to the well known propositional logic equivalences, such as DeMorgan's law, these equivalences allow a variational solver to refactor input possibly yielding simpler variational sentences.

   (b) An efficient algorithm for translating a set of propositional formulae into a single VPL formula. The prototype variational SAT solver used a naive algorithm, and preliminary results showed that the encoding impacts solver performance. Hence, finding a more efficient encoding algorithm is desirable. This work is yet to be done but there are two promising routes forward. First, a naive algorithm which interleaves syntactic equivalences to produce a VPL formula that is easier to solve. Second, an algorithm similar to Huffman codes [34] to translate the SAT problems into a data structure, then use heuristics to select high quality candidates to combine. With such an algorithm the end-user of the variational solver only needs to input their problem sequence rather than a VPL formula.

2. *A variational satisfiability solver*: This is the central contribution of my thesis. It is completed and is published in a peer-reviewed conference [33] paper. Preliminary results are promising but based on only two case studies from the SPL community.

   Sec. 4 discusses these results and provides an overview of the variational solving algorithm. The following contributions are based on this work:

   (a) ✓ Formalization of a variational SAT solving algorithm that inputs a VPL formula and outputs a *variational model*.

   (b) ✓ Formalization of variational models; that is satisfying assignments of values to variables in input formula that succinctly represent results in the context of variation.

   (c) ✓ A method for determining the amount of variation in a given VPL formula.

   (d) A method for determining the relative hardness of a VPL formula based on work in the random-SAT community [35]. This item is orthogonal to all other items and thus can be done in parallel.

3. *A concurrent variational satisfiability-modulo theories (SMT) solver*: Contingent on item 2, *satisfiability modulo theories* extends SAT solvers such that they are able to reason about logical formulas in combination to *background theories*, such as arithmetic or arrays. Furthermore, with variation statically represented in a VPL formula, the SAT or SMT procedure can be made asynchronous leading to speedups on multi-core machines. The approach is to change the semantics of a choice; in the prototype SAT solver each choice blocks future SAT problems from being solved, by creating an asynchronous solving algorithm these future problems are unblocked and can be processed earlier.

   This item is an extension of the central contributions of the thesis. There are two extensions to the previous work to construct a variational SMT solver and one to make it asynchronous.

   First, the extensions to VPL abstract logical connectives in VPL allowing for theories which conclude to a Boolean value, such as arithmetic inequalities, and thus can be reasoned about in a SMT solver. Second, variational models are similarly extended, rather than assuming only Boolean values, the extension allows for polymorphic results through the use of SMTLIB2 compliant functions.

   Third, I extend the semantics of choices in the variational SAT solver to include atomic concurrent operations. When a choice is observed the solver state is copied and sent to a thread with instructions to compute continue the computation.

   This work is completed but unpublished. Sec. 5 expands on this item and discusses the evaluation of the prototype variational SMT solver with additional case studies. The following summarizes the expected contributions:

(a) ✓ Formalize the extension of VPL with SMT theories.

(b) ✓ Formalize the extension of variational models to express SMT results.

(c) ✓ Formalize the asynchronous variational solving algorithm

(d) A set of optimizations based on work on nanopass compilers [36] from the scheme programming language community [37]. The goal is to leverage VPL equivalence rules and other compiler optimizations, such as inlining, on SMTLIB2 programs, thus optimizing variational SMT programs. The prototype variational SMT solver is architected as a nanopass compiler and thus is able to perform optimizations as a single pass over the input formula. However, no optimizations are performed as of yet, although all requisite items for this work to begin are done.

(e) An empirical evaluation of solver performance. The empirical evaluation will reuse the datasets the prototype SAT solver was evaluated on. In addition, three new data sets will be added, two by harvesting SAT problems from work on variational lexing, parsing, and type checking [38] real world software such as Busybox [39] and the Linux kernel [40], and one by generating variational SMT problems. This dataset will be used several times in the thesis and will be made public. First, as a foundation to test the encoding strategies from item 1b. Second, to evaluation the optimizations from item 3d and third, to evaluate the performance of the single threaded and multi-threaded variational SMT solver. This work is partially complete, random generation of variational SAT and SMT problems is done, as is the Busybox dataset. The remaining work is to scale the logging solution to handle the Linux kernel.

4. *Proof of variation preservation*: A proof of variation preservation is a proof that the results of the variational solvers are sound, i.e., for any variant v, if a variational solver finds $VSat(v) = True$, then $Sat(v) = True$. Both item 2 and item 3 are verified sound via property-based testing [41] but the variational solving algorithm itself has not been proven sound up to the soundness of the underlying incremental solver. This work is in progress using the proof assistant Agda [42] and is expected to yield such a proof. Sec. 5 discusses this item further and lists the specific tasks left to do.

1.2 Significance and Potential Impact

The goal of this thesis is to explore the design and architecture of a variational satisfiability solver. The solver should allow end-users to input a set of propositional formulae and output a model that is useful *without* requiring the end-user to understand or be aware of research on variation.

This work is applied programming language theory in the domain of satisfiability solvers. Many analyses in the software product-lines community use incremental SAT solvers. By creating a variational SAT solver it is likely that such analyses would directly benefit from this work, and thus advance the state of the art.

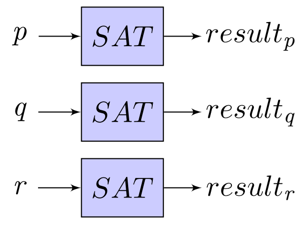
For researchers in the incremental satisfiability solving community, this work serves as an avenue to construct new incremental SAT solvers which efficiently solve classes of problems that deal with variation, by exploiting results from the programming language community.

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 possible benefits, such as parallelism, researchers in other domains may attain and thereby magnifies the impact of any results produced by the variational research community. Furthermore, the result of my thesis, a variational SAT solver, provides a new logic and tool to reason about variation itself.

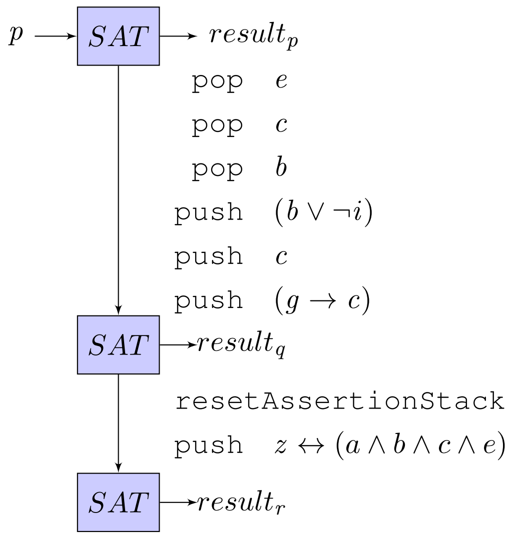
For researchers in other domains, a requisite result in constructing a variational satisfiability solver is a variational compiler; which translates VPL to a solver-domain programming language. Thus, while my thesis is focused on the domain of SAT solvers, this work describes a first of its kind variational compiler whose architecture may be reused to create new variation-aware tools such as build systems or programming languages. Such compilers could directly benefit from item 3d as this item describes performance improvements that *are only possible* with an explicit and static representation of variation.

2 Background

This section provides necessary background on incremental SAT solving. All descriptions follow the SMT-LIB2 [43] standard and describe incremental solvers as a black box eliding internal details of any specific solver which adheres to the standard.



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



(b) Incremental procedure, reuse defined by pop and push calls.

Figure 1

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

$$p = \ a \land b \land c \land e \qquad q = \ a \land (b \lor \neg i) \land c \land (g \to c) \qquad r = \ z \leftrightarrow (a \land b \land c \land e)$$

p is simply a conjunction of variables. In q, relative to p, we can see that two variables are added, i, g, one variable is removed e, and and there are two new clauses: $(b \lor \neg i)$ and $(g \to c)$, 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 satisfying assignment for each formula. Using a classic SAT solver results in the procedure illustrated in Fig. 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. 1b; in this scenario, all of the formulas are solved by single solver instance where terms are programmatically added and removed from the solver throughout the process. The ability to add and remove terms from the solvers is enabled by a data structure within the incremental SAT solver called an *assertion stack*. The assertion stack is a stack of declarations, definitions, or formulas that determine the *context* of the solver. A solver context is the union of all global variable definitions and everything on the assertion stack. A program may add an assertion to the stack via the `push` operation and remove from the top via a `pop` operation [44]. A call to `resetAssertionStack` pops everything on the stack.

In an efficient process one would initially add as many *shared* terms as possible; p in this example. Then check for satisfiability, request a model, and manipulate the assertion stack to reach the next problem of interest; q in this case. Notice that to reach the next problem, q, from p, several operations are required: e and c must be removed, b must be updated, and the new clauses must be introduced. To reach r from q all assertions would need to be popped to add z, then re-pushed.

3 Proposal Contribution 1: VPL = Variation + Propositional Logic

In this section, I present the syntax and semantics of variational propositional logic. While this section fulfills the majority of item 1 I restate it here to serve as background for Sec. 4, Sec. 5 and the choice calculus. I conclude the section by summarizing work left to do. The logic is a conservative extension of classic two-valued logic (C_2)[1] with a *choice* construct from the choice calculus [27, 45], a formal language for describing variation.

3.1 Syntax

The syntax of variational propositional logic is given in Fig. 2a. 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 rest of the proposal.

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 researchers unfamiliar with work on variation, it is useful to think of VPL as analogous to `#ifdef`-annotated C_2, where choices correspond to a disciplined [46] application of `#ifdef` annotations. From a logical perspective, following the many-valued logic of Kleene [32], 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 [47], because a choice *waits* until there is enough information to choose an alternative (i.e., until the formula is *configured*).

The *configuration semantics* of VPL is given in Fig. 2b and describes how choices are eliminated from a formula. The semantics are 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

[1]Notation for propositional logic comes from work on many-valued logic, see [32].

$$
\begin{array}{lllll}
t & ::= & r \mid \mathtt{T} \mid \mathtt{F} & & \textit{Variables and Boolean literals} \\
f & ::= & t & & \textit{Terminal} \\
 & \mid & \neg f & & \textit{Negate} \\
 & \mid & f \vee f & & \textit{Or} \\
 & \mid & f \wedge f & & \textit{And} \\
 & \mid & D\langle f, f\rangle & & \textit{Choice}
\end{array}
$$

(a) Syntax of VPL.

$$C : D \to \mathbb{B}_{\perp} \textit{Configuration}$$

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

(b) Configuration semantics of VPL.

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

(c) VPL equivalence laws

Figure 2: Formal definition of VPL.

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. 2c defines a set of equivalence laws for VPL formulas. These laws follow directly from the configuration semantics in Fig. 2b 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 Research Plan

The previous sections describe VPL but is missing an efficient strategy for encoding sets of C_2 formulas in a VPL formula. The proposed thesis will directly address this gap:

Encoding strategies This item will produce an efficient algorithm that combines a set of C_2 formulas into a single VPL formula. Efficient has two meanings: It should produce a VPL formula in reasonable time, and it should produce a VPL formula that has measurably less variation if possible. Such an algorithm is desirable for two reasons. First, it is practically important; a result of a previous study of variational satisfiability solving [33] was that the greater the *sharing ratio*, the ratio of plain to total terms in a variational formula, the faster the VPL formula was solved, on average. Thus, by developing a more efficient encoding algorithm, the performance of the solver is less volatile. Second, it lowers the barrier of use, with such an algorithm, the a new user because the new user need not understand VPL, rather they only need to identify the problem set they desire to solve.

A naive encoding algorithm is easy to construct: one wraps all formulas in unique choices and then uses equivalency rules to increase the sharing ratio. However, it is likely that better algorithms exist. I envision an algorithm that utilizes a strategy similar to Huffman coding [34] to find similar formulas to merge.

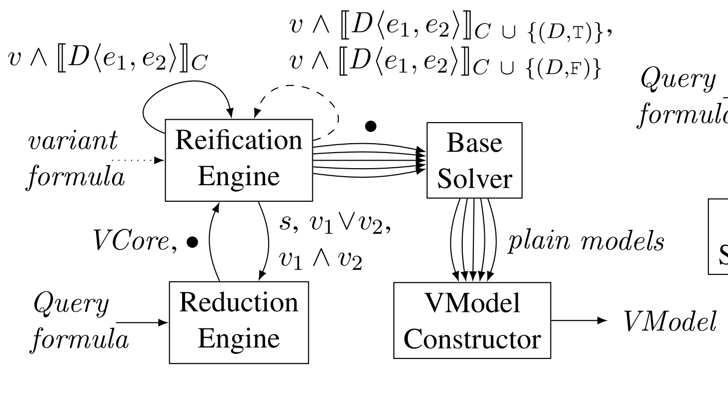
This work is able to be done in parallel to much of the other deliverables but intersects with two other items. First, a proof of variation preservation, item 4, for a variational solver must include a similar proof for the encoding strategy. Second, the encoding strategy will affect solver performance and thus also affect the evaluation, item 3e. Lastly, the performance of the encoding strategy itself will require a set of data to be evaluated on. I discuss harvesting such a set of data from real world software product lines in Sec. 5, although enough real world data is already available to begin on this item.

4 Proposal Contribution 2: Variational Satisfiability Solving

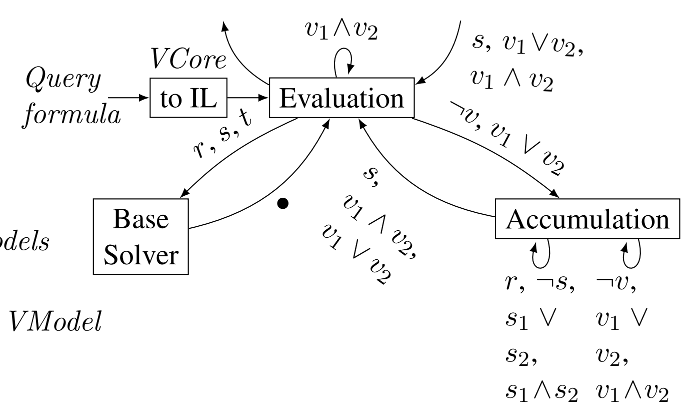
A core contributions of the proposed thesis is the design and architecture of a variational satisfiability solver. In this section, I review the architecture of a prototype solver called VSAT and conclude the section by summarizing work left to do on item 2d, i.e., assessing the hardness of variational satisfiability problems.

4.1 Design

Research on SAT and incremental SAT is fast moving with novel SAT solvers regularly competing in the International SAT Competition [48], to leverage these results we design variational satisfiability solving to target the incremental interface specified in the SMT-LIB2 [49] standard. Targeting the standard provides


$v \wedge [\![D\langle e_1, e_2\rangle]\!]_C$
$v \wedge [\![D\langle e_1, e_2\rangle]\!]_{C \cup \{(D,T)\}}$,
$v \wedge [\![D\langle e_1, e_2\rangle]\!]_{C \cup \{(D,F)\}}$
variant formula
Reification Engine
Base Solver
•
VCore, •
s, $v_1 \vee v_2$, $v_1 \wedge v_2$
plain models
Query formula
Reduction Engine
VModel Constructor
VModel

(a) System overview of a variational solver.


Query formula
VCore
to IL
Evaluation
$v_1 \wedge v_2$
s, $v_1 \vee v_2$, $v_1 \wedge v_2$
r, s, t
$\neg v$, $v_1 \vee v_2$
Base Solver
•
s, $v_1 \wedge v_2$, $v_1 \vee v_2$
Accumulation
r, $\neg s$, $s_1 \vee s_2$, $s_1 \wedge s_2$
$\neg v$, $v_1 \vee v_2$, $v_1 \wedge v_2$

(b) Overview of the reduction engine.

Figure 3

several benefits: an implementation of a variational solver is free to choose any SMT-LIB2 [49] compliant solver ranging from experimental solvers such as cvc4 [50], to industrial strength solvers such as z3 [51]. Since the implementation is SAT solver agnostic, a variational solver can be run as a *meta-solver*, i.e., a SAT solver that interleaves, or simultaneously uses several SAT solvers to solve a SAT problem, especially in asynchronous workloads. Lastly, any result from the SAT or SMT community that enters the standard is supported, this includes new SMT theories.

In addition to coupling to SMTLIB2, the variational SAT solver should not place further burden on the end-user. Thus users of a variational solver should not be required to understand the choice calculus in order to interpret their results, a variational solver's output should not contain choices. This design principle in combination with item 1b completes the approach and would allow an end-users to receive the benefits of this work without additional upfront cost.

4.2 Architecture

This section provides an informal description of variational satisfiability solving and variational models, I provide the formalization in the next section. A variational satisfiability solver is a compiler from the domain of variational formulas to SMT-LIB2 programs. Throughout this section, I use SMTLIB2 snippets to describe variational solving concepts in terms of an incremental solver. While I target SMTLIB2, conforming to the standard is not an essential requirement. Any solver that exposes an incremental API as defined by minisat [44] can be used to implement variational satisfiability solving following the same architecture and semantics.

A VPL formula is solved using a recursive approach, decoupling the handling of plain terms from the handling of variational terms. The idea is to define a process to evaluate plain terms and skip choices, then define another process that can only configures choices thus introducing new plain terms to the formula that can be recursively processed. The base case is a variant, at which point a model can be queried and the assertion stack can be popped to backtrack to solve another variant.

I present an overview of a variational solver as a state diagram in Fig. 3a 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 a formula ready for reification called a *variational core*. 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 inputs 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. The variational solver translates the query formula to a formula in an intermediate language (IL) that the reduction and reification engines operate over; its syntax is given below.

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

The IL includes two kinds of terminals not present in the input query formulas: plain sub-terms that can be reduced symbolically will be replaced by a *symbolic reference* s, and sub-terms 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.

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. 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 shown in Fig. 3b will produce a variational core that will assert $(a \wedge b)$ in the base solver, thus pushing it onto the assertion stack and create a symbolic reference for $(p \wedge \neg q)$. This is done in two states: *evaluation*, which issues commands to the base solver to process plain terms, and *accumulation* which is called by evaluation to create symbolic references.

Generating the core begins with evaluation. Evaluation will match on the root node: $\wedge$, of f and recur 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 will again match 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 last recursive call where $v_1'' = a$, and $v_2'' = b$. At the base case both a and b are references, thus evaluation will send 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, and 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 sub-trees, ensuring information is not lost, but still allowing for the sub-tree 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 recursive edges, 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 $v_1 \vee v_2$, $v_1 = (p \wedge \neg q)$, 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, thus 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 sub-tree to a single symbolic term, $s_{s_p \wedge s_{\neg q}}$, which will be returned as the result for v_1. v_2 is a base case, hence accumulation will return $s_{s_p \wedge s_{\neg q}} \vee B\langle e_3, e_4\rangle$ to evaluation. Evaluation will conclude with $A\langle e_1, e_2\rangle$ as the result for the left child of $\wedge$ and $s_{s_p \wedge s_{\neg q}} \vee B\langle e_3, e_4\rangle$ for the right child, yielding $A\langle e_1, e_2\rangle \wedge (s_{s_p \wedge s_{\neg q}} \vee B\langle e_3, e_4\rangle)$ 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. To save computation evaluation moves sub-formulas into the solver state to be reused among different variants, and 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 sub-tree of the query formula. From the perspective of the base solver a symbolic reference represents a set of programmatic statements. For example, s_{pq} represents three declarations in the base solver:

```
(declare-const p Bool)          ;; $s_{pq}$ represents
(declare-const q Bool)          ;; several declarations
(declare-fun $s_{ab}$ () Bool (or p (not c)))
```

The program language shown here is lisp [52] as defined in the SMTLIB2 standard. Function application begins with an open parenthesis, where the first symbol in the parenthesis is the name of the function, every symbol after the first is an argument to that function. In the above snippet, we see three function calls, two which declare constants in the program for p and q, both with type $Bool$, and one which declares a new function which takes no input and returns a $Bool$.

Similar to symbolic references, a variational core is a sequence of statements in the base solver with holes $\Diamond$. For example, the representation of $VCore_f$:

```
(assert (and a b))               ;; $a \wedge b$ on the assertion stack
(declare-const $\Diamond_A$)              ;; choice A
 ⋮                               ;; many declares may occur
(assert $\Diamond_A$)                     ;; many assertions may occur
 ⋮                               ;; $s_{pq}$
(declare-fun $s_{pq}$ () Bool (and p q))
(declare-const $\Diamond_B$)              ;; choice B
 ⋮
(assert (or $s_{ab}$ $\Diamond_B$))              ;; assert waiting on $[\![B\langle e_3, e_4\rangle]\!]_C$
```

Each hole is filled by configuring a choice and may require multiple statements to process the alternative as the alternative could introduce several new variables or functions.

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 VPL, a configuration was formalized as a function, for variational solvers we use a set of tuples $\{(D \times \mathbb{B})\}$. Fig. 3a shows two self-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 construct an initial configuration. Finally, a model is called from the base solver when the reduction engine returns $\bullet$, indicating that a variant has been found.

We display a subset of edges of the reification engine using the $\wedge$ connective. In general, these edges will be duplicated for each binary logical connective, e.g., $\vee$. 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 are 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: $C \cup \{(D, \mathtt{T})\}$, and $C \cup \{(D, \mathtt{F})\}$, and the recursive call is wrapped with a `push`, and `pop` command. To the base solver, this branching is a linear sequence of assertion stack manipulations that performs backtracking behavior, for example the representation of f is:

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

Where the hole $\lozenge_B$[2], will be filled with a newly defined variable s_{B_T} that represents the left alternative's formula.

Variational Models Classic SAT models map variables to Boolean values; variational models map variables to variational contexts that record the variants where the variable was assigned T. The variational context for a variable r is denoted as vc_r, and a variational model reserves a special variable called $\_Sat$ to track the configurations that were found satisfiable. As an example, consider an altered version of the

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

Figure 4: Possible plain models for variants of f.

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

Figure 5: Variational model corresponding to the plain models in Fig. 4.

query formula from the previous section $f = ((a \wedge \neg b) \wedge A\langle a \to \neg p, c\rangle) \wedge ((p \wedge \neg q) \vee B\langle q, p\rangle)$. 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; the corresponding variational model is shown in Fig. 5.

[2] the notation $[x \to v]p$ should be read "replace all free occurrences of x in p with v", it derives from work on explicit substitution from the programming languages community [53]

$$\frac{r \notin dom(\Delta) \qquad \texttt{spawn}(\Delta, r) = (\Delta', s)}{(\Delta, r) \mapsto (\Delta', s)}\ \textsc{Ac-Gen} \qquad \frac{\Delta(r) = s}{(\Delta, r) \mapsto (\Delta, s)}\ \textsc{Ac-Ref}$$

$$\frac{\texttt{negate}(\Delta, s) = (\Delta', s')}{(\Delta, \neg s) \mapsto (\Delta', s')}\ \textsc{Ac-Neg}$$

$$\frac{\texttt{or}(\Delta, s_1, s_2) = (\Delta', s')}{(\Delta, s_1 \vee s_2) \mapsto (\Delta', s')}\ \textsc{Ac-SOr} \qquad \frac{\texttt{and}(\Delta, s_1, s_2) = (\Delta', s')}{(\Delta, s_1 \wedge s_2) \mapsto (\Delta', s')}\ \textsc{Ac-SAnd}$$

Figure 6: Selected accumulation semantics on IL formulas.

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 $C_{FT} = \{(A, \texttt{F}), (B, \texttt{T})\}$ is returned, one can find a variable's value through substitution with the configuration; for example, substituting the returned model on vc_c yields:

$$\begin{aligned} &c \to (\neg A \wedge \neg B) \vee (\neg A \wedge B) && vc \text{ for } c \\ &c \to (\neg \texttt{F} \wedge \neg \texttt{T}) \vee (\neg \texttt{F} \wedge \texttt{T}) && \text{Substitute } \texttt{F} \text{ for } A,\ \texttt{T} \text{ for } B \\ &c \to \texttt{T} && \text{Result} \end{aligned}$$

Furthermore, to find variants where a variable c is satisfiable reduces 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. 5, 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 under with $\vee$ as the semigroup operation, and F as the unit value. I take note of this for mathematically inclined readers because it has important ramifications for the asynchronous version of variational satisfiability solvers.

4.3 Formalization of Variational Satisfiability Solving

This subsection presents a selection of inference rules that specify behavior described in the previous subsection. Many inference rules are similar to others due to commutativity of boolean operators and thus I only present an interesting subset. These inference rules will be significantly altered in the proposed thesis to fully generalize to SMT problems.

Accumulation is defined in Fig. 6 as a relation of the form $(\Delta, v) \mapsto (\Delta, v)$, where Δ is a symbolic store, and v is the syntactic category representing the set of all possible IL formulas; the tuples on the left and right are interpreted as input and output, respectively. Accumulation interacts with the symbolic execution engine via primitive operations represented in `typewriter` font. The ACC-GEN rule generates new symbolic references using `spawn`, which are looked up by ACC-REF. The ACC-AND and ACC-OR rules

$$\frac{\texttt{assert}\,((\Gamma,\Delta),t)\ =\Gamma'}{(\Gamma,\Delta,t)\rightarrowtail(\Gamma',\Delta,\bullet)}\ \textsc{Ev-Term}\qquad\frac{\texttt{assert}\,((\Gamma,\Delta),s)\ =\Gamma'}{(\Gamma,\Delta,s)\rightarrowtail(\Gamma',\Delta,\bullet)}\ \textsc{Ev-Sym}$$

$$\frac{}{(\Theta,\bullet\wedge v)\rightarrowtail(\Theta,v)}\ \textsc{Ev-UL}\qquad\frac{}{(\Theta,v\wedge\bullet)\rightarrowtail(\Theta,v)}\ \textsc{Ev-UR}$$

$$\frac{(\Delta,\neg v)\mapsto(\Delta',v')}{(\Gamma,\Delta,\neg v)\rightarrowtail(\Gamma,\Delta',v')}\ \textsc{Ev-Neg}$$

$$\frac{(\Delta,v_1)\mapsto(\Delta',v_1')\qquad(\Delta',v_2)\mapsto(\Delta'',v_2')}{(\Gamma,\Delta,v_1\vee v_2)\rightarrowtail(\Gamma,\Delta'',v_1'\vee v_2')}\ \textsc{Ev-Or}$$

$$\frac{(\Theta,v_1)\rightarrowtail(\Theta',v_1')\qquad(\Theta',v_2)\rightarrowtail(\Theta'',v_2')}{(\Theta,v_1\wedge v_2)\rightarrowtail(\Theta'',v_1'\wedge v_2')}\ \textsc{Ev-And}$$

Figure 7: Selected evaluation semantics over VPL formulas.

reduce symbolic sub-formulas to a new symbolic reference. This selection of rules do the work of reducing formulas to symbolic references. The remaining rules simply push negation down expressions and propagate accumulation over the $\wedge$ and $\vee$-connectives.

Evaluation is defined in Fig. 7 as a relation of the form $(\Theta, v) \rightarrowtail (\Theta, v)$, where $\Theta = (\Gamma, \Delta)$ and Γ represents the base solver state. As before, we show only a significant subset of the rules here. The rules EV-TERM and EV-SYM push new clauses to the base solver using the primitive `assert` operation. The EV-UL and EV-UR implement left and right unit, reducing conjunctions where one side has been processed by the base solver. Of special note is the difference between the EV-OR and EV-AND rules. While EV-AND is a straightforward congruence rule, EV-OR instead processes its arguments using accumulation ($\mapsto$). Disjunctions are a source of back-tracking in variational solving, and thus the solver cannot evaluate the left-hand side without evaluating the right, both of which may contain choices, hence evaluation must switch to accumulation, as we informally described in the previous subsection.

Solving the core is defined in Fig. 8, as a relation $(C, \Gamma, \Delta, m, v) \Downarrow_i (C, \Gamma, \Delta, m, v)$, where C, is a set which represents the configuration of the VPL formula, and m represents the variational model, which is initialized as empty. The count of `push`'s on the assertion stack are represented with the counter i. The solving process reifies choices by manipulating the configuration and uses accumulation and evaluation to process terms. The vc input to the solver pre-populates the configuration, thereby restricting the solver to a subset of variants. When no vc is input, the configuration is initialized as empty. CR-CAND processes novel choices by manipulating the configuration and performing a `push` in the base solver; resulting variational models are merged via an element-wise $\vee$, shown as $\cup$. Choices are removed through CR-AND-T and CR-OR-T by selection on the alternative. Once the choice is removed, the nested clauses are translated to the intermediate language through the `toIL` primitive and processed by accumulation and evaluation. A model is called from the base solver with GEN, once the core, and thus query formula is reduced to $\bullet$. Note again, CR-OR switches to accumulation to ensure sound results, we have omitted congruence rules such as CR-AND. Similarly, we omit rules which are commutative versions of those shown here, namely: rules which process the left branch of connectives, rules which select the *false* alternatives, and the OR version of CR-CAND.

$$\frac{\texttt{getModel}(\Gamma, \Delta) = m'}{(C, \Theta, m, \bullet) \Downarrow_i (C, \Theta, \oplus(C, m', m), \bullet)}\ \textsc{Gen} \qquad \frac{(\Theta, s) \rightarrowtail (\Theta', \bullet)}{(C, \Theta, m, s) \Downarrow_i (C, \Theta', m, \bullet)}\ \textsc{Sym}$$

$$\frac{(\Delta, v_1 \vee v_2) \mapsto (\Delta', v)}{(C, \Gamma, \Delta, m, v_1 \vee v_2) \Downarrow_i (C, \Gamma, \Delta', m, v)}\ \textsc{Cr-Or}$$

$$\frac{(D, \mathsf{true}) \in C \qquad (C, \Theta, m, v \wedge \texttt{toIR}(e_1)) \Downarrow_i (C, \Theta, m, v')}{(C, \Theta, m, v \wedge D\langle e_1, e_2\rangle) \Downarrow_i (C, \Theta, m, v')}\ \textsc{Cr-And-T}$$

$$\frac{(D, \mathsf{true}) \in C \qquad (C, \Theta, m, v \vee \texttt{toIR}(e_1)) \Downarrow_i (C, \Theta, m, v')}{(C, \Theta, m, v \vee D\langle e_1, e_2\rangle) \Downarrow_i (C, \Theta, m, v')}\ \textsc{Cr-Or-T}$$

$$\frac{\begin{array}{ll} D \notin C & (C \cup \{(D, \mathsf{true})\}, \Theta, m, v \wedge D\langle e_1, e_2\rangle) \Downarrow_{i+1} (C', \Theta', m', \bullet) \\ & (C \cup \{(D, \mathsf{false})\}, \Theta, m, v \wedge D\langle e_1, e_2\rangle) \Downarrow_{i+1} (C'', \Theta'', m'', \bullet) \end{array}}{(C, \Theta, m, v \wedge D\langle e_1, e_2\rangle) \Downarrow_i (C'', \Theta'', m' \cup m'', \bullet)}\ \textsc{Cr-CAnd}$$

Figure 8: Selected variational solving semantics on cores.

4.4 Research Plan

Estimating the difficulty of finding satisfiability This item will produce a method to determine the *hardness* of solving a VPL formula for satisfiability. A well known result in the random-SAT community is the phenomenon of a *phase transition* [35] in randomly generated SAT problems. The phase transition of SAT problems is an inflection point in the probability of finding a satisfying assignment as the ratio of clauses to variables is varied. Conceptually, one may think of the phase transition as a method to estimate the difficultly in solving a SAT problem. If there are many variables relative to clauses, then the SAT problem is likely easy to solve as it is under-constrained, in contrast, if there are too few variables relative to clauses then it is over-constrained and thus easy to compute as unsatisfiable.

Difficult problems are balanced with respect to the clause variable ratio and thus are at the phase transition point of a high probability of finding satisfiability to finding unsatisfiability. Estimating the difficulty of a SAT problem is thus theoretically useful to isolate sets of problems to study and progress the state of the art, but also practically useful, because an end-user may estimate the difficulty of a problem and choose to *not* solve it.

This item will replicate the analysis from the random-SAT community to determine if the phase transition exists for variational satisfiability problems. It is clear that the phase transition exists for variants, but having a method to assess the phase transition point *in terms of* VPL formulas would provide the aforementioned benefits for variational satisfiability solvers.

5 Proposal Contribution 3: Variational Satisfiable-Modulo Theory Solving

The final contribution to the thesis is generalizing the prototype variational solver to solve SMT problems. Like all objects that require craft, the architecture and design benefit greatly from lessons learned in the first prototype, I cover these advances below and conclude the section with a discussion of remaining work.

5.1 Motivation

This section discusses the motivation for a variational SMT solver. Specifically, for a variational SMT solver that *does not* use the theory interface of SMT solvers to reason about variation. Satisfiable modulo theory solvers, by virtue of abstracting satisfiability solving over background theories, are useful in multitudes of domains including: verification [54], test generation and bug finding [55, 56], planning or scheduling applications [57,58], interactive proof assistants [51,59], and many more [60].

The motivation in creating a variational SMT solver is identical to the motivation for a variational SAT solvers for a subset of theories i.e., the incremental interface is automated, the user need not hand-program the solver, performance benefits are now possible by virtue of a static explicit encoding. Furthermore, SMT solvers provide a good platform for research on variation. Understanding and implementing variational effects and effect handlers is an active and unsolved area of research on variation and variational programming. Essentially, the problem is soundly tracking side-effects such as file I/O or state mutation in the context of variation.

Variational SMT solvers side step this issue by building upon a ground theory of *uninterpreted functions*. Uninterpreted functions are functions that have no apriori meaning, such as a function which defines a constant value, as opposed to a function like + which apriori means to add two integers. Thus functions in the SMT domain, including those in background theorys are total and side-effect free, making SMT solvers an attractive target for variational research.

Besides the motivation deriving from variational research and inherited from the variational SAT solver, a variational SMT solver is desirable because it simplifies the use of a SMT solvers in real-world applications. For example, consider the following snippet of a C-like language that uses contract-based verification to ensure correctness:

```
@precondition: $x_{in} > y_{in}$
void swap(int x, int y) {
  x := x + y;
  y := x - y;
  x := x - y;
}
```

With a plain SMT solver one can prove that this code does indeed swap the variables by constructing an SMT problem that includes the following constraint $x_{out} = y_{in} \wedge y_{out} = x_{in}$, in addition to encoding the precondition, any constraints derived from the function body, and constraints derived from any post condition. However, in practice, code bases are variational artifacts, either through explicit variation annotations such as C preprocessor `#ifdefs` or through implicit practices such as branching and forking in version control systems. For example, in addition to the snippet above, one might have another variant that requires verification, perhaps to prevent a bug or as a safety check after a refactor:

```
@precondition: $x_{in} > y_{in}$
@precondition: $x_{in} > -1$      // new
int swap(int x, int y) {
  x := x + y;
  y := x - y;
  x := x - y;
  return 1;                 // new
}
```

In this variant, we see two minor additions; an additional precondition, $x_{in} > -1$, and a return value, note that most of the code has not changed. With a variational SMT solver one could use choices to express this difference and verify *both* variants instead of each variant at a time. Thus, a variational SMT solver, by virtue of being variation-aware, would allow verification tools to directly express variation using choices and verify the entire code base or software system, rather than each of its' variants one at a time.

Verifying the entire code base is possible using plain SMT solvers but the situation is analogous to the difference between a programming language which has the concept of looping, compared to a language that is not *loop-aware*. One might still express loops in the latter language, e.g., with `Goto` or `Gosub` primitives, but doing so is more error-prone, and more difficult than using a construct such as `While` that encapsulates and expresses the concept of looping.

One approach to construct a variational SMT solver is to add a background *theory of variation* to a plain SMT solver. This approach has many desirable properties; the vc that determines the variants of interest could be expressed in the solver, the solver would enforce the synchronization of choices, new theories could be supported, and the difference between a variational SAT solver and variational SMT solver would only be the inclusion of other theories in a problem.

Unfortunately, there are several issues which make this approach undesirable. First, it is overly solver specific. Some solvers such as openSMT [61] are architected specifically so the end-user can include custom theories, other solver such a yices [59], cvc4 [50] and z3 [51] have varying degrees of support. Yices and z3 provide no support[3], while cvc4 includes an API for custom theories but with incomplete documention. Second, adding a theory of variation to the solver limits the asynchronous implementation to the asynchronous attributes of the SMT solver. Lastly, the interaction between plain and the hypothetical variational theory is not clear. The interaction between combination of theories in SMT solvers is an active area of research [1]. By including a theory of variation, which is a *meta-theory*, i.e., a theory which operates on other theories to construct variation-aware theories, these problems are exacerbated. Thus, while it is possible to construct a theory of variation, we leave this to future research. Instead, we choose to create the variational prototype as a proof of concept variational SMT solver that uses a plain SMT solver as a black-box.

5.2 VPL Extensions

Extending the variational solver for SMT background theories requires non-trivial extensions to variational propositional logic, and consequently the intermediate language the solver operates upon. In SMT solving, Boolean values correspond to constraints over individual variables which range over different domains, such as arrays, arithmetic, bitvectors or strings. To support SMT theories the variational SMT solver must be able to abstract these theories and reason about them in a variational context. We show a simple extension of VPL to include integer arithmetic, and conclude the section by extending the variational SMT solver with an array theory.

To begin, we require formalizations of SMT background theories, for our purposes, we'll represent any SMT theory as a 2-tuple, consisting of a formal grammar G, and a semantic function with type $[\![\cdot]\!] : G \rightarrow \mathbb{B}$[4]. For the remainder of the proposal we represent all grammars in Backus-Naur form. For example,

[3]z3 had plugin support for custom theories but this feature was removed because model construction became problematic. The z3 developers now suggest treating z3 as a black box and building around it, just as this thesis proposes. See: https://stackoverflow.com/questions/46508907/smt-solver-with-custom-theories

[4]This formulation follows conventions from the programming language community. We choose this formulation to build upon the notation and background in Sec. 3. The SAT/SMT community would represent this a set called a *signature* that consists of words in the theory called *atoms*, functions which operate on the words, such as $+$ and $<$ and functions which maps sentences to Booleans called *predicates*

consider a simple background theory of integer addition, subtraction and two inequalities:

$$\begin{array}{rcll} i & \in & \mathbb{Z} & \textit{Integer literals} \\ \\ a & ::= & i & \textit{Terminal} \\ & | & a + a & \textit{Addition} \\ & | & a - a & \textit{Subtraction} \\ & | & a < a & \textit{Less Than} \\ & | & a > a & \textit{Greater Than} \end{array}$$

Note that with this formulation, the semantic function is partial, the theory should only allow syntax trees that have inequalities at the root such that a Boolean is the only type of value that can result. With the semantic function and grammar, the integer theory can be integrated into VPL. For example, one can imagine the following sound formula: $a \land \neg [\![((1 + 5) < 1729)]\!] \lor c$

With the definition of an SMT background theory we define a *variation-aware* background theory as a 2-tuple that consists of a variation-aware grammar, $G^{\langle\rangle}$, and a variation-aware semantic function $[\![\cdot]\!] : C \to G \to \mathbb{B}$. The alterations to the semantic function are minimal, requiring a configuration as the extra input to track choices. The semantics follow from the semantic function described in Fig. 2b only distributing over integer connectives instead of logical connectives such as $\land$ and $\lor$. Converting a grammar to a variation-aware grammar depends on the grammar at hand. In this case, the theory is a context free grammar and thus the only difference is adding choices as a recursive case:

$$\begin{array}{rcll} i & \in & \mathbb{Z} & \textit{Integer literals} \\ \\ a^{\langle\rangle} & ::= & i & \textit{Terminal} \\ & | & D\langle a^{\langle\rangle}, a^{\langle\rangle}\rangle & \textbf{\textit{Choice}} \\ & | & a^{\langle\rangle} + a^{\langle\rangle} & \textit{Addition} \\ & | & a^{\langle\rangle} - a^{\langle\rangle} & \textit{Subtraction} \\ & | & a^{\langle\rangle} < a^{\langle\rangle} & \textit{Less Than} \\ & | & a^{\langle\rangle} > a^{\langle\rangle} & \textit{Greater Than} \end{array}$$

Note that we could define the variation-aware theory only with a variation-aware domain $i^{\langle\rangle}$ which would add choices to the set of integers. Doing so would allow the variation-aware grammar to express expressions such as $A\langle 1, 2\rangle + 4$, *but not* $A\langle 10 + A\langle 2, 3\rangle, 3\rangle + 2$ because choices would only be allowed to range over integers and not expression. We'll use this behavior in the following array example.

More complicated theories, such as arrays [62, 63] require more careful handling. The array theory parameterizes an array with a type to determine the type of the array's elements, and includes only two functions: $\textit{select} : \textit{Array}\ \mathbb{N}_0\ X \to \mathbb{N}_0 \to X$, which given an array and a natural number index, creates a constraint that an element $x \in X$, is at index $n \in \mathbb{N}_0$, in the input array.
Similarly, $\textit{store} : \textit{Array}\ \mathbb{N}_0\ X \to \mathbb{N}_0 \to X \to \textit{Array}\ \mathbb{N}_0\ X$ constructs a constraint that at index $n \in \mathbb{N}_0$, the input array contains value $x \in X$. In SMTLIB2, these constraints obey the following law $\forall a \in \textit{Array}\ \mathbb{N}_0\ X,\ \forall i \in \mathbb{N}_0,\ e \in X,\ (= (\textit{select}\ (\textit{store}\ a\ i\ e)\ i)\ e)$. A simple formulation then for an array theory is:

$$\begin{array}{rcll} a & \in & Array\ \mathbb{N}_0\ X & \textit{all possible arrays} \\ i & \in & \mathbb{N}_0 & \textit{Natural Numbers} \\ x & \in & X & \textit{set of elements} \\ \\ arr & ::= & \textit{select}\ a\ i & \textit{Selection} \\ & | & \textit{store}\ a\ i\ x & \textit{Storage} \end{array}$$

The semantic function for this grammar is stateful, such that it can track the array and its constraints. The prototype SMT solver offloads this work to the underlying incremental solver and instead places holes in the array constraints.

We present a variation-aware grammar, $arr^{\langle\rangle}$, which maintains the same interface, i.e., *store* and *select*, but operates on variation-aware domains such as $\mathbb{N}_0^{\langle\rangle}$, $X^{\langle\rangle}$:

$$
\begin{array}{rcll}
a & \in & (Array\ \mathbb{N}_0\ X)^{\langle\rangle} & \textit{choice of all possible arrays} \\
i & \in & \mathbb{N}_0^{\langle\rangle} & \textit{choice of Natural Numbers} \\
x & \in & X^{\langle\rangle} & \textit{choice of elements} \\
\\
arr^{\langle\rangle} & ::= & select\ a\ i & \textit{Selection} \\
 & | & store\ a\ i\ x & \textit{Storage}
\end{array}
$$

This is a design decision, one could easily add an array theory which allows for a choice of *select* or *store*, similar to $a^{\langle\rangle}$, in addition to including variation-aware domains. Furthermore, we could restrict the language by choosing to only use a variation-aware element domain $X^{\langle\rangle}$, which would yield an array of choices, or only a variation-aware domain of arrays, $(Array\ \mathbb{N}_0\ X)^{\langle\rangle}$, yielding a choice of arrays. Both are possible with this formulation, for example, consider the following variational SMT program:

```
(declare-const e1 Int)
(declare-const e2 Int)
(declare-const a1 (Array Int Int))                   ;; an array of integers
(declare-const a2 (Array Int Int))                   ;; second array of integers
(assert (= (store a2      3 A⟨1202,2718⟩) a2))      ;; an array of choices
(assert (= (store A⟨a1,a2⟩ 3 1729       ) a1))      ;; a choice of arrays
(assert (= (select A⟨a1,a2⟩ 3) e1))
(assert (= (select a2      3) e2))
(check-sat)
(get-model)
```

We see that there are two integer arrays, $a1$ and $a2$, and three choices: one choice which chooses an array in *store*, another which chooses an element to store in *store* in $a2$, and a third choice to determine which array $e1$ retrieves its value from. All choices are parameterized by the dimension A yielding two variants, and results are returned in the $e1$ and $e2$ variables. The variational core for this program would simply replace the choices with holes:

```
 ⋮
(assert (= (store ◊A 3 1729) a1))  ;; a choice of arrays
(assert (= (store a2 3 ◊A)   a2))  ;; an array of choices
(assert (= (select ◊A 3) e1))
 ⋮
```

To solve such a program, the variational SMT solver will compile to SMTLIB2 wrapping variation-aware statements and statements affected by variation-aware statements, such as `(get-model)`, with a `push` and `pop` instruction:

```
(declare-const e1 Int)
(declare-const e2 Int)
(declare-const a1 (Array Int Int))       ;; an array of integers
(declare-const a2 (Array Int Int))       ;; second array of integers
(push)                                   ;; a configuration on A has occurred
(assert (= (store a2 3 1202) a2))
```

```
(assert (= (store a1 3 1729) a1))
(assert (= (select a1 3) e1))          ;; e1 set to 1729
(assert (= (select a2 3) e2))          ;; e2 set to 1202
(check-sat)
(get-model)
(pop)
(push)                                 ;; Right alternative of A
(assert (= (store a2 3 2718) a2))
(assert (= (store a2 3 1729) a1))      ;; a1 unifies with a2
(assert (= (select a1 3) e1))          ;; e1 set to 1729
(assert (= (select a2 3) e2))          ;; e2 set to 2718
(check-sat)
(get-model)
(pop)
```

5.3 Variational SMT Models

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 varational SMT model must allow users to find satisfying values for a variant. The model must allow users to find all variants at which a variable has a particular value or range of values. Furthermore, several useful properties of varational models, as presented in Sec. 4.2, should be maintained: The model is non-variational; hence the user does not need to understand the choice calculus in order to understand their results. The model produces results that can be fed into a plain SAT solver. The model can be built incrementally and without regard to the ordering of results because it forms a commutative monoid under $\{\texttt{F}, \vee\}$. The model maps variables to a context free grammar and can thus be parsed quickly[5].

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. By virtue of this strategy, variables are disallowed from changing types across the set of variants and hence disallowed from changing types as the result of a choice in the variational model. 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 *if-statements*. Specifically, we use the function $ite : \mathbb{B} \to T \to T$ as the `cons` operation to build the list. ite is defined in the ground theory of Booleans as defined in the SMTLIB2 standard. All variables are initialized as `undefined` until a value is found in a variant. To ensure the correct value of a variable corresponds to the appropriate variant, we translate the configuration that determines the variant to a variation context, and place the appropriate value in the *then* branch, with the else branch linking to the previous expression.

Consider the following variational SMT problem extended with an integer arithmetic theory: $f = (A\langle i, 13\rangle - c < b + 10) \to B\langle a, c > i\rangle$. 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.

Fig. 9 show possible plain models for f with the corresponding variational SMT model display in Fig. 10. We've added line breaks to emphasize the branches the $then$ and $else$ branches of the ite SMTLIB2 primitive.

This formulation maintains the user requirements of the model. We maintain a special variable $\_Sat$ to track the variants that were found satisfiable. 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

[5]The use of "quickly" here means linear or quadratic time using context-free grammar parser such as an Earley parser [64]

$$
\begin{array}{cccc}
\begin{array}{l} i \rightarrow \text{-1} \\ c \rightarrow 0 \end{array} & & \begin{array}{l} i \rightarrow 0 \\ c \rightarrow 1 \end{array} & \begin{array}{l} i \rightarrow 0 \\ c \rightarrow 0 \end{array} \\
 & a \rightarrow \texttt{T} & & \\
 & & & b \rightarrow \text{-10} \\
C_{FF} = \{(A, \texttt{F}), (B, \texttt{F})\} & C_{FF} = \{(A, \texttt{F}), (B, \texttt{T})\} & C_{FT} = \{(A, \texttt{T}), (B, \texttt{F})\} & C_{TT} = \{(A, \texttt{T}), (B, \texttt{T})\}
\end{array}
$$

Figure 9: Possible plain models for variants of f.

$$
\begin{array}{ll}
\_Sat & \rightarrow (\neg A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge \neg B) \vee (A \wedge B) \\
i & \rightarrow (\mathit{ite}\ (A \wedge B)\ \ 0 \\
 & \qquad\qquad (\mathit{ite}\ \ (A \wedge \neg B)\ \ 0 \\
 & \qquad\qquad\qquad\qquad (\mathit{ite}\ \ (\neg A \wedge \neg B)\ \ \text{-1}\ \mathit{Undefined}))) \\
c & \rightarrow (\mathit{ite}\ (A \wedge B)\ \ 0 \\
 & \qquad\qquad (\mathit{ite}\ \ (A \wedge \neg B)\ \ 1 \\
 & \qquad\qquad\qquad\qquad (\mathit{ite}\ \ (\neg A \wedge \neg B)\ \ 0\ \mathit{Undefined}))) \\
a & \rightarrow (\mathit{ite}\ (\neg A \wedge B)\ \ \texttt{T}\ \mathit{Undefined}) \\
b & \rightarrow (\mathit{ite}\ (A \wedge B)\ \ \text{-10}\ \mathit{Undefined})
\end{array}
$$

Figure 10: Variational model corresponding to the plain models in Fig. 9.

were interested in the value of i in the $\{(A, \texttt{T}), (B, \texttt{T})\}$ variant they would substitute the configuration into vc_i and recover 0 from the first ite case. To find the variants at which a variable has a value a user can employ a SMT solver, add vc_i as a constraint, and query for a model.

This also maintains the desirable properties of variational SAT models while allowing any type known to the SMT solver. The variational SMT model does not require knowledge of choice calculus or variation, it is still monoidal, although not 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, variational SAT models clearly compressed results by preventing duplicate values with constant variables. In contrast, the variational SMT model allows for duplicate values as long as those values are parameterized by disjoint variants. For example, both i and c contain duplicate values, but only one: i is easy to check in $O(1)$ time as the duplicates are sequential in vc_i and can thus be checked during model construction. Such a case would be easily avoided by tracking the set of all values a variable has been assigned in all variants. However, we chose to keep variational models as simple as possible and therefore only present the minimum required machinery.

5.4 Requirements and Design Principles

As before the items that are completed are marked with a ✓. The final thesis will address each item:

1. The user must be able to incrementally add to a variational core. This item recovers some of the incrementality lost from the synthesis of a VPL formula. I hypothesize two possible solutions: Given a variational core, a user can add new clauses to the core under the condition that no bound variable is removed in the new variational core. We require this constraint because if a user attempted to remove a variable in the variational core then the solver would need to unpack the symbolic references which could lead to unsound results. A second method for incrementality comes from work on CLU [65], under this method one would derive a variational core and then serialize or marshal it to disk, effectively caching the core, and the solver state, for future use or transmission to another solver instance.

2. ✓ Provide a *general* method to solve a variational core. A variational SMT solver can be extended with many background theories. Depending which theories the user requires, many possible formulas with varying types (or *sorts* in the SAT literature) and operators can be present in a query formula and thus in the variational core. Hence, the variational SMT solver must have a general method to reason about these types and operations. The improvement is to use a Huet zipper [66] to capture operators as a context, such as negation, as an Algebraic Data Type. With a zipper, one can traverse the variational core and delay the semantic of the operator thus processing the rest of the tree to a symbolic variable. Then, evaluating with any operator is reduced to only the denotational semantics of the operator, preserving compositionality and completeness.

3. ✓ The variational SMT solver should be able to solve a query formula concurrently or in a single-threaded mode. This item change requires changing the semantics of choices. A benefit of the static and explicit approach of representing variation using the choice calculus is that we can alter the denotational semantics of a choice very easily. In the prototype SAT solver, the semantics of a choice was wrapping both alternatives with a `push` and `pop` call, for the variational SMT solver, the semantics of a choice are extended to also capture the solver state and transmit a continuation over an asynchronous channel [67] to a worker thread. In order to ensure sound results, we exploit the monoidal design of variational models to ensure that the variational model is insensitive to the order results are produced from the base solver.

5.5 Research Plan

While I have made significant progress in pursuit of the aforementioned design goals. Several deliverables are still to be completed:

Optimizations based on nanopass compiler research In the worst case, a variational core will be evaluated $2^{|D|}$ times where $|D|$ is the number of unique dimensions in the query formula. Therefore, any optimizations that can simplify the variational core once are likely to have an observable impact on performance. We employ a nanopass compiler architecture to increase the flexibility of the approach, since varying background theories are likely to produce different optimizations. This work has not begun but is architected for in the prototype variational SMT solver. The proposed thesis will answer the following research questions:

1. Which optimizations produce a corresponding effect on runtime performance for both real world case studies and random SMT problems?

2. For any given optimization, what is the magnitude of the effect?

3. Is the effect on performance sensitive to the order optimizations are applied?

Evaluation of solver performance Previous work on the variational SAT solver [33] used two real-world case studies [23] from the SPL community for an empirical semantic. Thus, while these case studies are representative of practical use cases more evaluation could be done. The proposed thesis will perform a more robust evaluation and will consist of a mix of real-world and randomly generated data.

Specifically I will reuse the aforementioned case studies, both of which include SMT versions, and will use a forked version of a variational type checker, TypeChef [38], to log SAT problems produced by lexing, parsing and type checking the Linux kernel [40] and Busybox [39] open source projects. The forked version of TypeChef is complete but does not scale for the Linux kernel at time of this writing, and thus requires a refactor to use a more sophisticated logging infrastructure such as a database.

With this data, a convincing evaluation of the variational SMT solver architecture and design is possible. However, the majority of data will only exercise the solver for Boolean propositional formulas. To test the architecture in the SMT use case, the study will use the SMT versions of the previous case studies and will generate mixed [35] random SMT problems according to established methods in the random-SAT community.

This data serves several other purposes. It will be used to evaluate performance of the encoding strategies from item 1b on the prototype SMT solver. It will serve as the test data to assess the impact of sharing on solver performance. The non-random portion will be made publicly available as a dataset of real-world variational SAT and SMT problems, and as a dataset of related SAT problems, thus increasing the impact of this thesis.

Proof of variation preservation A proof of variation preservation is a key contribution of the thesis. This work is partially complete with a proof of progress for accumulation and evaluation, and a lemma that plain formulas always result in a $\bullet$ value in Agda. The strategy is to show progress and preservation over the inference rules deriving and solving a variational core. Implicit in this is to show variational preservation for constructing a VPL formula. There properties left to prove are as follows:

1. *Encoding preservation: An encoding algorithm preserves the uniqueness of variants.* In other words whichever encoding algorithm is used to synthesize the set of problems to VPL, that encoding algorithm does not lose variants.

2. *Variational core progress: Given a VPL formula a variational core is always derivable*

3. *Variational core preservation: For a set of C_2 problems, the corresponding variational core can recover that set of problems and if a problem in the initial set was satisfiable, then the corresponding variant is also satisfiable.* For any variational core, the set of SAT or SMT problems can be recovered by enumerating all variants with a total configuration and substituting symbolic terms with their concrete representations. The resulting formula should then be semantically equivalent to the original SAT or SMT problem. This item is a direct reproduction of the quick-check properties that verify the prototype SAT and SMT solver's as sound.

4. *Variational model Preservation: A variational model produces a satisfying assignment for every satisfiable variant.* For any variant, a variational model always results in a satisfying assignments to the corresponding SAT or SMT problem. Similar to the previous item this is the second half of

6 Related Work

Related work has been discussed throughout the previous sections. This section collects related work that was not previously covered. To my knowledge, this work is the first to translate the specific ideas of a nanopass architecture, and a variational compiler, to the SAT/SMT domains. Furthermore, at time of this writing this work is the first to investigate variational concurrency.

This work is most similar to [68], 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 uses heuristics to find canonical forms of sliced programs, and caches solver results on these canonical forms in a key-value store [69]. In contrast, variational SAT solving is domain agnostic, solves SAT problems expressed in VPL, returns a variational model, and uses incremental SAT solving.

Variational SAT solving is the latest in a line of work that uses the choice calculus to investigate variation as a computational phenomena. The choice calculus has been successfully applied to diverse areas of computer science, such as databases [70, 71], graphics [72], data structures [30, 73–75], type systems [76–79], error messages [78, 80–82], and now satisfiability solving. Our use of choices is similar to the concept of *facets* [83] and *faceted execution* [84–86], which have been successfully applied to information-flow security and policy-agnostic programming.

The use of compiler optimization techniques in SAT solvers is not novel, for example common subexpression elimination (CSE) and variable elimination has been successfully implemented in SAT solvers [87, 88]. Other pre-processing techniques such as removing blocked clauses [89], and detecting autarkies [90] have been very successful at automatically increasing performance of the SAT or SMT solver. From this perspective, our use of choices is a pre-processing technique to speed up incremental solving over sets of related problems.

Some solvers such as z3 [51], allow users to program the heuristics used by the solver to find choose efficient solving techniques, z3 in particular calls these constructs *strategies*. Strategies are thus similar to many optimizing compiler techniques [91], only applied to the SAT domain

7 Research plan

Sec. 1.1 lists the deliverables for this thesis. Each item has been discussed in more depth in previous sections. This section summarizes these items and provides an itemized list of each deliverable, broken down into discrete tasks, and their corresponding time estimations. The time estimates are guidelines with the real schedule being dictated by our publication schedule.

7.1 Encoding strategies of VPL formulae

The naive and naive-with-optimizations encoding algorithms are complete. The remaining work is a literature review for other representations of boolean formulae which may be useful, and to implement a Huffman based encoding algorithm and assess its performance:

1. Construct a list of alternative representations of boolean formulae which may lend itself to encoding strategies
2. Implement the Huffman coding algorithm over boolean formulae, recursively merge formulas into a VPL formula based on a similarity metric.

Estimated time to complete: 1-2 weeks

7.2 A method to determine the hardness of VPL formulae

The time consuming parts of this item: creating a random generator suitable for the analysis and creating a benchmark platform to test are complete. The only items remaining are to perform the analysis.

1. Review literature from random-acsat community on observing phase transition for both random k-SAT and mixed-SAT problems.
2. Replicate the transition detection analysis from item 1.

Estimated time to complete: 1 week

7.3 Variational SMT Solving

A prototype asynchronous variational SMT solver is complete and operational but several auxiliary tasks are still required. I expect this item to require the majority of time besides writing the the proposed thesis.

1. Implement an enhanced user interface which treats the solver like a server rather than a batch process.
2. Write the formal grammar for SMT-flavored VPL. The current formalization exists in Haskell code and thus must be translated into a publishable format.
3. Write the formalization for SMT-flavored variational models. An identical item to item 2 only with variational models.
4. Write the denotational semantics for SMT-flavored variational solving algorithm. An identical item to item 2 only for the solving algorithm.
5. Construct a proof for variation preserving semantics using the formalizations in items 2-4.
6. Review literature on nanopass compilers to construct a set of possible optimizations.
7. Implement optimizations from item 6
8. Implement configurations of the variational solver to provide an interface suitable for benchmarking.

Estimated time to complete 1-4: 1-2 weeks
Estimated time to complete item 5: 4 weeks (concurrent with other work)
Estimated time to complete items 6-7: 2 weeks
Total estimated time to complete: 6-8 weeks

7.4 Assembling Case Study Data

Most of the work left to assemble the case study data involves refactoring and running the forked TypeChef system on the Linux kernel; the Busybox case study dataset is finished. Parsers for the SPL case studies and the Busybox/Linux results are already complete and have been tested.

1. Implement a database caching system in TypeChef to record SAT problems at scale
2. Run the Linux case study to generate SAT problems.
3. Evaluate the Linux and Busybox case studies using encoding strategies from item 2.

Estimated time to complete item 1: 2-3 weeks (with help from collaborator Paul Maximillian Bittner)
Estimated time to complete item 2: 1 week
Estimated time to complete item 3: 1-2 weeks
Total estimated time: 4-6 weeks

8 Conclusion

SAT and SMT solvers are ubiquitous and powerful tools in computer science and software engineering. Incremental SAT and SMT provide an interface that supports solving many related problems efficiently. However, the interface could be automated and improved.

The goal of this thesis is to explore the design and architecture of a variational satisfiability solver that automates and improves on the incremental SAT interface. Through the application of the choice calculus the interface can be automated for satisfiability problems, the solver interaction can be formalized and made asynchronous, and the solver is able to directly express variation in a problem domain.

The thesis will present a complete approach to variational satisfiability and satisfiability modulo theory solving based on incremental solving. It will include a method to automatically encode a set of Boolean formulae into a variational propositional formula. A method for detecting the difficulty of solving such a variational propositional formula. A data set suitable for future research in the SAT, SPL and variation research communities. A variational satisfiability solver, an asynchronous variational satisfiable modulo theory solver and a proof of variational preservation.

References

[1] 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, NLD, 2009.

[2] 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.

[3] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, May 1999.

[4] João P. Marques Silva and Karem A. Sakallah. Grasp&mdash;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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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. Association for Computing Machinery.

[12] 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.

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

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

[15] 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.

[16] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Automated Reasoning on Feature Models. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*, pages 491–503, 2005.

[17] 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.

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

[19] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 643–654. ACM, 2016.

[20] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 1–13. ACM, 2018.

[21] 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. ACM, 2016.

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

[23] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. Anomaly Detection and Explanation in Context-Aware Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 18–21. ACM, 2017.

[24] 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. In *Proc. Europ. Conf. on Computer Systems (EuroSys)*, pages 47–60. ACM, 2011.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg, 2007.

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

[33] Jeffrey M. Young, Eric Walkingshaw, and Thomas Thüm. Variational satisfiability solving. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, SPLC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[34] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

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

[36] Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 343–350, New York, NY, USA, 2013. Association for Computing Machinery.

[37] Working Group 1. R7rs-small standard for the scheme programming language. `https://small.r7rs.org/`. Accessed: 2020-11-16.

[38] 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. ACM, 2010.

[39] BusyBox. `https://busybox.net/`. Accessed at October 27, 2020.

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

[41] 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.

[42] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan; Claypool, 2016.

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

[44] 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.

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

[46] 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.

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

[48] Co located with the Interaction Conference on Theory and Applications of Satisfiability Testing. International SAT Competition. `www.kernel.org`. Accessed at December 02, 2019.

[49] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[50] 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.

[51] 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.

[52] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[53] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[54] Rob DeLine and Rustan Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical report, March 2005.

[55] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[56] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[57] Wilfried Steiner. An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, page 375–384, USA, 2010. IEEE Computer Society.

[58] Rupak Majumdar, Indranil Saha, and Majid Zamani. Performance-aware scheduler synthesis for control systems. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, page 299–308, New York, NY, USA, 2011. Association for Computing Machinery.

[59] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 737–744, Cham, 2014. Springer International Publishing.

[60] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[61] University of Lugano: Formal Verification and Security Lab. openSMT. `http://verify.inf.usi.ch/opensmt`. Accessed at Nov 27th, 2020.

[62] Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. Technical Report MSR-TR-2009-121, September 2009. A conference version of this report appears in the proceedings of FMCAD 2009.

[63] J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[64] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970.

[65] Maurice P. Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Trans. Program. Lang. Syst.*, 4(4):527–551, October 1982.

[66] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[67] Simon Marlow. *Parallel and Concurrent Programming in Haskell*, pages 339–401. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[68] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM.

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

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

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

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

[80] 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.

[81] 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.

[82] 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.

[83] 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.

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

[85] K. Micinski, D. Darais, and T. Gilray. Abstracting faceted execution. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 184–198, 2020.

[86] 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.

[87] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, SAT'05, page 61–75, Berlin, Heidelberg, 2005. Springer-Verlag.

[88] Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically improving sat encoding of constraint problems through common subexpression elimination in savile row. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming*, pages 330–340, Cham, 2015. Springer International Publishing.

[89] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for cnf formulas. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, page 357–371, Berlin, Heidelberg, 2010. Springer-Verlag.

[90] Mark Liffiton and Karem Sakallah. Searching for autarkies to trim unsatisfiable clause sets. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008*, pages 182–195, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[91] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.