Variational Satisfiability Solving

PhD. Thesis

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

January 27, 2021

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

2 Background

3 Variational SMT Solving

[TODOs]

- [worked example]
- [Plain model alignment]
- [Proof read]
- [Claims align with introduction]

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

(a) Syntax of Integer arithmetic extension.

$$
\begin{array}{rcll}
t & ::= & r \mid \texttt{T} \mid \texttt{F} & \textit{Variables and Boolean literals} \\
\otimes & ::= & < \mid \leq \mid \geq \mid > \mid \equiv & \textit{Binary relations} \\
 & & & \\
f & ::= & t & \textit{Terminal} \\
 & \mid & \neg f & \textit{Negate} \\
 & \mid & f \vee f & \textit{Or} \\
 & \mid & f \wedge f & \textit{And} \\
 & \mid & ar \otimes ar & \textit{Integer comparisons} \\
 & \mid & D\langle f, f\rangle & \textit{Choice}
\end{array}
$$

(b) Syntax of extended VPL.

Figure 1: Formal definition of extended VPL.

In this section we describe an extension of variational satisfiability solving to variational satisfiability-modulo theories (SMT) solving. SMT solvers generalize satisfiability solving (SAT) solvers through the use of *background theories* that allow the solver to reason about values and constructs outside the Boolean domain. The SMTLIB2 standard defines seven such background theories: `Core` (Boolean theory), `ArraysEx`, `FixedSizeBitVectors`, `FloatingPoint`, `Ints`, `Reals`, and `Real_Ints`. In this section, we use integer arithmetic (`Ints`) as an example SMT extension for variational SMT solving. However, the technique illustrated here can be used to extend the approach to any theory that adds a regular language to the SMT domain. Extending the variational solving algorithm to context sensitive theories, such as `FixedSizeBitVectors` and `ArraysEx` are open research questions.

Not sure what "adds a regular language to the SMT domain" means. -E

Syntax Fig. 1a defines the syntax of the integer arithmetic extension, which consists of integer variables, integer literals, a set of standard operators, and choices. The sets of Boolean and arithmetic variables are disjoint, thus an expression such as $(s < 10) \wedge (s \vee p)$, where s occurs as both an integer and Boolean variable is disallowed. The syntax of the language prevents type errors and expressions that do not yield Boolean values. For example, $D\langle 1, 2\rangle \wedge p$ is syntactically invalid. Choices in the same dimension are synchronized across Boolean and arithmetic sub-expressions, for example, the expression $g = (A\langle 1, 2\rangle + j \geq 2) \vee (a \wedge A\langle c, d\rangle)$ represents two variants: $[\![g]\!]_{A_\text{T}} = (1 + j \geq 2) \vee (a \wedge c)$ and $[\![g]\!]_{A_\text{F}} = (2 + j \geq 2) \vee (a \wedge d)$.

Semantics Fig. 2 shows a set of primitive operations that the base solver is assumed to support. We use the nonterminals in the grammar as metavariables to range over operations. The naming scheme of the metavariables is to use U, B, and S to indicate unary, binary, and solver functions in the base solver domain, and use subscripts to indicate the argument type and (in the case of binary operations) result type of the

$U_{\mathbb{B}}$	::=	`Not`	*Boolean negation*
$U_{\mathbb{Z}}$	::=	`Negate`	*Negation*
	|	`Signum`	*Sign function*
	|	`Abs`	*Absolute Value*
$B_{\mathbb{BB}}$	::=	`And`	*Conjunction*
	|	`Or`	*Disjunction*
$B_{\mathbb{ZB}}$	::=	`LT`	*Less than*
	|	`GT`	*Greater than*
	|	`LTE`	*Less than Equal*
	|	`GTE`	*Greater than Equal*
	|	`Eqv`	*Equivalency*
$B_{\mathbb{ZZ}}$	::=	`Add`	*Addition*
	|	`Sub`	*Subtraction*
	|	`Mult`	*Multiplication*
	|	`Div`	*Division*
	|	`Mod`	*Modulus*
S	::=	`Assert`	*Assert a constraint*
	|	`Spawn`	*Create a new variable*
	|	`GetModel`	*Generate a model*

Figure 2: Assumed base solver primitive operations

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

$$\frac{r \notin dom(\Delta) \quad \texttt{Spawn}\,(\Delta, r_i) = (\Delta', s)}{(\Delta, r_i) \mapsto (\Delta', s)}\ \textsc{Ac-GenI}$$

$$\frac{\Delta(r) = s}{(\Delta, r) \mapsto (\Delta, s)}\ \textsc{Ac-Ref} \qquad \frac{\Delta(r_i) = s}{(\Delta, r_i) \mapsto (\Delta, s)}\ \textsc{Ac-RefI}$$

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

$$\frac{\texttt{Not}(\Delta, s) = (\Delta', s')}{(\Delta, \neg s) \mapsto (\Delta', s')}\ \textsc{Ac-BU} \qquad \frac{U_{\mathbb{Z}}(\Delta, s) = (\Delta', s')}{(\Delta, \dagger s) \mapsto (\Delta', s')}\ \textsc{Ac-IU}$$

$$\frac{B_{\mathbb{B}\mathbb{B}}(\Delta, s_1, s_2) = (\Delta', s')}{(\Delta, s_1 \otimes s_2) \mapsto (\Delta', s')}\ \textsc{Ac-SBinB}$$

$$\frac{B_{\mathbb{Z}\mathbb{Z}}(\Delta, s_1, s_2) = (\Delta', s')}{(\Delta, s_1 \oplus s_2) \mapsto (\Delta', s')}\ \textsc{Ac-SBinI}$$

$$\frac{B_{\mathbb{Z}\mathbb{B}}(\Delta, s_1, s_2) = (\Delta', s')}{(\Delta, s_1 \bowtie s_2) \mapsto (\Delta', s')}\ \textsc{Ac-SBinIB}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, s_1) \qquad (\Delta_1, v_2) \mapsto (\Delta', s_2)}{(\Delta, v_1 \otimes v_2) \mapsto (\Delta', s_1 \otimes s_2)}\ \textsc{Ac-BinB} \qquad \frac{(\Delta, v_1) \mapsto (\Delta_1, s_1) \qquad (\Delta_1, v_2) \mapsto (\Delta', s_2)}{(\Delta, v_1 \oplus v_2) \mapsto (\Delta', s_1 \oplus s_2)}\ \textsc{Ac-BinI}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, s_1) \qquad (\Delta_1, v_2) \mapsto (\Delta', s_2)}{(\Delta, v_1 \bowtie v_2) \mapsto (\Delta', s_1 \bowtie s_2)}\ \textsc{Ac-BinIB}$$

Figure 3: Accumulation inference rules

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

$$\frac{\texttt{GetModel}(\Theta) = m}{(\Theta, \bullet) \rightarrowtail m}\ \textsc{Ev-Model}$$

$$\frac{}{(\Theta, D\langle e_1, e_2\rangle) \rightarrowtail (\Theta, D\langle e_1, e_2\rangle)}\ \textsc{Ev-Chc}$$

$$\frac{\otimes = \texttt{And}}{(\Theta, \bullet \otimes v) \rightarrowtail (\Theta, v)}\ \textsc{Ev-UL} \qquad \frac{\otimes = \texttt{And}}{(\Theta, v \otimes \bullet) \rightarrowtail (\Theta, v)}\ \textsc{Ev-UR}$$

$$\frac{(\Delta, \neg v) \mapsto (\Delta', v')}{(\Gamma, \Delta, \neg v) \rightarrowtail (\Gamma, \Delta', v')}\ \textsc{Ev-BU} \qquad \frac{(\Delta, \dagger v) \mapsto (\Delta', v')}{(\Gamma, \Delta, \dagger v) \rightarrowtail (\Gamma, \Delta', v')}\ \textsc{Ev-IU}$$

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

$$\frac{\otimes \neq \texttt{And} \qquad (\Delta, v_1) \mapsto (\Delta', v_1') \qquad (\Delta', v_2) \mapsto (\Delta'', v_2')}{(\Gamma, \Delta, v_1 \otimes v_2) \rightarrowtail (\Gamma, \Delta'', v_1' \otimes v_2')}\ \textsc{Ev-AccB}$$

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

Figure 4: Evaluation inference rules

$$\frac{(\Theta, s) \rightarrowtail (\Theta', \bullet)}{(C, \Theta, vm, s) \Downarrow_i (C, \Theta', m, \bullet)}\ \textsc{Sym}$$

$$\frac{(\Theta, \bullet) \rightarrowtail m \quad \texttt{Combine}(vm, m) = vm_C}{(C, \Theta, vm, \bullet) \Downarrow_i (C, \Theta, vm_C, \bullet)}\ \textsc{Gen}$$

$$\frac{(D, \mathsf{true}) \in C \qquad (C, \Theta, vm, v \otimes \texttt{toIL}(e_1)) \Downarrow_i (C, \Theta, vm, v')}{(C, \Theta, vm, v \otimes D\langle e_1, e_2\rangle) \Downarrow_i (C, \Theta, vm, v')}\ \textsc{Cr-LB}$$

$$\frac{\begin{array}{c} D \notin dom(C) \qquad (C \cup \{(D, \mathsf{true})\}, \Theta, vm, v \bowtie \texttt{toIL}(e_1)) \Downarrow_{i+1} (C_l, \Theta_l, vm_l, \bullet) \\ (C \cup \{(D, \mathsf{false})\}, \Theta_l, vm_l, v \bowtie \texttt{toIL}(e_2) \Downarrow_{i+1} (C_r, \Theta_r, vm_r, \bullet) \end{array}}{(C, \Theta, vm, v \bowtie D\langle e_1, e_2\rangle) \Downarrow_i vm_C}\ \textsc{Cr-IB-ChcR}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, v_1') \qquad (\Delta, v_1) \mapsto (\Delta', v_2')}{(C, \Gamma, \Delta, vm, v_1 \otimes v_2) \Downarrow_i (C, \Gamma, \Delta', vm, v_1' \otimes v_2')}\ \textsc{Cr-BinB}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, v_1') \qquad (\Delta, v_1) \mapsto (\Delta', v_2')}{(C, \Gamma, \Delta, vm, v_1 \bowtie v_2) \Downarrow_i (C, \Gamma, \Delta', vm, v_1' \bowtie v_2')}\ \textsc{Cr-BinIB}$$

$$\frac{(\Delta, v_1) \mapsto (\Delta_1, v_1') \qquad (\Delta, v_1) \mapsto (\Delta', v_2')}{(C, \Gamma, \Delta, vm, v_1 \oplus v_2) \Downarrow_i (C, \Gamma, \Delta', vm, v_1' \oplus v_2')}\ \textsc{Cr-BinI}$$

Figure 5: Choice removal inference rules

operations they range over. For example, integer inequality is ranged over by $B_{\mathbb{ZB}}$, indicating that inequality is a binary function whose arguments are of type $\mathbb{Z}$ and whose result is of type $\mathbb{B}$. We do not show subscripts for S as these operations are run for their side effects in the base solver. {not sure if this is a good strategy, we would have to show models and a unit value for side effects if we didn't elide these though}.

{also not sure about the unary symbol, but I think making the variational level all symbols and using text for base solver level is a good idea, open to suggestions here} Similarly we define metavariables for these functions in the variational SMT domain. We use $\bowtie$ to represent binary relations over integers, $\oplus$ to represent arithmetic binary functions such as addition, $\dagger$ for unary arithmetic functions, and $\otimes$ to represent binary logical connectives. Thus a term such as $(i < j) \wedge a$ is represented as $B_{\mathbb{BB}}(B_{\mathbb{ZB}}(i, j), a)$ in the base solver domain, and $(i \bowtie j) \otimes a$ in the variational SMT solver domain.

Can we define these directly in the grammar in Fig. 6? I think this is confusing and overwhelming in its current form. -E

The previous mini-section is called *Semantics*, but it just introduces a set of primitive operations and some syntax for referring to them.

The rules in Figure 3 are passing the store to the primitive operations. Why this is makes sense when you read the explanation of the store, but the syntax is inconsistent with the explanation of how the metavariables are used in the previous paragraph.

Accumulation With primitive operations and metavariables defined we specify accumulation in Fig. 3. Since the metavariable s has two meanings: sub-trees of `IL` in the variational SMT domain and sequences of clauses in the base solver, we treat it as overloaded. An implementation of the rules would require a store which maps symbolics at the variational level to terms or sequences at the base solver level.

I only see one definition of s (in Sec. 4). Can we refactor to avoid this problem? -E

Accumulation is represented as a binary relation with $\mapsto$. The rules follow a simple pattern: `Ac-Chc` skips any choices, `Ac-Gen` and `Ac-Geni` provide a method to inject references into the symbolic domain, `Ac-Ref` and `Ac-Refi` cache references to ensure the same reference is mapped to the same symbolic, and the rest of the rules provide operations on symbolic terms, e.g., `Ac-SBinB`, or are congruence rules such as `Ac-BinI`. We elide rules which process formulas composed of constants such $(\mathtt{T} \wedge \mathtt{F})$ or $(1+2+3)$. In cases such as $1 + 2 < i$, constants are reduced and treated as references, thus this formula becomes $3 < i$ and is accumulated to $s_3 < i$.

Accumulation maintains a store, Δ, to track and cache symbolic terms. For example, given formula such as: $g = a \wedge (a \wedge b)$, `Ac-Gen` will spawn only two new references, one for a and one for b, and `Ac-Ref` ensures the same symbolic will represent the a reference. This will produce $g = s_a \wedge (s_a \wedge s_b)$, because we g contains two boolean connective `Ac-BinB` will be called twice beginning with the inner conjunction. `Ac-BinB` will combine s_a and s_b into a new symbolic s_{ab}, update the store to Δ'. The new store will include entries for both references *and* symbolic references, thus, in this example Δ' contains $a \rightarrow s_a$, $b \rightarrow s_b$, and $s_{ab} \rightarrow (s_a \wedge s_b)$. Finally `Ac-BinB` will repeat the last procedure on the outermost conjunction adding a new entry to the symbolic store.

Figure 4 is mixing up the syntax of the IL and the operations of the base solver. For example, several of the rules have conditions like $\otimes = \texttt{And}$, but `And` is not part of the syntax of the IL. Also, in cases like this, we should just use the corresponding operation directly in the conclusion of the rule rather than expressing it as a side condition.

Given $\Theta = (\Delta, \Gamma)$, the structure of the judgment is inconsistent since the tuple is often shown "inlined" in the parent tuple. I suspect it'd be simpler to just not introduce Θ at all, but if it is used, then when both Δ and Γ are needed, they should be shown as a tuple.

Evaluation Evaluation is defined in Fig. 4 as a relation of the form $(\Theta, \mathrm{v}) \rightarrowtail (\Theta, \mathrm{v})$, where $\Theta = (\Gamma, \Delta)$ and Γ represents the base solver state. The rules `EV-TM` and `Ev-Sym` push new clauses to the base solver using the primitive assert operation. `Ev-Model` calls for a plain model from the base solver, only once a variant is fully reduced to $\bullet$. `Ev-Chc` skips choices, `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-AccB` and `Ev-And` rules. While `Ev-And` is a straightforward congruence rule, `Ev-AccB` instead processes its arguments using accumulation ($\mapsto$). Disjunctions are a source of backtracking 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. This problem is repeated for inequalities as well. `Ev-AccIB` switches to accumulation as one side of an inequality cannot be processed without knowledge of the adjacent side. Thus, evaluation contains no rules for arithmetic.

{part of me wants to say: with these rules if we input a CNF formula then we'll get the incremental pattern that is desirable as stated in the background...not sure if this is the place though.}

Is the forgotten Δ_1 correct in the bottom three rules of Figure 5?

Choice removal Choice removal is defined in Fig. 5 as a relation between the evaluation/accumulation stores (Θ), the configuration (C), and terms in IL. Furthermore, we track the current variational model as part of the 4-tuple. The vast majority of rules are either commutative versions of the presented rules; such as `CR-RB` which is `CR-LB` but with a choice as the left child of $\otimes$, or the same rules over different operators, such as `CR-LIB` which is `CR-LB` only for $\bowtie$; thus we only present a subset.

The interesting rules are `Gen` and `Sym` which use evaluation to query for a plain model, and construct a new variational model through the `Combine` function. `CR-LB` ensure the property of synchronization; when a choice is observed as the right child of a boolean operator, and the dimension has a value in the configuration (in this case `true`), then the proper alternative (in this case the left alternative) of the choice is retrieved. `CR-IB-ChcR` removes choices when the choice is not present in the configuration. We present the version of `CR-IB-ChcR` for $\bowtie$; the same rule exists for $\otimes$, $\oplus$, and for choices as the left children of $\bowtie$. The assertion stack counter, i, is incremented indicating that all recursive processing occurs in a new `push`/`pop` context. Each configuration is updated to process both alternatives, `true` for the left and `false` for the right alternative. Both alternatives eventually conclude to a $\bullet$ and thus a variational model, which are combined to a final result.

The remaining rules are congruence rules that recursively call accumulation after a choice has been found, and new terms are introduced as the result of a replacing a choice with an alternative. Careful readers will recognize that the provided rules can easily become stuck. For example, given the formula $a \vee (b \leq D\langle p, q\rangle)$ the rules cannot further reduce the formula due to the disjunction and inequality, and the choice cannot be accumulated. What is required is to find the choice while storing the *context* around the choice. We leave this as an implementation detail, the prototype variational solvers utilize a Huet zipper [1] data structure to capture this context[1], searches the variational core until a choice is in the focus position, and then applies a choice removal rule such as `Cr-IB-ChcR` or `Cr-LB`.

Derivation of a Variational Core Consider the query formula $h = ((1 + 2 < (i - A\langle k, l\rangle)) \wedge a) \wedge (A\langle c, \neg b\rangle \vee b)$; derivation of the variational core h begins with evaluation and all stores Δ, Γ initialized to empty. When a sure inputs a vc the configuration, C, is initialized to it, otherwise C is initialized to empty.

[1] that the Huet zipper has been successful implies delimited continuations {cite} may be an alternative and efficient method to capture the context

`Ev-And` is the only applicable rule, matching $\otimes$ with $\wedge$ at the root of h. Thus, $v_1 = ((1 + 2 < (i - A\langle k, l\rangle)) \wedge a)$, and $v_2 = (A\langle c, \neg b\rangle \vee b)$. We traverse v_1 first, leading to a recursive application of `Ev-And`. We denote the recursive levels with a tick mark $'$, thus
$v_1' = (1 + 2 < (i - A\langle k, l\rangle))$ is the recursive left child and the right child is $v_2' = a$.

`Ev-AccIB` matches $\bowtie$ with the $<$ at the root of v_1' and switches to accumulation. v_2' is a terminal, will match `Ev-Tm`, be sent to the base solver, and replaced with $\bullet$. `Ev-Tm` updates Γ, recording the interaction and yields $(\Gamma_{v_2'}, \Delta, \bullet)$, where $\Gamma_{v_2'} = \{a\} \cup \Gamma$ as the result for v_2'.

Accumulation on v_1' matches $\bowtie$ to $<$, applying `Ev-AccIB` yields two recursive cases: $v_1'' = 1 + 2$; and $v_2'' = i - A\langle k, l\rangle$. {v1" will be turned into a constant but we don't show rules for constants because they aren't interesting, should we? We could also transform Ac-Ref and Ac-Refi to work on t and t_i. Thoughts?}. v_1'' will be preprocessed to the value 3, and accumulated to a symbolic with `Ac-Refi` yielding $(\Delta_{v_1''}, s_3)$ where $\Delta_{v_1''} = \{(3, s_3)\} \cup \Delta$. v_2'' is the interesting case. `Ac-BinI` will match $-$ at the root node, i will be accumulated to s_i with `Ac-Refi` and the choice is skipped with `Ac-Chc`. Hence we have $(\Delta_{v_2''}, s_i - A\langle k, l\rangle)$, where $\Delta_{v_2''} = (\{i, s_i\} \cup \Delta_{v_1''})$ as the result for v_2''. Note that the stores, Θ, are threaded through from the left child to the right child and thus can only monotonically increase until the query formula is processed.

With results for v_1'', v_2'', and v_2' the recursive calls can finally resolve. v_1' yields $(\Delta_{v_1'}, s_3 < s_i - A\langle k, l\rangle)$, where $\Delta_{v_1'} = \{(i, s_i), (3, s_3)\}$, v_2''s result only manipulated Γ and thus v_1's result is $(\Gamma_{v_2'}, \Delta_{v_1'}, (s_3 < s_i - A\langle k, l\rangle) \wedge \bullet)$, which can be further reduced by `Ev-UR` to
$(\Gamma_{v_2'}, \Delta_{v_1'}, (s_3 < s_i - A\langle k, l\rangle))$.

This process is repeated for $v_2 = (A\langle c, \neg b\rangle \vee b)$ with the final stores from processing v_1. The only rule that matches $\vee$ is `Ev-AccB`, thus v_2 is processed in accumulation. Accumulation matches on the disjunction and applies `Acc-BinB` with $v_1' = A\langle c, \neg b\rangle$ and $v_2' = b$. The choice, by `Ac-Chc`, is skipped over; b, by `Ac-Gen` will be converted to a symbolic s_b yielding $(\Gamma_{v_2}, \Delta_{v_2}, A\langle c, \neg b\rangle \vee s_b)$, where $(\Delta_{v_2} = \{(b, s_b)\} \cup \Delta_{v_1'})$, and $\Gamma_{v_2} = \{a\}$ as the result for v_2. With both v_1 and v_2 processed the variational core for h is found to be $h_{core} = (s_3 < (s_i - A\langle k, l\rangle)) \wedge (A\langle c, \neg b\rangle \vee s_b)$ with stores $\Gamma_{h_{core}} = \{(a)\}$, $\Delta_{h_{core}} = \{(b, s_b), (i, si), (3, s_3)\}$.

Solving the variational core Solving the variational core begins with choice removal and proceeds with recursive calls to evaluation and consequently accumulation. We assume an empty configuration for the remainder of the example because the vc case is a sub-case. The computation rules which remove choices, such as `Cr-LB`, and `Cr-IB-ChcR`, require a choice in the child node of a binary relation, however h_{core}'s immediate child nodes are binary relations themselves, $<$ on the left, and $\vee$ on the right. We use a zipper to manipulate the core such that a choice is in position for removal, while the remainder of the core is held in a context, a variational SAT solving may instead choose to migrate choices according to Boolean equivalency laws.

Assuming $A\langle k, l\rangle$ is found to be the focus, then the left version of `Cr-IB-ChcR`, `Cr-IB-ChcL` would apply. Clearly $D \notin C$, thus a recursive case for each alternative, beginning with the left alternative e_1, is performed. Several changes occur: the assertion stack is incremented; indicating a push for the next call to evaluation, the configuration mutates to account for the selection, and e_1 is translated into IL and replaces the choice, thereby introducing a *new* plain term: l. Thus, the recursive call for the left alternative is $(s_3 < (s_i - k)) \wedge (A\langle c, \neg b\rangle \vee s_b)$ where $C_L = \{(A, \mathsf{true})\}$, and $i_L = 1$. Similarly the right alternative is $(s_3 < (s_i - l)) \wedge (A\langle c, \neg b\rangle \vee s_b)$ with $C_R = \{(A, \mathsf{false})\}$, and $i_R = 1$.

With the choice removed the rules are no longer stuck. `Cr-BinB` will apply to both alternatives because their root node, $\wedge$ matches $\otimes$. We walk through the processing of the left alternative in detail, the right alternative follows the same procedure. `Cr-BinB` produces two calls to accumulation with $v_1 = (s_3 < (s_i - k))$, and $v_2 = A\langle c, \neg b\rangle \vee s_b$, v_2 is still stuck and will thus be returned, v_1 is no longer stuck will be

fully reduced to a symbolic term.

Accumulation will apply Ac-BinIB with $v'_1 = s_3$ and $v'_2 = s_i - k$. v'_1 is already accumulated and will be returned, Ac-BinI will be applied to v'_2, will translate k to a symbolic s_k via Ac-GenI, and update $\Delta_{h_{core}}$ to $\Delta_{h_L} = \{(k, s_k) \cup \Delta_{h_{core}}\}$. Thus we have v'_2 accumulated to $v'_2 = s_i - s_k$ which allows an application of the computation rule Ac-SBinI to produce a single symbolic, $v'_2 = s_{i-k}$ with $\Delta_{h_L} = \{(s_i - s_k, s_{i-k}), (k, s_k)\} \cup \Delta_{h_{core}}$. The recursion continues to unwind with the result of Ac-BinIB as $v'_1 = s_3 < s_{i-k}$, the rule Ac-SBinIB can be applied yielding the result for v_1 as $v_1 = s_{s_3 < s_{i-k}}$ with store $\Delta_{h_L} = \{(s_3 < s_{i-k}, s_{s_3 < s_{i-k}}), (s_i - s_k, s_{i-k}), (k, s_k)\} \cup \Delta_{h_{core}}$.

With v_1 accumulated we have a new variational core $s_{s_3 < s_{i-k}} \wedge (A\langle c, \neg b\rangle \vee s_b)$, only this time, depending on the alternative, C has enough information to configure A. Again, we must find a choice in the focus in order to proceed, once $A\langle c, \neg b\rangle$ is in focus Cr-RB (the right version of Cr-LB) will be applied. $A \in C_L$ and so the left alternative c will replace the choice for $s_{s_3 < s_{i-k}} \wedge (c \vee s_b)$. This formula will switch into accumulation due to Cr-BinB and be processed to a single symbolic similarly to $s_{s_3 < s_{i-k}}$. Once the symbolic has been created, the Sym rule calls evaluation which performs the assertion stack manipulation, writes the symbolic to the base solver. A model is generated with the Gen rule and combined with an empty variational model. With the model for the true variant of A the process backtracks to compute the false variant.

$i \to$ -1
$c \to 0$

$a \to$ T

C_{FF} = {(A, F), (B, F)} C_{FF} = {(A, F), (B, T)}

$i \to 0$
$c \to 1$

$i \to 0$
$c \to 0$

$b \to$ -10

C_{FT} = {(A, T), (B, F)} C_{TT} = {(A, T), (B, T)}

Figure 6: Possible plain models for variants of f. {correct these}

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 variational model must allow users to find satisfying values for a variant. The model must allow users to find all variants a variable has a particular value or range of values.

Furthermore, several useful properties of variational models should be maintained: The model is non-variational; thus the user does not need to understand the choice calculus to understand their results. The model produces results that can be fed into a plain SAT solver (or SMT solver in the extension). The model can be built incrementally and without regard to the ordering of results, because it forms a commutative monoid under $\vee$.

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 type as the result of a choice. For any variable in the model, we assume the type returned by the base solver is correct,

and store the satisfying value in a linked list constructed *if-statements*[2]. Specifically, we utilize the function $ite : \mathbb{B} \rightarrow T \rightarrow T$ from the SMTLIB2 standard to construct the list. All variables are initialized as undefined (*Und*) until a value is returned from the base solver for a variant. To ensure the correct value of a variable corresponds to the appropriate variant, we translate the configuration which determines the variant to a variation context, and place the appropriate value in the *then* branch. For example, a possible entry for j in the variational model of g would be $j \rightarrow (ite\ A\ 1\ (ite\ \neg A\ 0\ Und))$.

Fig. 6 show possible plain models for f with the corresponding variational SMT model display in Fig. 7. We've added line breaks to emphasize the branches the $then$ and $else$ branches of the ite SMTLIB2 primitive.

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

Figure 7: Variational model corresponding to the plain models in Fig. 6.

This formulation maintains the functional 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 were interested in the value of i in the $\{(A, \mathtt{T}), (B, \mathtt{T})\}$ 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 may employ a SMT solver, add vc_i as a constraint, and query for a model.

This maintains the desirable properties of variational SAT models while allowing any type specified in the SMTLIB2 standard. 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, the variational SMT model allows for duplicate values, if 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 in an implementation by tracking the values a variable has been assigned in all variants. However, we desire to keep variational models as simple as possible and therefore only present the minimum required machinery.

[2]Also called a church-encoded list

4 Related Work

5 Future Work

6 Conclusion

References

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