Towards a Verified CDCL SAT Solver in Lean

Arjun S. Bhamra Cameron C. Hoechst

Georgia Institute of Technology
School of Computer Science

Abstract

This project proposes a verified-by-construction SAT solver leveraging the Conflict-Driven Clause Learning (CDCL) algorithm written in the Lean proof assistant. We extend the canonical Davis–Putnam–Logemann–Loveland (DPLL) algorithm with the first unique implication point (1-UIP) clause learning framework, but forgo more complicated structures like the Two Watched Literal (2WL) scheme or restarts. Using Lean, we design effective data structures to represent our solver state, and have also provided several lemmas and theorems for various key intermediary invariants of our SAT solver, particularly related to termination conditions of the Boolean Constraint Propagation (BCP) and 1-UIP subroutines, which are critical for effective SAT solvers.

1 Introduction

Owing to the reductions that many complicated problems have to boolean satisfiability instances, SAT solvers have been used to great success in a plethora of domains, ranging from hardware verification [9] and formal verification [16] to program synthesis [19] and more. As these applications become more critical, it is imperative we can verify outputs of the solvers that we use to ensure correctness [2, 17]. Across software projects, there are a few key strategies to try, including fuzzing, model checking, and simple code reviews. These methods are less robust but allow for a high level of confidence with quicker iteration timelines. On the flipside, *formal verification* allows developers to *prove* the correctness of their code relative to an implementation and specification. One way of doing this is via interactive theorem provers (ITPs) or proof assistants such as Rocq and Lean [1, 13], which leverage specially designed compilers to verify user-defined theorems about data structures and operations in addition to standard type-correctness guarantees.

Within the realm of verified SAT solvers, there are two main techniques that have seen practical application, the *trust-but-verify* and ITP verified approaches. In the trust-but-verify framework, one allows their SAT solver itself to be unverified, and at output time generates either a satisfying assignment or a claim of UNSAT with a *proof trace* - a certificate of the solver's correctness through a derivation of the UNSAT result. Proof traces can take on a number of formats, but the most common are Deletion-Resolution Asymmetric Tautologies (DRAT) [10] and Linear RAT (LRAT) [4]. After a solver generates a proof trace, the trace is passed through a formally verified proof checker, which is typically easier to write owing to its simpler design and implementation relative to a full SAT solver, especially a CDCL solver. Most practical verified SAT solvers use the trust-but-verify strategy primarily because it removes the burden of verifying complicated techniques such as Learn, Restart, and Forget semantics, as well as the 2WL scheme, both of which are crucial for scalable solver performance but incredibly challenging to prove [2, 7]. Instead, they can leverage develop inherently fast solvers and simply add on proof trace generators, for use in verification post-termination; the correctness doesn't live in the solver. A key example is CreuSAT, which is a Rust-based SAT solver that uses the Creusot verification framework on traces; it is among the fastest verified solvers [18].

On the other hand, there are a series of projects that implement formally verified SAT solvers via ITPs/proof assistants; notable examples include IsaSAT by Fleury *et al.* [2, 7] and `versat` [14]. SAT solvers verified in this manner focus on building up verified programs via *invariants*, rules of data structures or subroutines that must hold for the lifetime of their usage, in a specified context. In *A verified SAT solver with watched literals using imperative HOL*, for example, cites the key 2WL invariant as "Unless a conflict has been found, a watched literal may be false only if the other watched literal is true and all the unwatched literals are false" [7]. This theorem, written in Isabell HOL, was built up from several other intermediary lemmas.

Our contribution to the space of verified SAT solvers is a partially verified CDCL SAT solver written in the Lean proof assistant. To our knowledge, this is the first CDCL SAT solver written in Lean, although the SATurn project [8] by Siddhartha Gadgil presents a fully verified DPLL SAT solver. The main invariants we prove are related to the monotonicity of our assignment trail, the termination of our BCP subroutine, and the termination of our 1-UIP subroutine, with the eventual goal of proving the no-relearning theorem from [2]. Simply put, this theorem states "No clause can be relearned twice"; we can prove this as humans quite

CS 8803-SAT, December 3, 2025, Georgia Institute of Technology, Atlanta, GA.

ACM ISBN 979-8-0000-0000-0/00/00…$15.00
https://doi.org/10.1145/0000000000

naturally via contradiction and by invoking definitions from standard CDCL calculi, but formally verifying this for our implementation has turned out to be challenging, and we are currently still in the process of verifying termination for BCP and 1-UIP.

2 Background

2.1 The DPLL Algorithm

Below is pseudocode for the DPLL algorithm [5], which relies on BCP and a few key subroutines to function effectively.

```
fn DPLL(phi: Formula) bool {
  phi_prime = BCP(phi); // resolution
  if (phi_prime = True) return SAT;
  else if (phi_prime = False) return UNSAT;
  pivot = decide(phi_prime); // choose var as pivot
  if (DPLL(phi_prime[pivot = True])) return SAT;
  else return (DPLL(phi_prime[pivot = False]));
}
```

The general idea of DPLL is to use BCP to reduce the formula as much as possible, by deriving constraints via the Resolution rule. Then, we can use a decision heuristic like VSIDS [12] to pick a variable to be a good pivot candidate for further resolution steps. In this way, DPLL combines search and deduction techniques in an effective way. However, DPLL has a few limitations that necessitated iteration, namely:

1. DPLL does not learn new clauses: CDCL is known for its clause learning primarily through the 1-UIP framework, which allows for additional constraints on variables to be learnt. DPLL does not have this capability.
2. At a conflict, DPLL simply backtracks up to the previous set of assignments and tries the alternate assignment to see if that maintains (partial) satisfiability; with 1-UIP, CDCL can leverage *non-chronological backtracking* to undo multiple assignments.
3. Owing to learned clauses, DPLL implements *tree-like resolution* whereas CDCL implements *general resolution*; this is a significant gap that allows CDCL to have shorter proofs for a larger class of input formulas.
4. While DPLL doesn't typically use advanced decision heuristics, CDCL regularly leverages subroutines such as VSIDS or LRB to optimally pick pivots.

2.2 Modifications for CDCL

The crucial modifications we introduced in our CDCL SAT solver are the 1-UIP learning scheme and the VSIDS decision heuristic. We describe both of these below.

2.2.1 First Unique Implication Point (1-UIP)

A *unique implication point* is any node at the current decision level such that any path from the decision variable to a conflict node must pass through it. There can be many unique implication points in an implication graph, but we care about the first UIP - the earliest cut that satisfies the criteria. The clause learned by 1-UIP also determines the decision level a solver should backjump to (the second highest decision level in the learnt clause), and 1-UIP works effectively in this regard. 1-UIP works as follows:

1. Start from conflict clause, set the "curr" clause to be the negation of all literals in the clause. For example, with conflict = $(\neg x_1 \vee x_2)$, curr becomes $(x_1 \vee \neg x_2)$
2. In the current clause c, find the last assigned literal l
3. Pick an incoming edge to l (a clause c' that contains literal l)
4. Resolve `curr` and c'
5. Set `curr = resolve(curr,` c'`)`
6. Repeat until only one literal in curr at current decision level
7. Done!

3 Our work: a CDCL SAT Solver in Lean

3.1 Data Structures

Our work first builds off of a series of key data structures, beginning with variables and literals in boolean formulae. We choose to represent these simply as `Var := Nat` and `Lit := Int`, with the conditions that both are nonzero, and `Lit`s must be negative when representing a negated variable. SATurn's construction of clauses is interesting, treating variables as `Nat`s in $\{0, \ldots, n\}$ and then having clauses be a finite sequence of length n with values in `Option Bool` [8] . In contrast, we define clauses as follows:

```
structure Clause where
  lits    : Array Lit
  learnt  : Bool := false -- default
  deriving Repr, Inhabited
```

This allows for using indexing to access literals instead of working through the lens of functions, which is typically more efficient. In an analogous manner, we define `Formula`s as vectors of clauses and `Assignment`s as vectors of `AssignState`s, both bounded by the number of clauses. Another crucial data structure is the clause database, which stores all existing (from the formula) and new (learnt) clauses:

```
structure ClauseDB (nc : Nat) where
  clauses : Vector Clause nc -- indices >= #vars -> learnt
clauses.
  num_unassigned : Vector Nat nc :=
    clauses.map (λ c => c.lits.size)
  deriving Repr
```

The two final data structures of import are the Assignment Trail and proof traces. For the Assignment Trail, we build up from a stack and specialize it. Our stack is built up from a clean recursive structure, generic over any type α:

```
inductive Stack (α : Type) where
  | empty : Stack α
  | push : α → Stack α → Stack α
  deriving Repr
```

```
def push {α : Type} (x : α) (s : Stack α) : Stack α :=
  Stack.push x s

def Stack.top {α : Type} : Stack α → Option α
  | empty => none
  | push x _ => some x

-- Other helpers like pop, isEmpty, size, etc.
```

From here, the `AssignmentTrail`'s definition is clear:

```
structure AssignmentTrail where
  stack : Stack (Lit × Nat) := Stack.empty
```

The trail stores both the literal and its associated decision level in the stack, as appropriate.

Finally, in order to provide proper proofs of UNSAT, we need to design a structure that can store a path for derivations leading to a negative conclusion - the natural choice is a recursive resolution tree.

```
inductive ResolutionTree where
  /- Leaves are clauses from the original formula, we only
     start with leaves and build up conflict clauses + our
     resolution tree from there
  -/
  | leaf    (clauseIdx : Nat)
  | resolve (pivotVar  : Var)
            (pivotSign : Bool) -- T => l has piv, r has ¬piv
            (left      : ResolutionTree)
            (right     : ResolutionTree)
```

The `ResolutionTree` stores the exact resolution steps used to reach any conclusion, and when working in tandem with our solver, provides an accurate trace to prove unsatisfiability of a given input formula. Across the board, our design focuses on using `Array`s or bounded `Vector`s for O(1) index-based access, and recursive structures for easier proving.

3.2 Tying it all together: The SAT Solver

We parameterize the solver over the number of variables `nv` and number of clauses currently in the `ClauseDB nc` in the unknown state to simplify proofs relating to indexing validity/bounds checking.

```
structure Solver (nv nc : Nat) where
  clauses       : ClauseDB nc
  assignment    : Assignment nv
  decision_lvl  : Nat := 0
  trail         : AssignmentTrail
  -- Stores indices to clauses in ClauseDB
  is_satisfied  : Vector Bool nc := Vector.replicate nc
false
  -- How many clauses are still in the "unknown" state?
  contingent_ct : Nat := nc
  -- Stores clause whose unit status led to propagation.
prop_reason[n] = m → n is forced to ⊤/⊥ because of clause
m.
  prop_reason   : Vector (Option Nat) nv := Vector.replicate
nv none
  activity      : VsidsActivity
  -- rtree      : ResolutionTree
```

This is in contrast to a previous iteration which didn't contain the numerical bounds as a part of the type, and thus required separate lemmas for bounds checking when accessing clauses through the `ClauseDB` or variables in each clause. The rest of the solver works as described via coursework; there is a BCP invocation, followed by an `analyzeConflict` subroutine that updates activities for the Variable State Independent Decaying Sum (VSIDS) decision heuristic and then uses the 1-UIP framework to learn a new conflict clause, before determining the backjump level. If we know the backjump level is 0, we have reached a contradiction at the root of the formula and we return UNSAT with our `ResolutionTree`. Otherwise, we can proceed as normal after rolling back our assignments via the `backjump` function. Notably, we include the `contingent_ct` to aid our termination proof for BCP, which will be discussed in §3.3.

3.3 Proving BCP and 1-UIP

3.3.1 BCP Proofs

It is relatively straightforward to argue the soundness of BCP. We already know that unit resolution is a sound proof rule, so we say an implementation of BCP is sound if it always eventually terminates and correctly characterizes a formula as SAT, UNSAT, or ambiguous based on the number of clauses that are satisfied and the existence of any "empty" clauses. However, if one cannot prove that an algorithm terminates, no other function or proof that depends on any value it produces is provably sound.

At a first brush, proving the termination of BCP appears straightforward. The number of iterations it requires is upper-bounded by the number of clauses in the formula. However, BCP will continue if it discovers another set of unit clauses

Moreover, proof assistants, do not trust such high-level descriptions, and require evidence that a term becomes smaller with each iteration of a recursive function.

We chose to formalize our termination condition for BCP using a variable in the solver called `contingent_ct`, which counts the total number of clauses whose status is still unknown in the formula. Taking inspiration from 2WL (but with more predictable behavior), the solver also maintains the number of unassigned literals in each clause (see Listing 1) to quickly determine if a clause is unit or in conflict.

Lean uniquely allows the programmer to specify any function of any value passed into a definition as the measure for termination using the `termination_by` clause, which allows for simpler, more direct implementations.

3.3.1.1 First Attempt: Two nested folds

To make proving termination easier, we also chose to implement the propagation loops using `foldl` because it's bounded by the number of elements in the array. We chose to split our implementation into three main functions: `propNonUnit`, which propagates a variable within a particular unit clause, `propOne`, which calls `propNonUnit` on all non-unit

clauses for a particular unit clause, and `propUnits` which applies `propOne` to the state of the solver for each unit clause.

`propNonUnit` propagates the effects of making the literal `lit` true to all other clauses that contain both it and its negation. It returns the state of the solver after propagation in addition to returning the clauses that are now unit after propagation. By repeatedly applying `propNonUnit`, `propOne` accumulates all newly unit clauses

```
abbrev PropTriple (nv nc : Nat) := Solver nv nc × Array (Fin
nc) × Array (Fin nc)
def ptContingentCt : PropTriple nv nc → Nat :=
(Solver.contingent_ct ∘ (·.1))

def propNonUnit (lit : Lit) (in_prop : PropTriple nv nc)
(uci : Fin nc) : PropTriple nv nc :=
  let (s', units', non_uc') := in_prop
  let prop_clause := s'.clauses.clauses[uci]
  if ¬(s'.is_satisfied[uci])
  then if prop_clause.lits.contains lit
    then ({ s' with is_satisfied := s'.is_satisfied.set uci
true
                    contingent_ct := s'.contingent_ct - 1
    }, units', non_uc'.push uci)
    else if prop_clause.lits.contains (-lit)
      then
        let s' : Solver nv nc := { s' with clauses :=
s'.clauses.propLit uci }
        let (units', non_uc) := if
s'.clauses.num_unassigned[uci] = 1
          then (units'.push uci, non_uc')
          else (units', non_uc'.push uci)
        (s', units', non_uc)
      else (s', units', non_uc'.push uci)
  else (s', units', non_uc')

def propOne (in_prop : PropTriple nv nc) (uci : Fin nc) :
PropTriple nv nc :=
  let (s, units', non_uc') := in_prop
  let uc := s.clauses.clauses[uci]
  let lit := (uc.lits.find? (λ (l : Lit) =>
¬(s.assignment.isAssigned l.var))).get!
  let s := { s with
    assignment := s.assignment.assign lit.var (lit > 0)
    is_satisfied := s.is_satisfied.set uci true -- It's a
unit clause, so it's saitsfied!
    contingent_ct := s.contingent_ct - 1
    ...
  }
  -- Now we can just scan over the clauses that we know
aren't unit.
  non_uc'.foldl (propNonUnit lit) (s, #[], #[])
```

We first establish that one call to `propNonUnit` does not increase `contingent_ct`:

```
-- The proof of this is not particularly interesting.
lemma propNonUnit_leq (lit : Lit) (pt : PropTriple nv nc) :
    ∀ uci, (propNonUnit lit pt uci).fst.contingent_ct ≤
pt.fst.contingent_ct
```

Then, we argue that one application of `propOne` decreases `contingent_ct`:

```
lemma propOne_lt (pt : PropTriple nv nc) {hcz :
pt.fst.contingent_ct > 0} :
    ∀ uci, (propOne pt uci).fst.contingent_ct <
pt.fst.contingent_ct := by
    ... -- unpacking preamble
    have hcm : s.contingent_ct = s'.contingent_ct - 1 := rfl
    have hc : s.contingent_ct < s'.contingent_ct := by omega
    let s'' := (Array.foldl (propNonUnit lit) (s, #[], #[])
non_uc').fst
    have hleq : s''.contingent_ct ≤ s.contingent_ct :=
      Array.foldl_leq_monotone non_uc' (propNonUnit lit) (s,
#[], #[]) ptContingentCt (propNonUnit_leq lit)
    subst s'' s
    omega
```

Lean's standard library graciously provides a nice lemma for proving inductive properties on folding over arrays, which we leverage to argue that if a function does not increase a value accessed through `get_nat`, then folding that function on an array also does not increase this value.

```
theorem Array.foldl_leq_monotone
    {α β : Type}
    (as : Array α)
    (f : β → α → β)
    (init : β)
    (get_nat : β → Nat)
    (hleq : ∀ (b : β) (a : α), get_nat (f b a) ≤ get_nat
b) :
    get_nat (as.foldl f init) ≤ get_nat init := by
  let motive (_ : Nat) (acc : β) := get_nat acc ≤ get_nat
init
  have h0 : motive 0 init := by
    unfold motive
    simp
  have hf : ∀ (i : Fin as.size) (acc : β), motive (↑i) acc →
motive (↑i + 1) (f acc as[i]) := by
    intros i acc ih
    unfold motive
    unfold motive at ih
    have iha : get_nat (f acc as[i]) ≤ get_nat acc := hleq
acc as[i]
    omega
  apply Array.foldl_induction motive h0 hf
```

We then attempt to prove the termination of `propOne` in much the same way as `propNonUnits`, with some minor adjustments to handle the edge-case of folding a strictly decreasing function on an empty list in a helper lemma.

```
theorem Array.foldl_lt_monotone
    ... -- Same signature as before except:
    (hnz : as.size > 0)
    -- Make note of this assumption!
    (hlt : ∀ (b : β) (a : α), get_nat (f b a) < get_nat b) :
    get_nat (as.foldl f init) < get_nat init
  := by
  -- < is just a special case of <=
  have hleq : ∀ (b : β) (a : α), get_nat (f b a) ≤ get_nat
b := λ b a => Nat.le_of_lt (hlt b a)
  have h1 : get_nat (f init as[0]) < get_nat init := hlt
init as[0]
  have hs : get_nat (Array.foldl f (f init as[0]) as[1:]) ≤
get_nat (f init as[0]) :=
    Array.foldl_leq_monotone as[1:] f (f init as[0]) get_nat
hleq
  -- Omitted: Fanangling to relate the subarrays and first
element of `as` with `as`.
```

```
  omega

lemma propOne_lt (pt : PropTriple nv nc) {hcz :
pt.fst.contingent_ct > 0} :
    ∀ uci, (propOne pt uci).fst.contingent_ct <
pt.fst.contingent_ct := by
    ...
    have hcm : s.contingent_ct = s'.contingent_ct - 1 := rfl
    have hc : s.contingent_ct < s'.contingent_ct := by omega
    let s'' := (Array.foldl (propNonUnit lit) (s, #[], #[])
non_uc').fst
    have hleq : s''.contingent_ct ≤ s.contingent_ct :=
      Array.foldl_leq_monotone non_uc' (propNonUnit lit) (s,
#[], #[]) ptContingentCt (propNonUnit_leq lit)
    subst s'' s
    ...
```

But when we finally try to prove that `propUnits` terminates, we hit a snag:

```
def propUnits (in_prop : PropTriple nv nc) (hc :
in_prop.fst.contingent_ct > 0) (huc :
¬in_prop.snd.fst.isEmpty) : BCPResult nv nc :=
  let (s, uc_inds, non_uc) := in_prop
  let (s', uc_inds', non_uc') := uc_inds.foldl propOne (s,
#[], #[])

  have hcc : s'.contingent_ct < s.contingent_ct := by
    subst s'
    unfold out_prop
    -- FAILS TO TYPECHECK !!!
    apply Array.foldl_lt_monotone uc_inds propOne (s, #[],
#[]) ptContingentCt h_uc_nonzero (propOne_lt (hcz := hc))
    ... (continue if you still have unit literals)
  termination_by (in_prop.fst.contingent_ct)
```

In order to argue that calls to this function strictly decrease, we need evidence that `contingent_ct` is currently greater than zero:

```
propOne_lt (pt : PropTriple nv nc) {hcz :
pt.fst.contingent_ct > 0} : ...
```

...but the type of `hcz` *depends* on the particular value for `pt`! You can only decrease the count in a particular state of the solver that has a non-zero count! Changing the type of `hlt` makes the type significantly more difficult to work with, but even worse is that a solver with `contingent_ct = 0` is a valid state that we want the function to produce! This led us to refactor our implementation into...

3.3.1.2 Second Approach: Three-way split, avoid `lt_monotone`

We chose to refactor our implementation based on the following insight: given a function f that conditionally decreases a value, folding f over an array with some state also decreases the value if you can show at least one of its calls satisfies the condition. Hence `propOne2` is split into two variants: one that requires that `s.contingent_ct > 0` and provably decreases it, and one that does not require `s.contingent_ct > 0` and can only be proven to not increase it:

```
def propOne2InnerHaveCt {nv nc : Nat} (uci : Fin nc) (pi :
PropInfo nv nc) (hc0 : pi.s.contingent_ct > 0) : PropInfo nv
nc :=
    -- This implementation is much slower than the foldl one
    -- but has some macro-level properties that make it much
nicer to prove.
    let lit := (pi.s.clauses.clauses[uci].lits.find? (λ (l :
Lit) => ¬(pi.s.assignment.isAssigned l.var))).get!
    let s := satisfyUnit pi.s uci
    have hs : s.contingent_ct < pi.s.contingent_ct :=
satisfyUnit_decr_contingent_ct pi.s uci (hc := hc0)

    let pos_props := pi.two_plus.filter (λ tpi => !
s.is_satisfied[tpi] ∧ s.clauses.clauses[tpi].lits.contains
lit)
    let is_satisfied' := pos_props.foldl (λ acc sat_i =>
acc.set sat_i true) s.is_satisfied
    let contingent_ct' := s.contingent_ct - pos_props.size

    let neg_props := pi.two_plus.filter (λ tpi =>
s.clauses.clauses[tpi].lits.contains (-lit))
    let s' := neg_props.foldl (λ acc prop_i => { acc with
clauses := acc.clauses.propLit prop_i }) s
    let (unsat', units', two_plus') := categorizeClauses s'
    let contingent_ct'' := contingent_ct' - unsat'.size

    have hct : contingent_ct'' < pi.s.contingent_ct := by
omega
    { s := { s' with
             is_satisfied := is_satisfied'
             contingent_ct := contingent_ct''
           }
      unsat := unsat'
      units := units'
      two_plus := two_plus'
    }

def propOne2Inner {nv nc : Nat} (pi : PropInfo nv nc) (uci :
Fin nc) : PropInfo nv nc :=
    -- ...same as `HaveCt` variant
    have : s.contingent_ct ≤ pi.s.contingent_ct := by
      have : s.contingent_ct = pi.s.contingent_ct - 1 := rfl
      omega
    -- ...same as `HaveCt` variant
    have hct : contingent_ct'' ≤ pi.s.contingent_ct := by
omega
    {s := ...} -- ...same as `HaveCt` variant

lemma propOne2InnerHaveCt_decr {nv nc : Nat} (uci : Fin nc)
(pi : PropInfo nv nc) (hc0 : pi.s.contingent_ct > 0) :
    (propOne2InnerHaveCt uci pi hc0).s.contingent_ct <
pi.s.contingent_ct := ...

lemma propOne2Inner_monotone {nv nc : Nat} (pi : PropInfo nv
nc) (uci : Fin nc) :
    (propOne2Inner pi uci).s.contingent_ct ≤
pi.s.contingent_ct := ...
```

Then, we define `propOne2` as a function that simply assumes that `s.contingent_ct > 0`, calls the decreasing version upfront, and then folds the monotonic version on the rest of the array, leveraging the `foldl_leq_montone` theorem from earlier.

```
def propOne2 {nv nc : Nat} (s : Solver nv nc) (units : Array
(Fin nc)) (hsz : units.size > 0) (hc : s.contingent_ct >
0) : PropInfo nv nc :=
  have hnempty : units.toList ≠ [] := by
    simp
    apply Array.ne_empty_of_size_pos
    exact hsz
  let units := units.toList
  let hd := units.head hnempty
  let tl := units.tail
  let pi' := propOne2InnerHaveCt hd (PropInfo.empty s) hc
```

```
  tl.toArray.foldl propOne2Inner pi'

lemma propOne2_decr {nv nc : Nat} (s : Solver nv nc)
(units : Array (Fin nc)) (hsz : units.size > 0) (hc :
s.contingent_ct > 0) :
    (propOne2 s units hsz hc).s.contingent_ct <
s.contingent_ct := by
  unfold propOne2
  extract_lets hnempty units' hd tl pi'
  have h1 : pi'.s.contingent_ct < s.contingent_ct := by
    subst pi'
    let pi := (PropInfo.empty s)
    have hpi : pi.s.contingent_ct > 0 := by
      subst pi
      unfold PropInfo.empty
      omega
    apply propOne2InnerHaveCt_decr hd pi hpi
  have hmono : (tl.toArray.foldl propOne2Inner
pi').s.contingent_ct ≤ pi'.s.contingent_ct :=
Array.foldl_leq_monotone tl.toArray propOne2Inner
pi' (·.s.contingent_ct) propOne2Inner_monotone
  omega
```

Finally, we conditionally call the propagation logic and recur only if there are any unit and two-plus clauses remaining, both of which are considered "contingent" before propagation:

```
def propUnits2 (s : Solver nv nc) (unsat : Array (Fin nc))
(units : Array (Fin nc)) (two_plus : Array (Fin nc)) :
BCPResult nv nc :=
  if hs : s.contingent_ct = 0
    then .ok s -- Successfully propagated through every
clause -> you are done.
    else if have_unsat : unsat.size > 0
      then .error (s, s.clauses.clauses[unsat[0]])
      else if only_units : two_plus.size = 0
        then
          let s := units.foldl satisfyUnit s
          .ok s
        else if no_units : units.size = 0
          then .ok s -- Done, no unit clauses to propagate.
          else
            have hs : s.contingent_ct > 0 :=
(Nat.ne_zero_iff_zero_lt.mp hs)
            let pi' := propOne2 s units
(Nat.ne_zero_iff_zero_lt.mp no_units) hs
            have : pi'.s.contingent_ct < s.contingent_ct :=
propOne2_decr s units (Nat.ne_zero_iff_zero_lt.mp no_units)
hs
            propUnits2 pi'.s pi'.unsat pi'.units
pi'.two_plus
  termination_by (s.contingent_ct)

def bcp {nv nc : Nat} (s : Solver nv nc) : BCPResult nv
nc :=
  let (unsat, units, two_plus) := categorizeClauses s
  propUnits2 s unsat units two_plus
```

While exploring the space of possible implementations, we chose to avoid the two-watched literal scheme because proving its soundness was out-of-scope for the project, and would have been time consuming [7]. Even though proving the overall soundness of this scheme is difficult, we may be able to argue that it terminates with minimal changes to our proofs if Lean's termination check has evidence that `propUnits2` will not increase the number of watched literals in the solver and observe that in the case where the count does not decrease, the function immediately returns. An immediate follow-up we could try is to implement 2WL and argue termination by the total number of watched literals in the solver state, with the additional condition to return early if propagating did not reduce the total number of watched literals.

3.3.2 1-UIP Proofs

For the 1-UIP clause learning framework, we design a recursive algorithm that implements the overall iteration loop given by §2.2.1, with some things still in progress. This is a snippet of the `learn` function[1], focusing on the recursive loop:

```
  let rec loop (s : Solver nv nc) (curr : Clause) (seen :
Std.HashSet Nat)
    (h_curr_assigned : ∀ l ∈ curr.lits, containsVar l.var
s.trail.stack = true) : (Solver nv nc × Clause) :=
    let lits_at_dl :=
      curr.lits.filter (fun (l : Lit) =>
        let var_dl := AssignmentTrail.dlOfVar s.trail l.var
|>.getD 0 -- default 0 else var_dl
        var_dl = dl)
    if lits_at_dl.size == 1 then (s, curr) else
      -- need to show that for all clauses found by
pickIncomingEdge, last assigned variable is in trail
      -- and last assigned in curr\{last_assigned_lit} is
also in trail
      let last_assigned_lit := -
AssignmentTrail.findLastAssigned s.trail curr
      -- pick incoming edge
      let (clause_that_implied, clause_idx) :=
pickIncomingEdge s last_assigned_lit seenClauses
      -- resolve clause_that_implied and curr
      let curr := resolveOnVar curr clause_that_implied
last_assigned_lit.var
      let seenClauses := seenClauses.insert clause_idx

      -- update trail
      -- NOTE: Do we know that last_assigned_lit is in
s.trail before the pop?
      let s' : Solver nv nc := { s with trail :=
s.trail.popVar last_assigned_lit.var }

      have : s'.trail.size < s.trail.size := by
        simp only [s']
        apply AssignmentTrail.popVar_size_lt_containsVar

      have h_curr'_assigned : ∀ l ∈ curr.lits, containsVar
l.var s'.trail.stack = true := by
        intro l hl
        -- l came from either curr or clause_that_implied
but not last_assigned_lit.var, since
        -- it was resolved over
        -- both had all vars in s.trail (prove)
        -- we only removed last_assigned_lit
        -- so l.var is still in s.trail (prove)
        sorry
```

[1] https://github.com/abhamra/verified-CDCL-datastructures/blob/main/VerifiedCdclDatastructures/Solver.lean#L296-L371

```
      loop s' curr seen h_curr'_assigned
  termination_by s.trail.size
```

The key invariant we have proven thus far, via intermediary results, etc., is that our `AssignmentTrail`'s `popVar` function either decreases or maintains the size of the trail. More formally, this is the implementation of `popVar` on the stack/trail presented in §3.1:

```
-- takes stack, var (nat), pops literal referred to by var
def popVar (t : AssignmentTrail) (v : CDCL.Var) :
AssignmentTrail :=
  let rec loop (s acc : Stack (CDCL.Lit × Nat)) : Stack
(CDCL.Lit × Nat) :=
  match s with
  | Stack.empty => acc -- didn't find var, return
accumulated
  | Stack.push (lit, dl) rest =>
    if lit.var == v then
      Stack.pushAll acc rest
    else
      loop rest (Stack.push (lit, dl) acc)
  { t with stack := loop t.stack Stack.empty }
```

This function works by ignoring variables that aren't the one we search for, and if it finds a matching variable, will concatenate the previous section with the latter section, effectively removing the selected variable. By inspection, we can clearly see that this is true by cases, so we apply a similar technique to the proofs.

```
-- popVar's loop size <= the input loop size (either -1 or
stays same)
lemma loop_size (v : CDCL.Var) : ∀ (s acc : Stack (CDCL.Lit
× Nat)),
  if containsVar v s then
    (popVar.loop v s acc).size = s.size + acc.size - 1
  else (popVar.loop v s acc).size = s.size + acc.size
  | Stack.empty, acc =>
    by simp [AssignmentTrail.popVar.loop, containsVar,
Stack.size]
  | Stack.push (l, dl) rest, acc =>
    by
      by_cases h : l.var == v
      · simp[AssignmentTrail.popVar.loop, containsVar, h,
Stack.size, size_pushAll, Nat.add_comm]
        omega -- simplify arithmetic then do IH
      · simp only [containsVar, popVar.loop, h, Stack.size]
        have ih := loop_size v rest (Stack.push (l, dl) acc)
        simp only [Stack.size] at ih
        convert ih using 2 <;> omega
```

Using the `loop_size` lemma, we can easily extend this to the proper `popVar` function, and we add a key hypothesis that the variable we want to pop is contained within the trail; this allows us to show that `(t.popVar v).size < t.size`, with the hope of applying this to `s.trail` for solver `s`.

```
lemma popVar_size_lt_containsVar (t : AssignmentTrail) (v :
CDCL.Var)
  (hcv : containsVar v t.stack = true) : (t.popVar v).size <
t.size := by
  unfold popVar
  have h := loop_size v t.stack Stack.empty
  simp only [Stack.size, size] at h ⊢
  split_ifs at h
  · have hpos := containsVar_true_nonempty v t.stack hcv
    omega
```

The next steps for proving the overall termination invariant are to get an intermediary understanding of the `pickIncomingEdge` function, which decides which edge to continue backwards in the implicit implication graph during the 1-UIP resolution procedure. This is challenging because 1-UIP works in tandem with the decisions that BCP makes via resolution, so there will likely be nontrivial interplay between the BCP resolution and VSIDS decision procedures, and 1-UIP's `lastAssignedLit` selection and usage of the `AssignmentTrail` to work backwards in the implicit implication graph. There are a few other lemmas that we have proved regarding 1-UIP, but they are more foundational and help set-up the `popVar` and termination theorems.

3.4 Proving No-Relearning

The no-relearning theorem states that no clause can be relearned twice [2]. Our original goal was to prove this, but we have had challenges with proving BCP and 1-UIP, unfortunately leading to minimal progress on this front. That being said, based on our developments in §3.3.1 and §3.3.2, we have some forward momentum in tackling the no relearning proof. The approach taken in Fleury's work relies on the internalization of the formal CDCL calculus being used; we believe that with our intermediary invariants regarding BCP, 1-UIP, and the data structures we've chosen to use (largely sets, stacks, and arrays with verified monotonic operations), we can show via induction and similar reductions that it is impossible to generate the same learned clause twice via 1-UIP.

4 Experimental Evaluation

Owing to previously mentioned development challenges, our evaluation is limited to simple, handwritten Boolean formulas to verify functionality of our core subroutines, namely BCP, 1-UIP, and a more complex instance that involves nontrivial application of both via the solver while guaranteeing termination. A few of these simple examples are below, and the rest can be found in our examples file.[2] Below is an example of an UNSAT case:

```
-- x1 ∧ ¬x1 (via CNF)
def unsat_example : CDCL.Formula 2 :=
  { num_vars := 1, clauses := #[ {lits := #[1] }, { lits :=
#[-1] } ].toVector }
```

We also showcase a slightly more involved example related to the `AssignmentTrail`'s `popVar` subroutine, which is key in proving termination of 1-UIP:

```
-- check for popVar
def l1 : CDCL.Lit := 1
def l2 : CDCL.Lit := -2
def l3 : CDCL.Lit := 3

def trail : AssignmentTrail :={ stack := Stack.empty }
def trail1 := trail.push l1 0
def trail2 := trail1.push l2 1
def trail3 := trail2.push l3 2
```

[2] https://github.com/abhamra/verified-CDCL-datastructures/blob/main/VerifiedCdclDatastructures/Examples.lean

```
#eval trail3.toList
-- expect [(3, 2), (-2, 1), (1, 0)

def trailPostPop := trail3.popVar 2

#eval trailPostPop.toList
-- expect [(3, 2), (1, 0) bc removed l2 with var 2
```

Our original plan included benchmarks from prior SAT competitions [3] and SATLIB [11], however our progress is unfortunately behind schedule and we were unable to get key components finished, such as either a Lean-based parser from DIMACS CNF to our internal formula representation from §3.1 or a translation layer from prior work to our Lean format. See §6 for a discussion on our upcoming plans to continue our evalutaion.

5 Related Work

As alluded to in §1, there are several key verified SAT solvers that have pushed the bar forward, in a variety of frameworks and techniques. Mathias Fleury *et al.*'s verified solver in Isabelle has made great strides in translating the Weidenbach CDCL calculus, adding a host of features present in SOTA solvers like CaDiCaL [15], such as learn, restart, and forget semantics as well as the incredibly useful Two Watched Literal scheme. On the flipside, both CreuSAT and Varisat are Rust-based SAT solvers that provide proof traces; Varisat in particular has a custom proof format which uses clause hashes instead of clause ids, in addition to the ability to output and check DRAT and LRAT. It provides verified proof checkers for all three formats. CreuSAT uses Creusot [6], a deductive verifier for Rust code that can verify correctness with additional annotations, and is currently the fastest deductively verified SAT solver, owing to a combination of systems engineering and the inclusion of circular search, backtracking to asserting level, the VMTF decision heuristic, search restart, clause deletion and phase saving, which go beyond other verified solvers.

For fully verified solvers (i.e., those that are not verified via trace or through the trust-but-verify framework), the largest problem is typically the time invested into building up lemmas, theorems, and overarching strategies in order to mechanize proofs for key invariants of your solver. According to several of Fleury's papers, IsaSAT's verification and refinement to imperative programming took on the order of tens of thousands of lines of code and months to complete [2, 7]. In our Lean-based CDCL SAT solver's current iteration, we have worked on ~600 LOC of data structures, solver implementations, and proofs, over a 2 month span. With more time, we would likely be comparing against IsaSAT, CreuSAT, Varisat, and `versat`.

6 Conclusions and Future Work

In this paper, we present a partially verified CDCL SAT solver implemented in Lean, with the 1-UIP clause learning framework and VSIDS decision heuristic. Our key result is working towards proving the termination of our solver, with progress made towards the termination proofs for BCP and 1-UIP. Our next steps are to continue with our implementation and finish codifying our invariants for both of the aforementioned subroutines before moving on to a key invariant, the no-relearning theorem (Theorem 10 from [2]). After this, we aim to design helpful DIMACS CNF ingestion mechanisms, along with integration of `leansat`'s verified LRAT checker. Finally, a reasonable long-term goal is to focus on adding further support for advanced concepts such as the 2WL scheme, with the ability to learn from existing mechanized proofs like in IsaSAT, etc.

References

[1] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[2] Jasmin Christian Blanchette, Mathias Fleury, and Christoph Weidenbach. 2017. A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017. ijcai.org, 4786–4790. https://doi.org/10.24963/ijcai.2017/667

[3] Cayden Codel, Katalin Fazekas, Marijn Heule, and Ashlin Iser. SAT Competition '25. Retrieved from https://satcompetition.github.io/2025/index.html

[4] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. 2016. Efficient Certified RAT Verification. In *CADE*, 2016. Retrieved from https://api.semanticscholar.org/CorpusID:6933678

[5] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397. https://doi.org/10.1145/368273.368557

[6] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a Foundry for the Deductive Verification of Rust Programs. In *Lecture Notes in Computer Science (Lecture Notes in Computer Science)*, October 2022. Springer Verlag, Madrid, Spain. Retrieved from https://inria.hal.science/hal-03737878

[7] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. 2018. A verified SAT solver with watched literals using imperative HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, 2018. ACM, 158–171. https://doi.org/10.1145/3167080

[8] Siddhartha Gadgil. 2021. Siddhartha-Gadgil/saturn: Experiments with SAT solvers with proofs in Lean 4. Retrieved from https://github.com/siddhartha-gadgil/Saturn

[9] Aarti Gupta, Malay K. Ganai, and Chao Wang. 2006. SAT-Based Verification Methods and Applications in Hardware Verification. In *Formal Methods for Hardware Verification*, 2006. Springer Berlin Heidelberg, Berlin, Heidelberg, 108–143.

[10] Marijn J. H. Heule. 2016. The DRAT format and DRAT-trim checker. *CoRR* (2016). Retrieved from http://arxiv.org/abs/1610.06229

[11] H. Hoos and T. Stutzle. 2000. SATLIB: An Online Resource for Research on SAT. In *SAT2000*, 2000. IOS Press, 283–292.

[12] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001. 530–535. https://doi.org/10.1145/378239.379017

[13] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, 2015. Springer International Publishing, Cham, 378–388.

[14] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. 2012. versat: A Verified Modern SAT Solver. In *Verification, Model Checking, and Abstract Interpretation*, 2012. Springer Berlin Heidelberg, Berlin, Heidelberg, 363–378.

[15] Florian Pollitt, Mathias Fleury, and Armin Biere. 2023. Faster LRAT Checking Than Solving with CaDiCaL. In *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023) (Leibniz International Proceedings in Informatics (LIPIcs)*, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:12. https://doi.org/10.4230/LIPIcs.SAT.2023.21

[16] {Mukul R.} Prasad, Armin Biere, and Aarti Gupta. 2005. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer* 7, 2 (April 2005), 156–173. https://doi.org/10.1007/s10009-004-0183-4

[17] Natarajan Shankar and Marc Vaucher. 2011. The Mechanical Verification of a DPLL-Based Satisfiability Solver. *Electronic Notes in Theoretical Computer Science* 269, (2011), 3–17. https://doi.org/https://doi.org/10.1016/j.entcs.2011.03.002

[18] Sarek Høverstad Skotåm. 2022. CreuSAT, Using Rust and Creusot to create the world's fastest deductively verified SAT solver. Master's thesis. Retrieved from https://www.duo.uio.no/handle/10852/96757

[19] Armando Solar Lezama. 2008. Program Synthesis By Sketching. Doctoral dissertation. Retrieved from http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html