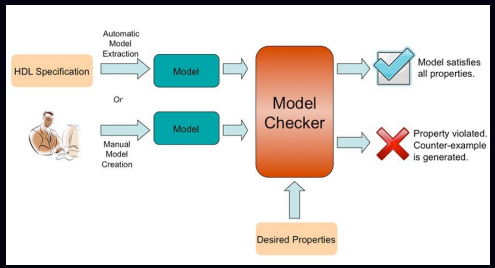
Towards a Verified CDCL SAT Solver in Lean

Arjun S. Bhamra and Cameron C. Hoechst

Verifying SAT Solvers; why care?

SAT solvers are used in a variety of mission-critical applications:

- Hardware Verification
- Program Synthesis
- Other general formal verification systems
- Much more



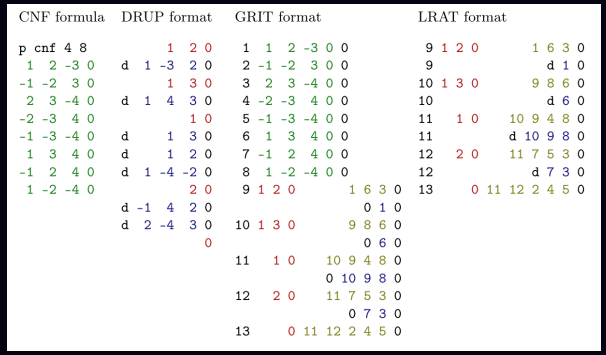
These applications care about how trustworthy SAT solvers' results are

Verification is a way to make sure we can trust solvers. It also allows us to directly implement formal specifications of algorithms in a cleaner manner, such as a CDCL calculus.

How can we verify SAT solvers?

There are two main ways to do so:

- Trust-but-verify: Design a fast SAT solver that then generates a proof trace - an output format that can be checked to ensure output correctness, especially in the UNSAT case.
  - Then, by integrating the solver with a verified trace checker (that has a smaller code surface), we can still trust that the output is correct!
  - There are a few proof trace formats, but the most common used in SAT competitions are Deletion-Resolution Asymmetric Tautologies (**DRAT**) and Linear RAT (**LRAT**).



- Full formal verification: Use an interactive theorem prover (ITP) to build up provably correct data structures and related functions that interact correctly based on invariants that are maintained via **lemmas** and **theorems**
  - If the code compiles, then it usually works
  - Can build up a formalism piece by piece
  - Some ITPs will automatically complain about termination conditions, etc., which are crucial for SAT solvers in their own right

Our Contribution: CDCL in Lean (with some proofs!)

Over the course of our project, we designed a series of:

- Data Structures
- An implementation of overall flow/structure of CDCL in Lean
  - We used the 1-UIP conflict driven clause learning scheme
  - We also leveraged the VSIDS decision heuristic for improved BCP, perf
- Proofs for
  1. The termination of BCP
  2. Getting closer to showing the termination of the recursive 1-UIP scheme
  3. Other glue methods building up to this idea

▒▒ In proving things about BCP and 1-UIP, we made some headway towards the no-relearning theorem, which states that no clause can be learned twice

▓▓ In the next few slides, we'll go over some specifics!

Data Structures

Basics

```
abbrev Var := Nat
abbrev Lit := Int
```

Clauses

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

Clause Database

```
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
```

Data Structures (cont.)

Assignment Trail (via Stack)

```
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.
```

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

Data Structures (even more)

🟦 Resolution Tree (for proofs)

```
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 SAT Solver's Structure

Solver is parameterized with the number of variables `nv` and the number of clauses in an "unknown" state `nc`

```
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 T/⊥ because of clause m.
  prop_reason   : Vector (Option Nat) nv := Vector.replicate nv none
  activity      : VsidsActivity
  -- rtree      : ResolutionTree
```

The SAT Solver's Structure: Interconnects

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.

Proofs for 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.

▓▓ Saying "trust me" doesn't work with a proof assistant

▒▒ So we have to prove termination!

The key termination condition for us is `contingent_ct`, a variable in the solver from earlier that counts the number of unknown clauses in the formula. Because of its **monotonicity**, this crucially allows us to prove termination.

In our **first** approach, we choose to prove this via `foldl` induction, because based on our Formula structure design (using `Array`s), `foldl` is bounded by the number of elements in the array.

We also introduce a new data structure, the `PropTriple`, to help with proving BCP termination:

```
1 abbrev PropTriple (nv nc : Nat) := Solver nv nc × Array (Fin nc) × Array (Fin nc)
```

Proofs for BCP (cont.)

We chose to split our implementation into three main functions:

1. `propNonUnit` propagates a variable within a particular unit clause
2. `propOne` calls `propNonUnit` on all non-unit clauses for a particular unit clause
3. `propUnits` applies `propOne` to the state of the solver for each unit clause.

```
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')
```

Proofs for BCP (even more)

```
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 establish that one call to `propNonUnit` does not increase `contingent_ct`, and then show that a call to `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
```

Interlude: Array.foldl induction

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
```

Proofs for BCP (cont. again)

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.

```
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)
```

Trying a different way for BCP

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 :=
    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
    -- return updated prop info with solver here
```

BCP Way 2 (cont.)

```
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.

We omit `propOne2` and `propOne2_decr` from the slides, but they can be seen in our report and code.

BCP Way 2: Final

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
```

Note: 2WL is hard, hence we avoid!

Recall 1-UIP

1. Start from conflict clause, set the "curr" clause to be the negation of all literals in the clause. For example, with conflict = (¬x1 ∨ x2), curr becomes (x1 ∨ ¬x2)
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 @ s.dl
7. Set clause to learnt = True
8. We are happy, one step closer to enlightenment

Proofs for 1-UIP

Below is the core of the recursive algorithm that implements 1-UIP:

```
  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 :=

    -- We do a bunch of cool stuff here, per 1-UIP

  termination_by s.trail.size
```

🟦 Our core proofs for the 1-UIP subroutine thus far revolve around proving things surrounding the `popVar` function, to show that our trail size is guaranteed to decrease to a base case.

```
-- 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 }
```

Proofs for 1-UIP (even more)

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.

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

Proofs for 1-UIP (last one)

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
```

Strategies for proving No Relearning

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 previous slides, 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.

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.

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

---

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

#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
```

Related Work

1. **IsaSAT** (by Matthias Fleury et al.) is a formally verified SAT solver with learn, restart, and forget capabilities as well as the Two Watched Literal scheme. It directly translates the Weidenbach CDCL calculus to support the aforementioned features.
    - Notably, IsaSAT's completion and verification took around 100k LOC and over two years across the various papers.
2. **CreuSAT** and **Varisat** are Rust-based SAT solvers that provide proof traces.
    - Varisat provides DRAT, LRAT, and custom formats for efficient verification, each with their own verified checker.
    - CreuSAT is verified via Creusot, a deductive verifier for Rust code that can check correctness with annotations
    - These are two of the fastest verified SAT solvers in existence, owing to the combination of SOTA features and systems programming involved

There are other examples as well, such as `versat` and SATurn, a fully verified DPLL SAT solver in Lean which partially inspired our work.

In terms of more canonical SAT solvers, standard examples include Z3 and CaDiCaL.

Conclusions and Future Work

We presented a partially verified CDCL SAT solver implemented in Lean; this is, to our knowledge, the first CDCL SAT solver implemented in Lean, and thus the first to use 1-UIP and VSIDS.

Our key results were making progress towards termination proofs of BCP and 1-UIP, and the completed design of effective data structures for use in our solver.

Lean's ecosystem is very valuable, and contribing to it while leveraging its development will allow for others to take advantage of our developments.

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.

After this, we aim to:

1. Design helpful DIMACS CNF ingestion mechanisms, along with integration of `leansat`'s verified LRAT checker.
2. 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.

Where to find our work?

Our solver, proofs, and this presentation (made via presenterm) can be found at
https://github.com/abhamra/verified-CDCL-datastructures; feedback is appreciated!

The end (questions?)