Report (Formalizing SAT Components)

Tiago Campos Ferreira

January 2025

1 Introduction

This document presents a readme and a final report for the project of the class DCC831 Theory and Practice of SMT Solving 2024.2.

2 Preliminaries

Given a set A and a symbol S over A, we define the set of formulas Formula inductively as follows:

$$
\begin{array}{lll}
\text{Formula} ::= & & \\
\quad \top & & \text{(True)} \\
\quad \bot & & \text{(False)} \\
\quad \text{Literal}(a) & \text{where } a \in A & \text{(Literal)} \\
\quad \text{And}(\varphi_1, \varphi_2) & \text{where } \varphi_1, \varphi_2 \in \text{Formula} & \text{(Conjunction)} \\
\quad \text{Or}(\varphi_1, \varphi_2) & \text{where } \varphi_1, \varphi_2 \in \text{Formula} & \text{(Disjunction)} \\
\quad \text{Neg}(\varphi) & \text{where } \varphi \in \text{Formula} & \text{(Negation)}
\end{array}
$$

Let $\mathcal{I} : A \to \{\text{True}, \text{False}\}$ be an interpretation function that assigns a truth value to each element of A. We define the satisfaction relation $\models$ for a formula F inductively as follows:

$$
\begin{array}{ll}
\mathcal{I} \models \top & \text{(True is always satisfied)} \\
\mathcal{I} \not\models \bot & \text{(False is never satisfied)} \\
\mathcal{I} \models \text{Literal}(a) & \text{if and only if } \mathcal{I}(a) = \text{True} \\
\mathcal{I} \models \text{And}(\varphi_1, \varphi_2) & \text{if and only if } \mathcal{I} \models \varphi_1 \text{ and } \mathcal{I} \models \varphi_2 \\
\mathcal{I} \models \text{Or}(\varphi_1, \varphi_2) & \text{if and only if } \mathcal{I} \models \varphi_1 \text{ or } \mathcal{I} \models \varphi_2 \\
\mathcal{I} \models \text{Neg}(\varphi) & \text{if and only if } \mathcal{I} \not\models \varphi
\end{array}
$$

We want to know if there is any interpretation I, such for a formula F, $I(F) = F$, or for all I, $I(F) = T$.

We write:

$$\mathcal{I} \models \text{Literal}(a) \qquad \text{if and only if } \mathcal{I}(a) = \text{True}$$

For any mapping M, where A is a Symbol.

```
Definition Finite A := {l : list A & Full l}.
Definition EqDec A := forall n m : A, {n = m} + {n <> m}.

Class Symbol (A : Set) := {
    finite : Finite A;
    eq_dec : EqDec A;
}.

Definition Mapping A := A → bool.
```

Notice that A has to be finite and be equality decidable. An example of such types is the sequence of numbers, $1..n$, for any natural n, the type of strings s, such the size of s has the length n, for any natural n, a sequence fixed-width w, like bits, bytes, ...

We define a model a $\models$ for a formula F:

```
Inductive EVALUATION {A} {S : Symbol A} (M : Mapping A) : Formula → bool → Prop :=
    | ETop : EVALUATION M FTop true
    | EBottom : EVALUATION M FBottom false
    | ELiteral : forall (s : A),  EVALUATION M (Literal s) (M s)
    | EAnd : forall F F' b b',
        EVALUATION M F b → EVALUATION M F' b' → EVALUATION M (And F F') (b && b')
    | EOR : forall F F' b b',
        EVALUATION M F b → EVALUATION M F' b' → EVALUATION M (Or F F') (b || b')
    | ENeg : forall F b, EVALUATION M F b → EVALUATION M (Neg F) (negb b).


Definition Satisfability {A} {S : Symbol A} F := exists (M : Mapping A), EVALUATION M F true.
Definition UNSatisfability {A} {S : Symbol A} F := forall (M : Mapping A), EVALUATION M F false.
Definition UNSatisfability2 {A} {S : Symbol A} F := ∼(Satisfability F).
```

Also, we define a evaluate procedure by:

```
Definition evaluate {A} {S : Symbol A} (f : Formula) (M : Mapping A) : bool.
induction f.
(* FTop *)
exact true.
(* FBotoom *)
exact false.
(*ELiteral *)
exact (M a).
(*EAnd*)
exact (IHf1 && IHf2).
(*EOr*)
exact (IHf1 || IHf2).
(*ENeg*)
```

```
exact (negb IHf).
Defined.

Definition partial_evaluate {A} {S : Symbol A} (f : Formula) (M : A → option bool) : Formula.
induction f.
(* FTop *)
exact FTop.
(* FBotoom *)
exact FBottom.
(*ELiteral *)
destruct (M a).
destruct b.
exact FTop.
exact FBottom.
exact (Literal a).
(*EAnd*)
exact (partial_eval_and IHf1 IHf2).
(*EOr*)
exact (partial_eval_or IHf1 IHf2).
(*ENeg*)
exact (partial_eval_neg IHf).
Defined.
```

By Theorem $fully\_eval\_is\_normal$, we know that evaluating always yields a normalized formula for a model. Also, we define a $partial\_evalution$ whenever we have a partial mapping M, such for any literal l, $M(l) = true$ or $M(l) = false$ or $M(l) = undefined$. By the theorem $fully\_eval\_is\_normal2$ we know for any partial mapping M, if $M(l)\tilde{=}undefined$, we have $partial\_evalution = evaluate$.

Definition eval_is_normal
$\{A : Set\}$
$\{S : \text{Symbol } A\}$
$(F : Formula)$
$(M : \text{Mapping } A)$
$(M_2 : A \to \text{option bool}) :=$
$\forall (M' : \text{Mapping } A)$, evaluate (partial_evaluate $F\ M_2$) $M' =$ evaluate $F\ M$.

Theorem fully_eval_is_normal
$\{A\}$
$\{S : \text{Symbol } A\}$
$(F : Formula)$
$(M : \text{Mapping } A)$:
— —
eval_is_normal $F\,M\ (\lambda x, \text{Some } (M\,x))$.

Next definitions we will use a stronger definition of normalization:

Definition is_norm $\{A\}\{S : \text{Symbol } A\}(f : \text{Formula}) : \text{bool} := f = \top \lor f = \bot$

The first part of the *sat.v* file is an introduction to propositional logic and simple lemmas.

3 Decision Procedures

3.1 Brute force

We define a decision procedure *brute_force* by:

$$\text{brute\_force}(F) = evaluate(F, M_1) \lor evaluate(F, M_n), \forall M_i$$

such: $brute\_force(F) = true \iff \exists \models_m F \land brute\_force(F) = false \iff \forall M \not\models_m F$.

By the theorem *mimimal_symbol_set*$\land$*norm_mimimal_symbol_set* we know we only need the symbols presented in the formula to pre(evaluate) a formula F. So we define *from_truth_table* a mapping between any literal in the formula F and any value it may assume.

$$\begin{aligned} \text{Theorem minimal\_symbol\_set} : \ & \forall\{A : \text{Set}\} \\ & \{S : \text{Symbol } A\} \\ & F\,(M : A \to \text{option bool}) \\ & (M' : A \to \text{option bool}), \\ & (\forall x,\, x \in F \to M\,x = M'\,x) \to \\ & \text{partial\_evaluate } F\,M = \text{partial\_evaluate } F\,M'. \end{aligned}$$

The theorem *minimal_symbol_set* states that given any two mappings, we can evaluate and get the same result of the former mapping by mapping only the symbols (or literals) inside the formula. Also, we omit here, but the evaluation

is also normal if M' is not partially defined with respect to the literals presented in the formula. However, the theorem seems trivial, this is essential for us since we abuse a lot of pre-evaluation, and it may reduce the formula literals, so in order to not care about removed literals we need to use this theorem.

By *brute_force_is_complete* and *partial_evaluate_by_table_truth_is_norm* we know *brute_force* is normal and also complete in respect to any $\models F$. Finally we have a decision procedure to verify whether a formula is satisfiable or not.

Definition partial_evaluate_by_table_truth $\{A : \text{Set}\}\{S : \text{Symbol } A\}$
$(F : \text{Formula})\,(a : \text{list bool}) : \text{Formula}.$

Theorem brute_force_is_complete $\{A : \text{Set}\}\{S : \text{Symbol } A\}\,F :$
$\forall(M : A \to \text{bool}),\ \exists(H : \mathit{listbool}),$
partial_evaluate $F\,(\lambda x,\ \text{Some}\ (M\,x)) =$ partial_evaluate_by_table_truth $F\,H.$

Theorem completeness_truth_table $(xs : \text{list bool}) :$
$xs \in (\text{get\_truth\_table}\ (\text{length}\ xs)).$

We write the complete informal theorem here to illustrate:

Proof by Contradiction. Assume, for the sake of contradiction, that the `brute_force` method is **not complete**. By this assumption, there must exist a mapping M that `brute_force` cannot define.

By the **completeness_truth_table** theorem, if such a mapping M exists, it must correspond to a truth table (i.e., a list of Boolean values B) that fully characterizes M.

However, this directly contradicts the **brute_force_is_complete** theorem, which guarantees that `brute_force` can algorithmically construct *all* mappings representable by truth tables (and thus by Boolean lists B).

Since the existence of M leads to a logical contradiction, the initial assumption is false. Therefore, `brute_force` must be complete. □

3.2 DPLL

We use the same notation from Nieuwenhuis, et al..

$$\text{UnitPropagate}: \quad M \,||\, F, C \vee l \implies M\,l \,||\, F, C \vee l \quad \text{if} \quad \begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$$

$$\text{PureLiteral}: \quad M \,||\, F \implies M\,l \,||\, F \quad \text{if}$$

$$\begin{cases} l \text{ occurs in some clause of } F \\ \neg l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M \end{cases}$$

$$\text{Decide}: \quad M \,||\, F \implies M\, l^d \,||\, F \quad \text{if}$$

$$\begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$$

$$\text{Fail}: \quad M \,||\, F, C \implies \text{FailState} \quad \text{if}$$

$$\begin{cases} M \models \neg C \\ M \text{ contains no decision literals} \end{cases}$$

$$\text{Backtrack}: \quad M\, l^d\, N \,||\, F, C \implies M\, \neg l \,||\, F, C \quad \text{if}$$

$$\begin{cases} M\, l^d\, N \models \neg C \\ N \text{ contains no decision literals} \end{cases}$$

We define *get_units_candidate* as a function that searches for candidates of unit clauses, since we know that there could be more than one unit clause. *get_units_candidate* returns the pivot literal inside the unit clause. There is a few theorems about the correctitude of *get_units_candidate* and the unit rule as well.

Given a mapping any M and any interpretation $\models$, we state:

Theorem delete_unit
$(S : \text{Symbol } A)$
$(M : A \to \text{option bool})$
$(c : \text{list CLiteral})$
$(\text{unit} : \text{CLiteral})$
$(H_0 : \text{In unit } c)$
$(H_2 : \text{filter } (\lambda x : \text{CLiteral}, \text{is\_none } (M(\text{get\_symbol\_cliteral } x))))\, c = \{\text{unit}\})$
$(H_1 : \forall x : \text{CLiteral}, \text{In } x(\text{unit} :: c) \to x \neq \text{unit} \to \text{partial\_evaluate } (\text{CLiteral\_to\_Formula } x)\, M = \text{FBottom})$
$---$
partial_evaluate $(\text{Clause\_to\_Formula } c)\, M = \text{FTop}$

Theorem sat_unit_is_always_true $\{A : \text{Set}\}\{S : \text{Symbol } A\}\, (\text{cnf} : \text{CNF})\, M$
$(\text{sat} : \text{partial\_evaluate } (\text{CNF\_to\_Formula cnf})\, M = \text{FTop})\, \text{unit} :$
$\text{unit} \in \text{cnf}$
unit is an Unit clause
$---$
partial_evaluate $(\text{Clause\_to\_Formula unit})\, M = \text{FTop}.$

Theorem unit_property1 $\{A\}\{S : \text{Symbol } A\}\,(c : \text{Clause})\,(M : A \to \text{option bool})\,x$:
get_units_candidate $c\,M = \text{Some } x \to M\,(\text{get\_symbol\_cliteral } x) = \text{None}$.

Theorem unit_property2 $\{A\}\{S : \text{Symbol } A\}\,(c : \text{Clause})\,(M : A \to \text{option bool})\,\text{unit}\,x$:
get_units_candidate $c\,M = \text{Some unit} \to$
$x \in c \to x \neq \text{unit} \to M\,(\text{get\_symbol\_cliteral } x) = \text{Some } (\text{inverse}'\,x\,\text{false})$.

Lemma unit_correctness $(A : \text{Set})$
$(S : \text{Symbol } A)$
$(M : A \to \text{option bool})$
$(c : \text{list CLiteral})$
$(\text{unit} : \text{CLiteral})$
$(H0 : \text{In unit } c)$
$(H2 : \text{filter } (\lambda x : \text{CLiteral, is\_none } (M\,(\text{get\_symbol\_cliteral } x)))\,c = \{\text{unit}]\}$
$(H1 : \forall x : \text{CLiteral},$
$x \in \{unit\} \cup c \to$
$x \neq \text{unit} \to \text{partial\_evaluate } (\text{CLiteral\_to\_Formula } x)\,M = \text{FBottom})$:

---

partial_evaluate $(\text{Clause\_to\_Formula } c)\,M = \text{FTop}$.

Finally using the theorem above:

Theorem 1 (correctude_get_units_candidate)**.** *For any type A, symbol interpretation S : `Symbol` A, clause c, partial assignment $M : A \to$ `option bool`, and literal u:*

- ***Given:***
  1. $u \in c$ *(`In` predicate)*
  2. `get_units_candidate` c $M =$ `Some` u
- ***Then:*** *iff u has polarity 0:*

  `partial_evaluate` $\big($`Clause_to_Formula` $c\big)$ $\big(u = true \cup M\big)$ $=$`FTop`

  otherwise,

  `partial_evaluate` $\big($`Clause_to_Formula` $c\big)$ $\big(u = false \cup M\big)$ $=$`FTop`

3.2.1 Watched literals*

Watched literal property come almost free with the lemmas above:

Proof of Triviality via Given Theorems

Proof. **Case 1: Both Watched Literals Assigned Non-Satisfying Values**

- By `unit_property2`, all non-watched literals $x \neq w_1, w_2$ satisfy:

$$M(\text{get\_symbol}(x)) = \texttt{Some}(\text{inverse}'\, x\, \texttt{false})$$

- Thus, all literals except w_1, w_2 evaluate to `FBottom` under M
- Apply `unit_correctness`:
  - Filter unassigned literals: $\texttt{filter}\,(\lambda x.\, \texttt{is\_none}\, M(x))\, c = [w_1, w_2]$
  - By `unit_property1`, $M(w_1) = M(w_2) = \texttt{None}$
  - Contradicts clause being non-conflict; hence c must be conflict

Case 2: One Watched Literal Unassigned, Other Assigned Non-Satisfying

- Let w_1 be unassigned ($M(\text{get\_symbol}(w_1)) = \texttt{None}$), w_2 assigned false
- By `get_units_candidate`, identify w_1 as unit candidate
- Apply `correctude_get_units_candidate`:
  - Assign w_1 based on polarity: $M' = M \cup \{w_1 \mapsto \texttt{true/false}\}$
  - Theorem guarantees $\texttt{partial\_evaluate}\, c\, M' = \texttt{FTop}$
- Thus, w_1 must be propagated as unit

□

3.3 Pure Literal

Theorem 2 (pure_literal_theorem)**.** *For any type A, symbol interpretation S : `Symbol` A, partial assignment $M : A \to$ `option bool`, clause c, atom a, and purity proof H : `is_pure_literal` c a:*
iff a has polarity 0

$$\texttt{partial\_evaluate}\ \big(\texttt{Clause\_to\_Formula}\ c\big)\ \big(a = True \cup M\big)\ = \texttt{FTop}$$

otherwise,

$$\texttt{partial\_evaluate}\ \big(\texttt{Clause\_to\_Formula}\ c\big)\ \big(a = False \cup M\big)\ = \texttt{FTop}$$

4 Extracting the components

Once the key aspects are formalized we can mechanize in some programming language by extracting in Coq (we could also write the entirely DPLL function, however, due to the termination checker this would depend on non-trivial well-founded recursion termination proof).

We extract the components from Coq using the plugin extraction and we decide to target Ocaml:

```
let rec dpll s cnf formula ctx =
  (* Check memoization table first *)
  let memo_key = create_memo_key formula ctx in
  match Hashtbl.find_opt memo_table memo_key with
  | Some result → result
  | None →
    (* Apply unit propagation and pure literal elimination *)
    let ctx = apply_until_stop step_unit s cnf ctx in
    let ctx = apply_until_stop step_pure_literal s (List.flatten cnf) ctx in
    let formula = partial_evaluate s formula (coq_ContextMaps ctx) in

    let result = match check_sat formula with
      | Sat →
          (true, ctx)
      | Unsat →
          (false, ctx)
      | Incomplete →
          (* Choose next variable using heuristic *)
          let next_var = choose_next_variable s ctx cnf in
          (* Try true first, then false *)
          let (res, r) = dpll s cnf formula (MNat.add next_var true ctx) in
          let (res', r') = dpll s cnf formula (MNat.add next_var false ctx) in
          if res then (res, r) else (res', r')
    in
    (* Store result in memoization table *)
    Hashtbl.add memo_table memo_key result;
    result
```

A small and trivial optimization was implemented for memorization.

5 Benchmarks

Benchmarks are provided in `sat_bench.txt`. *Minisat* breaks us in every instance. While the solver demonstrates capability in handling non-trivial problem instances within practical time constraints, its performance remains substantially inferior to state-of-the-art SAT solvers. Two primary bottlenecks contribute to this efficiency gap:

- **Search Strategy Limitation:** The implementation exclusively employs DPLL.

- **Code Extraction Overhead:** The OCaml code generated through Coq's extraction mechanism retains unoptimized proof obligation:
  - Suboptimal memory management patterns (aka using list or maps instead of vectors)
  - We rely on AVL tree implementation
  - Redundant computational steps from constructive proofs

6 References

Nieuwenhuis, R., Oliveras, A., & Tinelli, C. (2006). Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM, 53(6), 937–977.