Formal Software Verification Project Report
SAT Solver

Nathan Schmidt

Aarhus University, Aarhus, Denmark
Student number: 202400801

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

1 Introduction

The *Boolean Satisfiability Problem (SAT)* consists in determining whether, given a formula of propositional logic, there exists a *valuation* that satisfies it, i.e., a mapping from the unknowns of the formula to boolean values such that the formula holds. Great numbers of problems in computer science can easily be reduced to SAT, and even though SAT is known to be **NP**-complete by the Cook-Levin theorem of 1971 [1] and its many proofs since then, there exist many efficient *SAT-Solvers* for certain classes of formulas.

In this project, we implement a small formally verified SAT-Solver in the Coq proof assistant for formulas containing conjunctions, disjunctions, implications, and negations. To that end, we start by formalizing the syntax of such formulas. Then, we introduce their semantic interpretation given a specified valuation. Following the implementation of a syntactic optimizer that we show to simplify a formula to its minimal form, we conclude this work by implementing the actual solver based on a brute-force search algorithm and prove it to be both correct and complete.

2 Implementation and Encountered Problems

In this section, we present our implementation choices and compare them to discarded alternatives to justify them. We also describe what difficulties we encountered during development and how we addressed them.

2.1 Syntax

First, we formalize the abstract syntax of boolean formulas containing conjunctions, disjunctions, implications, and negations.

Abstract Syntax. We state an inductive definition of the type `form` with a set of constructors describing how to build up a formula:

$$p, q ::= x \mid \texttt{true} \mid \texttt{false} \mid p \wedge q \mid p \vee q \mid p \to q \mid \neg p \tag{1}$$

For identifiers, we introduce the type `id` with a single constructor `Id` that wraps a string. We also define an equality function comparing two identifiers and returning `true` if and only if their contained strings are equal. To ease proof development in the rest of the project, we prove some basic lemmas and theorems about the function by case distinction, namely *reflexivity*, *equivalence with propositional equality*, and *decidability* of equality of identifiers. For the decidability theorem, one needs to ensure its proof is concluded by `Defined.` instead of `Qed.` as its proof object is needed in the computation of later used functions and thus has to be saved.

Concrete Syntax. To make reading and writing formulas easier, we use Coq's `Notation`s system, as well as a `Coercion` from an identifier as a formula. The main difficulty lies in assigning precedence levels to the individual constructors that pay respect to the commonly assumed binding of operators. Given the *binds stronger* relation ($>$), that is:

$$\neg > \wedge > \vee > \to \tag{2}$$

For example, $x \vee \neg y \wedge z \to \texttt{false}$ is interpreted as $(x \vee ((\neg y) \wedge z) \to \texttt{false}$.

2.2 Semantics

Now that we can write a formula in Coq, we want to define its semantics, i.e., if we interpret it as `true` or `false`.

Valuations. To do so, we need to know which boolean values to replace all identifiers occurring in a formula with. Therefore, we define the type `valuation`. A *valuation* is a function that, being passed an identifier, returns `true` or `false`, or, in other words, a valuation is a total map from identifiers to booleans. Our implementation is analogous to the total map defined by Pierce et al. in [3].

Again, to simplify writing and reading valuations, we introduce a `Notation x !-> b ;; v` to override a valuation `v` with a new value `b` for `x`. At first, we used just a single semicolon, but this caused conflicts with list notations. Additionally, we define an empty valuation to map all identifiers to `false`.

We also specify some lemmas of [3] for later reasonings. Their proofs rely on the functional extensionality axiom, stating that two functions are equal if and only if their applications to all their possible arguments are equal. It is known to be compatible with Coq's logical core.

Interpreter. The `valuation` type allows us to define a recursive interpretation function `interp`. Applied to a valuation and a formula, it returns `true` if and only if the formula holds by traversing a formula bottom-up and pattern matching on it. All the necessary functions, such as `andb`, `orb`, and `negb` are already implemented in Coq. They even suffice to compute the result of an implication since $p \to q$ is known to be equivalent to $\neg p \vee q$.

2.3 Optimizer

Sometimes, a formula's interpretation can be derived or, at least, simplified on a purely syntactical level, leaving it in a form that is easier to read and reason upon and marginally reducing the computation effort needed by the interpretation function while preserving the semantics. In this part of the project, we therefore introduce an optimization function `optim`.

Minimality. A key challenge was formally defining what a simplification is and what property the result of `optim` should have.

A *simplification* reduces the applications of the binary operators $\wedge$, $\vee$ and $\to$ to one of their two arguments or an *atom*, i.e., the boolean values `true` and `false`. Additionally, when applied to an atom, the unary operation $\neg$ can be simplified to an atom.

The aim is to leave a formula p in a form that cannot be simplified further on a syntactical level. We assume this to be the case in exactly two mutually exclusive situations:

- p is an atom, or
- p does not contain any atoms.

When one of the situations applies, we say p is in *minimal form*. Formalizing this definition in Coq, we require either a fixpoint returning `true` if and only if a formula does not contain atoms or an inductive proposition. While we initially opted for a fixpoint, proving the correctness of our optimizer later turned out easier with an inductive proposition. A formula does not contain an atom if it is an identifier or if all its subformulas do not contain atoms.

We settle on a set of laws that our optimizer uses and allow it to fulfill the above-mentioned correctness and minimality requirements:

$$\texttt{true} \wedge p \equiv p \qquad p \wedge \texttt{true} \equiv p \qquad \texttt{false} \wedge p \equiv \texttt{false} \qquad p \wedge \texttt{false} \equiv \texttt{false} \tag{3}$$

$$\texttt{true} \vee p \equiv \texttt{true} \qquad p \vee \texttt{true} \equiv \texttt{true} \qquad \texttt{false} \vee p \equiv p \qquad p \vee \texttt{false} \equiv p \tag{4}$$

$$\texttt{true} \to p \equiv p \qquad p \to \texttt{true} \equiv \texttt{true} \qquad \texttt{false} \to p \equiv \texttt{true} \qquad p \to \texttt{false} \equiv \neg p \tag{5}$$

$$\neg\texttt{true} \equiv \texttt{false} \qquad \neg\texttt{false} \equiv \texttt{true} \tag{6}$$

Note that we leave out some further simplification potential. For improved provability, we only concentrate on laws involving at least one atom. One could also go beyond and, e.g., add laws such as $p \wedge \neg p \equiv \texttt{false}$ for an arbitrary formula p to this set.

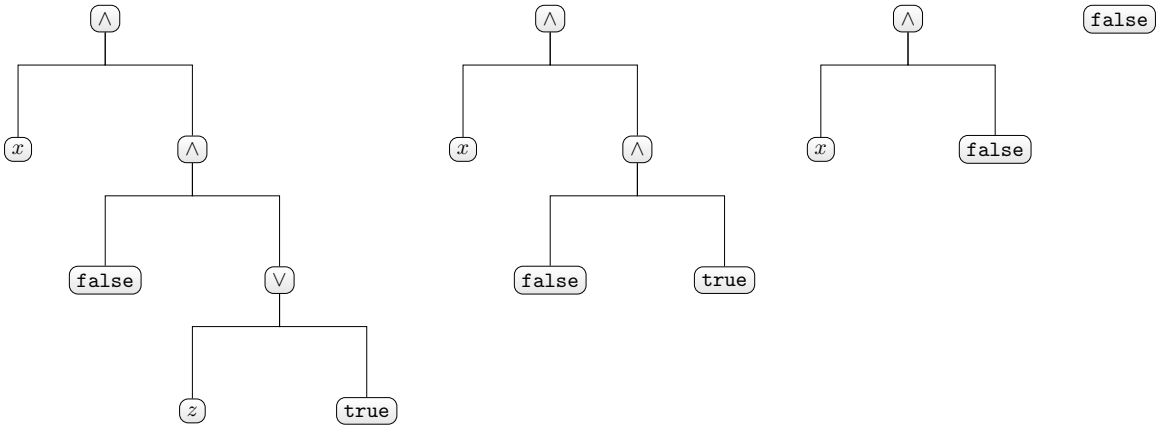

Fig. 1. Successive syntactical optimization steps on $x \wedge (\texttt{false} \wedge (z \vee \texttt{true}))$.

Implementation. The actual implementation of the optimizer in Coq requires a bit of thought. Indeed, our first implementation attempt was a top-down traversal of the abstract syntax tree of the given formula and successive application of the listed simplifications. However, this misses some optimization potential as it may only become available after optimizing subformulas. To try and remedy this, we wrote a recursive function repeatedly applying `optim` on a formula until reaching a fixpoint. While a fixpoint is eventually reached in practice, Coq cannot know and rejects the function as new arguments to `optim` are not obviously smaller.

Instead, our revised approach only traverses the passed formula once and hence only requires linear runtime. We achieve this by performing a post-order depth-first-search (DFS) on the abstract syntax tree of the passed formula. This way, optimizations of subformulas are directly taken into account. Even though sometimes it is sufficient to simplify only one or even no subformula, we always traverse the whole tree for easier proving of the required properties. Figure 1 illustrates such a case, where for the formula $x \wedge (\texttt{false} \wedge (z \vee \texttt{true}))$, the subformula $z \vee \texttt{true}$ is simplified to `true` first, even though $\texttt{false} \wedge (z \vee \texttt{true})$ could directly be simplified to `false` from the get-go. Note, however, that the formula is translated to `false` without even running the interpreter just by its syntax.

Properties. To close off this section, we must formally verify that our optimizer truly meets its requirements. First and foremost, the following theorem holds:

Theorem 1. *For all valuations v and formulas p, $\texttt{interp}\ v\ p = \texttt{interp}\ v\ (\texttt{optim}\ p)$, i.e., the optimizer is correct since it preserves the semantics of formulas.*

Proof. We proceed by induction on the structure of p. The non-trivial cases $p = q_1 \wedge q_2$, $p = q_1 \vee q_2$ and $p = q_1 \to q_2$ are all shown by case distinction on $\texttt{optim}\ q_1$ and $\texttt{optim}\ q_2$ and application of the induction hypotheses claiming the optimizer is correct for q_1 and q_2. $p = \neg q$ follows the same pattern for a single subformula q. □

The main challenges for formally proving this theorem in Coq do not lie in the reasoning but rather in removing redundancies. Consequently, as for many other proofs of the project, we decided to first write a mostly manually written version before looking out for automation potential. In a final step, we filtered out common patterns through custom tactics through the `Ltac` language.

We also prove this second theorem to show our optimizer is actually exhaustive:

Theorem 2. *For all formulas p, the result of the optimizer $\texttt{optim}\ p$ is in minimal form.*

Proof. We proceed by induction on the structure of p. For the cases $p = q_1 \wedge q_2$, $p = q_2 \vee q_2$, $p = q_1 \to q_2$ and $p = \neg q$, we perform case distinctions on the induction hypotheses (minimality of the subformulas), as well as the results of the optimizer applied to them. If necessary, we invert the induction hypotheses involving the deconstructed optimizer results. Then, it follows directly from the hypotheses set that p either does not contain atoms or, on the contrary, is an atom. □

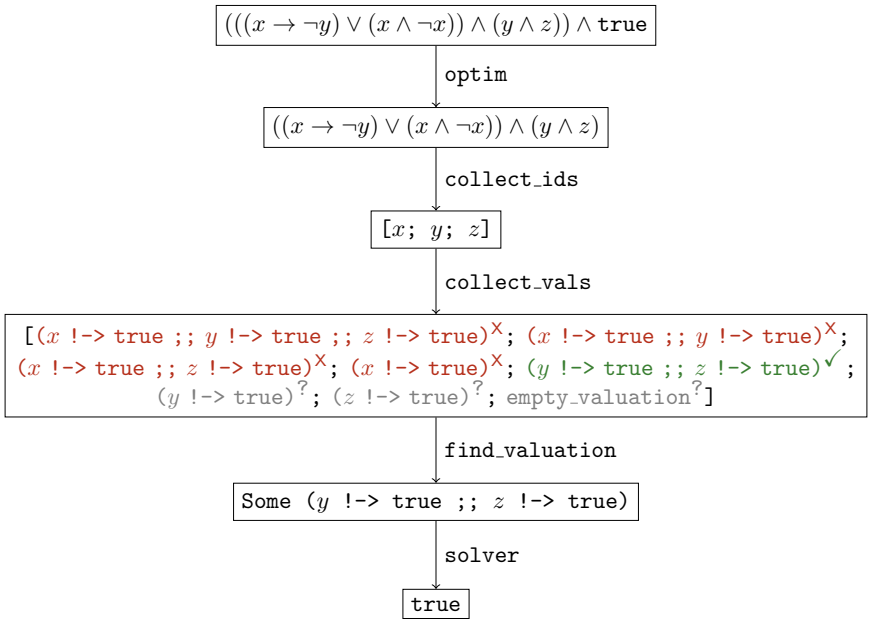

Fig. 2. Exemplary solving process for $(((x \to \neg y) \vee (x \wedge \neg x)) \wedge (y \wedge z)) \wedge$ `true`.

For this Coq proof, many cases are very similar, but, e.g., differ in the actual atom p is equal to, reducing the potential for automation, as one must carefully introduce a fitting witness for the left branch of minimality, i.e., `exists b, p = form_bool b`. Nevertheless, we factor out the recurrent patterns of deconstructing and rewriting with the induction hypotheses and destructing `optim` q for a subformula q and then inverting one of the induction hypotheses. In some rare cases, some forms are shelved in the process, requiring the use of `Unshelve`, which we were unaware of beforehand.

2.4 Solver

In this section, we can finally deal with our SAT solver. Our idea is to collect representatives for all classes of valuations that are susceptible of satisfying a given formula and test them on the formula until one positive interpretation is found or no valuation matches. Indeed, only identifiers that are in a formula influence its interpretation, and we can safely map all identifiers not contained in a formula to `false` in our collected valuations:

Lemma 1. *For any formula, two otherwise identical valuations differing in the value of an identifier that is not contained in the formula lead to an identical interpretation.*

Initially, we wanted to write a function that directly collects all relevant valuations for a given formula. While this is possible, it would have inevitably collected duplicate valuations as comparing valuations is not computable, as valuations are, in turn, functions themselves. Duplicate valuations can, however, lead to a significantly increased runtime of our solver in some cases. For instance, consider the unsatisfiable formula $x \wedge \neg x$. The initial function would collect `[`x `!-> true ;` x `!-> true ; empty_valuation]`. We would then check the valuation x `!-> true` twice, even though it does not satisfy the formula. Generalized to examples with many duplicate identifiers in respective subformulas, this does not scale.

Instead, we decide to implement our solver with three individual functions. We name them explicitly rather than defining them as anonymous fixpoints and can hence show some detached properties in Coq. Throughout this section, we will use the formula $(((x \to \neg y) \vee (x \wedge \neg x)) \wedge (y \wedge z)) \wedge$ `true` as running example to illustrate the solving steps as depicted in figure 2.

Extracting Identifiers. To tackle the above-mentioned problem, we provide a dedicated recursive function `collect_ids` that collects all identifiers of a formula without duplicates (compare example in figure 2).

We wanted the result of the function to be a finite set but did not find a standard library implementation that fully corresponded to our expectations. Thus, we first decided to return a list and take

```
Fixpoint ids_union (l1 l2 : list id) : list id :=
  match l1 with
  | [] ⇒ l2
  | x:: xs ⇒ if existsb (eqb_id x) l2 then ids_union xs l2 else ids_union xs (x :: l2)
  end.
```

List. 1. Custom duplicate free list merging function.

```
Fixpoint collect_vals (l : list id) (acc : list valuation) : list valuation :=
  match l with
  | [] ⇒ acc
  | x:: xs ⇒ collect_vals xs ((map (fun v ⇒ x !→ true ;; v) acc) ++ (map (fun v ⇒ x !→ false ;; v) acc))
  end.
```

List. 2. Original tail-recursive valuation collection function.

care ourselves to not include any duplicates. The abandoned implementation of a function merging the lists of collect identifiers of two subformulas without duplicates can be found in listing 1. We formally proved that elements included in either `l1` or `l2` are all kept in the result, that any element of the result is originally contained in `l1` or `l2`, and that if `NoDup l1` and `NoDup l2`, then `NoDup (ids_union l1 l2)` holds.

After some deeper search through the standard library, we, in the end, opted for the finite set implementation provided in `Coq.Lists.ListSet`[1]. Even though it does not hide many implementation details, it comes with some predefined set manipulation operations and some lemmas about its properties, which would not have justified a similar custom re-implementation. To confirm it matches our expectations, we show that `set_add` behaves as the identity function if we try adding an identifier x to a set that already contains it and that it behaves as an append of the one-element list `[`x`]` to the end of the set, which in fact is just a list, else.

The most relevant property of `collect_ids` is however the following:

Lemma 2. *An identifier is contained in a formula p if and only if it is contained in* `collect_ids` *p.*

Proof. We consider the forward ($\Longrightarrow$) and backward ($\Longleftarrow$) directions separately. For both directions, the proof of the statement by induction on p is fairly straightforward.

- In the forward direction, the conjunction, disjunction, and implication cases require a deconstruction of the hypothesis stating that some arbitrary but fixed identifier is contained in at least one subformula. We can then show that identifiers of subformulas are preserved using our induction hypotheses.
- The backward direction is analogous, using the fact that an identifier contained in the union of two sets is contained in at least one of the two sets. □

While the proof simply requires a case distinction on our hypothesis and applying the relevant induction hypothesis for the inductive cases, the main difficulty was finding and plugging in relevant standard library lemmas when needed, avoiding proving already known facts. Other than that, the formal proof contained a lot of repetition, and we were able to significantly shorten it using `Ltac` for a common pattern of deconstruction followed by application and a succession of `try` tactics.

Collecting Valuations. As we now have the ability to collect the identifiers of formulas, we need to transform identifier sets into a list of all relevant valuations that could satisfy this formula. We hence introduce the `collect_vals` function. Its implementation maps each identifier of the passed set to `true` on top of the already collected valuations, which are already kept. One does not need to explicitly map identifiers to `false` since the function returns the empty valuation for the empty set, which maps all identifiers to `false`, and is hence always included in the resulting list (which we formally prove in Coq).

Initially, we implemented a tail-recursive version of the function based on an accumulator element as depicted in listing 2. When called with `[empty_valuation]` as starting accumulator, the function yields the desired result. However, while this allows code optimizations as all calls to the function are last, it complicates proving properties as some assumptions on the accumulator are required. This illustrates

[1] `https://coq.inria.fr/doc/v8.9/stdlib/Coq.Lists.ListSet.html`

```
Definition check_vals (p : form) (l : list valuation) : option valuation :=
  let l' := map (fun v ⇒ if interp v p then Some v else None) l in
  match find (fun o ⇒ match o with Some _ ⇒ true | None ⇒ false end) l' with
  | Some o ⇒ o
  | None ⇒ None
  end.

Definition check_vals' (p : form) (l : list valuation) : bool :=
  existsb (fun b ⇒ b) (map (fun v ⇒ interp v p) l).
```

List. 3. Alternative valuation search functions.

the often encountered trade-off between efficiency considerations of smart implementations and proof complexity. Additionally, we suspected at first that explicitly mapping newly added identifiers to `false` would make some proofs easier, but the opposite turned out to be the case.

To verify the `collect_vals` correctly includes valuations for both truth values of every identifier contained in the passed finite set, we prove we formally prove the following lemma in Coq:

Lemma 3. *For all identifier sets S, any $x \in S$ if and only if there exist valuations in `collect_vals` S where x is respectively mapped to `true` and `false`.*

A significant part of the backward direction of this lemma was proven using a custom `Ltac` tactic. Still, as `Ltac` is only quickly tackled in the course, we lacked the knowledge to automatically derive witnesses from the list of assumptions.

Searching Through Valuations. The final solving step performed by `check_vals` is to search through a list of valuations l and return `Some` v if v, contained in l, satisfies the formula p. It returns `None` if no valuation matches. One considered option was to use the library function `map` to interpret all collected valuations on a formula, and then either `find` a matching valuation or, if the actually found valuation does not matter, return the first found `true` in the altered list. Two possible implementations are shown in listing 3. Even though we avoid explicitly writing pattern matching and recursion, these options are both less efficient – `interp` is applied to all valuations, not just the first satisfying the formula – and complicate proofs, especially improving reliance on standard library lemmas. Consequently, in our final version of the function, we just recurse over the list of valuations until the first match, which is returned. Later valuations are not considered, as illustrated by our running example in figure 2.

Putting it all together, the function `find_valuation` accepts a formula p and returns the result of the successive applications of `optim`, `collect_ids`, `collect_vals`, and `check_vals`. Note that in the worst case, `find_valuation` searches through all relevant valuations if no valuation satisfies the formula, leading to a worst-case complexity of $\mathcal{O}(n^2)$ with n being the number of identifiers contained in the formula. Indeed, this makes sense as **NP** $\subseteq$ **EXPTIME**.

The function `solver` then connects the result of `find_valuation` to booleans, i.e., returns `true` for `Some` v and `false` for `None`. In other words, `solver` is a *decision procedure* [2] for the boolean satisfiability problem. This means `solver` is an algorithm that, given a concrete formula p, can always answer the problem by yes (`true`) if p is satisfiable, and by no (`false`) if it is not satisfiable. We will deal with the formal proof of this claim in the remainder of this subsection.

Soundness. First, we need to show that `solver` has no false positives, i.e., does not falsely claim a formula to be satisfiable that is not. This gives rise to the *soundness* lemma:

Lemma 4. *For all formulas p, if `solver` p returns `true`, then p is satisfiable.*

Proof. Assume `solver` p to return `true`. This is only the case when `find_valuation` p returns `Some` v. We show by induction on the length of the list l = `collect_vals (collect_ids (optim` p`))` that `interp` v `(optim` p`)` is true, from which our claim directly follows as `optim` is semantics-preserving and v is our witness for the satisfiability of p.

- Base case: if l = `[]`, `find_valuation` p must be `None`, a contradiction with our hypothesis.
- Inductive step: let $l = v'$`::`vs and our induction hypothesis be `check_vals (optim` p`) vs = Some` $v \implies$ `interp` v p `= true`. If `interp` v' p `= true`, then $v' = v$, and thus `find_valuation` p returns `Some` v. Otherwise, we can apply our induction hypothesis and our goal immediately follows from our hypothesis. □

This can be concisely formalized in Coq. One needs to perform some careful unfoldings while not revealing the inner parts of `solver`: it is irrelevant which valuation is returned, just that some valuation is returned. Only then should the induction be performed.

Correctness.

Lemma 5. *For all formulas p, if p is satisfiable, then* `solver` *p returns* `true`*.*

Theorem 3. *For all formulas p,* `solver` *p returns* `true` *if and only if p is satisfiable.*

2.5 Miscellaneous

Difficulty with Import/Export Proved quite a lot of lemmas, often needed to keep track to not prove again (e.g., collect vals not empty can simply be proven by empty valuation in collect vals, no need for induction again as originally performed)

3 Conclusion

References

[1] S. A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047.

[2] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Berlin Heidelberg, 2016. ISBN: 9783662504970. DOI: 10.1007/978-3-662-50497-0.

[3] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Logical Foundations*. Ed. by B. C. Pierce. Vol. 1. Software Foundations. Version 6.7, http://softwarefoundations.cis.upenn.edu. Electronic textbook, 2024.