Formal Software Verification Project Report
SAT Solver

Nathan Schmidt

Aarhus University, Aarhus, Denmark
Student number: 202400801

Abstract. SAT solvers have proven themselves useful for a vast amount of problems in computer science. This report describes the implementation and proof-scripting challenges for developing and formally verifying a small brute-force SAT solver in Coq. We tackle how to state the syntax and semantics of boolean formulas and present a way of optimizing formulas based on their syntax, addressing constraints arising from using Coq. Moreover, we formalize SAT and build and verify our decision procedure step-by-step, describing provability considerations, most prominently concerning functions. We also briefly reflect on our experience working with Coq.

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 be reduced to SAT, and even though it 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 for formulas containing conjunctions, disjunctions, implications, and negations with the help of the Coq proof assistant [4]. 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 present our actual solver based on a brute-force search algorithm and prove it to be both sound and complete, i.e., a valid decision procedure.

2 Implementation and Encountered Problems

This section presents our implementation choices and compares 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 syntax of the types of boolean formulas we consider.

Abstract Syntax. We inductively define the type `form` with the help of which we can build up formulas:

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

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 wrapped strings are equal. To ease proof development in the rest of the project, we prove some basic lemmas and theorems about the equality function by case distinction, namely *reflexivity*, *equivalence with propositional equality*, and *decidability* of identifier-equality. 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 defined functions and thus has to be saved.

Concrete Syntax. To make reading and writing formulas easier, we use `Notations` and a `Coercion` from an identifier as a formula. The main difficulty lies in assigning precedence levels to the individual constructors of `form` that pay respect to the commonly assumed binding of operators. Precedences are assigned according to a *binds stronger* relation (>):

$$\neg > \wedge > \vee > \rightarrow$$

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

2.2 Semantics

Now that we can write formulas in Coq, we want to define their semantics, i.e., determine when to interpret them as `true` or `false`.

Valuations. To do so, we need to know which boolean values to replace the 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, it is a total map from identifiers to booleans. Our implementation is analogous Pierce et al.'s 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`. `x !-> b` overrides the empty valuation, defined to map all identifiers to `false`. At first, we used a single semicolon instead of two, but this caused conflicts with list notations. We also specify some lemmas based on [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. It traverses the provided formula bottom-up and applies pattern matching to 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 of formulas. 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 properties the result of `optim` should have. A *simplification* reduces selected 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 the opposite atom.

The aim is to leave a formula p in one of two mutually exclusive shapes:

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

When one of the situations applies, we say p is in *minimal form*. As we will later see, this does not necessarily mean that a formula cannot be simplified further on a syntactic level, even though these phenomena sometimes coincide. 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. Either way, both options should reflect that a formula does not contain an atom if it is an identifier or all its subformulas do not contain atoms. While we initially opted for a fixpoint, proving the correctness of our optimizer later turned out easier with an inductive proposition.

We settle on a set of laws to fulfill the above-mentioned correctness and minimality requirements:

$$
\begin{array}{cccc}
\texttt{true} \wedge p \equiv p & p \wedge \texttt{true} \equiv p & \texttt{false} \wedge p \equiv \texttt{false} & p \wedge \texttt{false} \equiv \texttt{false} \\
\texttt{true} \vee p \equiv \texttt{true} & p \vee \texttt{true} \equiv \texttt{true} & \texttt{false} \vee p \equiv p & p \vee \texttt{false} \equiv p \\
\texttt{true} \to p \equiv p & p \to \texttt{true} \equiv \texttt{true} & \texttt{false} \to p \equiv \texttt{true} & p \to \texttt{false} \equiv \neg p \\
 & \neg\texttt{true} \equiv \texttt{false} & \neg\texttt{false} \equiv \texttt{true} &
\end{array}
$$

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

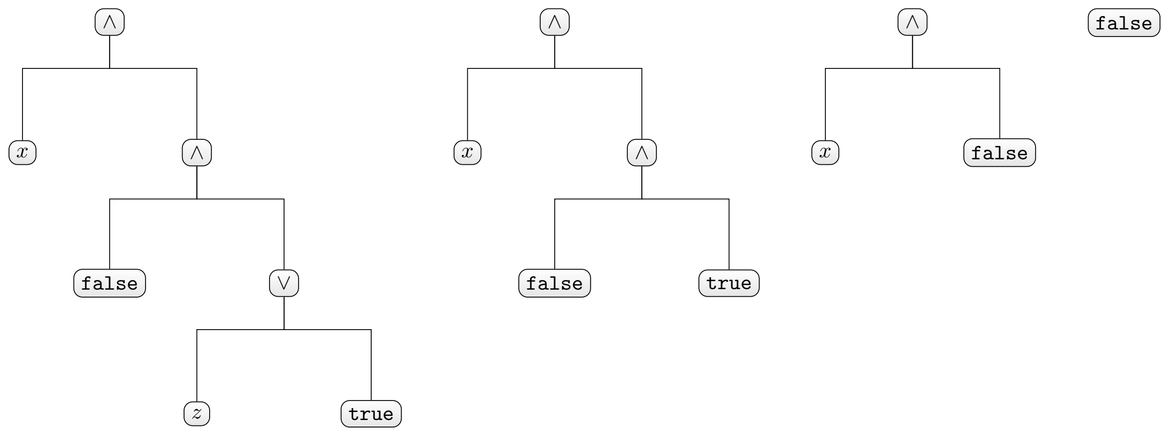

Fig. 1. Successive syntactical optimization steps on $x \wedge (\texttt{false} \wedge (z \vee \texttt{true}))$.

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 a given formula and successive application of the listed simplifications. However, this is not minimizing as some simplifications 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 rejects the function as new arguments 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, simplified subformulas are directly taken into account. Even though it is occasionally 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 observing its syntax.

Properties. To close this section on `optim`, 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 in removing redundancies. Consequently, as for many other proofs, we wrote a mostly manual version before looking for automation potential. In a final step, we filtered out common patterns through custom `Ltac` tactics.

We also prove this second theorem to show our optimizer is 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 their optimization results. 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. □

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

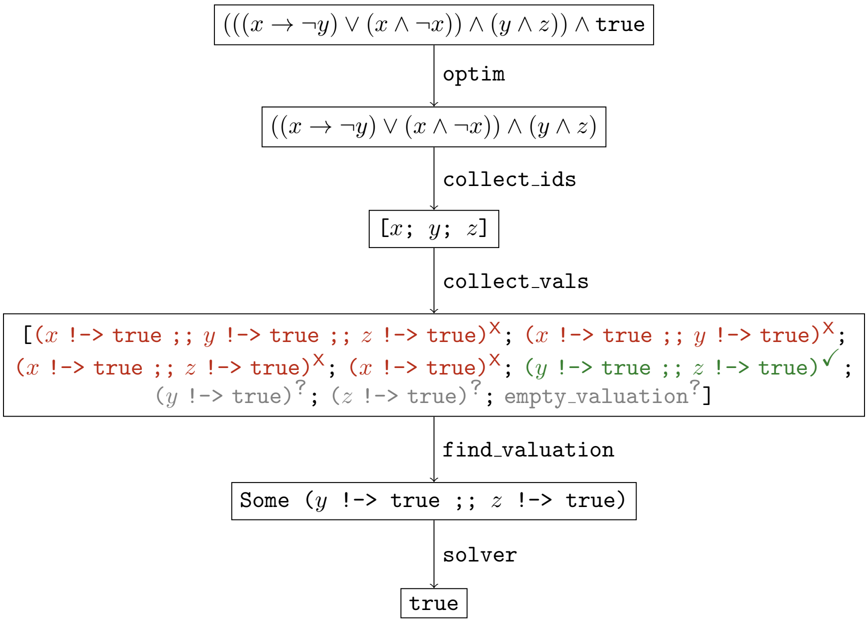

Fig. 2. Exemplary solving process for $(((x \to \neg y) \vee (x \wedge \neg x)) \wedge (y \wedge z)) \wedge$ true.

```
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.

2.4 Solver

In this section, we can finally directly deal with our SAT solver. Our idea is to collect representatives for all classes of valuations susceptible of satisfying a given formula and test them on it until one positive interpretation is found or no valuation matches. Indeed, only identifiers contained in a formula influence its interpretation, and we can safely map all other identifiers to `false` in all 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 the same interpretation.*

Initially, we wanted to directly collect all relevant valuations for a given formula. While this is possible, it would have inevitably collected duplicate valuations as comparing valuations is not computable since they are functions. Duplicate valuations can, however, lead to a significantly increased runtime of our solver in several cases. For instance, consider the unsatisfiable formula $x \wedge \neg x$. Our initial function would collect `[(`x` !-> true); (`x` !-> tru)e; 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 implement our solver with three individual functions. We name them explicitly rather than defining them as anonymous fixpoints and can hence show 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 problem mentioned above, we provide a dedicated recursive function `collect_ids` that gathers all identifiers of a formula without duplicates (compare 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 decided at first to return a list and take care of not including duplicates ourselves. The abandoned implementation of a function merging the lists of collect identifiers of two subformulas duplicate-free can be found in listing 1. We formally proved that

```
Fixpoint collect_vals (ids : set id) (acc : list valuation) : list valuation :=
  match ids 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.

elements included in either `l1` or `l2` are all kept in the result, that any element of the result is initially 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 its implementation details, it comes with predefined set manipulation operations and useful lemmas about its properties, which would not have justified a similar custom re-implementation. To confirm it matches our expectations, we show in Coq that `set_add` behaves as the identity function if we add an identifier x to a set that already contains it and that it behaves as a right-hand-side append of the one-element list `[`x`]` 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 only 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 is repetitive, and we thus 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. Next, we need to transform identifier sets into lists of relevant valuations that could satisfy a formula. To this end, we introduce the `collect_vals` function. It maps each identifier of the provided set to `true` on top of already collected valuations, which are kept. One does not need to explicitly map identifiers to `false` since the function returns the empty valuation for the empty set and it is hence always included in the resulting list (which we formally prove in Coq).

Initially, we implemented a tail-recursive version based on an accumulator as depicted in listing 2. The function yields the desired result when called with `[empty_valuation]` as the starting accumulator. While all function calls being last enables code optimizations, it complicates proving properties as some assumptions on the accumulator are required. This illustrates the often-encountered trade-off between efficiency considerations and proof complexity. On another note, we suspected at first that explicitly mapping newly added identifiers to `false` would make some proofs easier, but the opposite is the case.

To verify `collect_vals` behaves as expected, we formally prove the following lemma in Coq:

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

A significant portion of the backward direction of this lemma was proven using a custom `Ltac` tactic. Still, 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 the valuation 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 call `interp`

[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.

on all collected valuations and p, and then either `find` a matching valuation or, if it 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 on l, these options are both less efficient: `interp` is applied to all valuations, not just the first satisfying the formula. It also complicates proofs, especially increasing reliance on standard library lemmas. Consequently, in our final version of the function, we 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 everything together, the function `find_valuation` takes a formula p as parameter and returns the result of the successive applications of `optim`, `collect_ids`, `collect_vals`, and `check_vals`. Note that in the worst case, it searches through all relevant valuations if none satisfies the formula, leading to a worst-case complexity of $\mathcal{O}(n^2)$, n being the number of identifiers contained in the formula.

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

Soundness. First, we need to show that `solver` does not falsely claim formulas to be satisfiable that are 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 returns `true`. This is only the case when `find_valuation` p returns `Some` v. We will show by induction on the structure 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.

- Base case: if $l =$ `[]`, `find_valuation` p must be `None`, a contradiction with our hypothesis.
- Inductive step: let $l = v' :: vs$. Our induction hypothesis claims that `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, our claim follows from our induction hypothesis and our hypothesis. □

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

Completeness. Showing the completeness of `solver` was the most challenging part of the project, which we unfortunately did not fully manage to wrap up. Indeed, one cannot blindly use induction to prove it as the satisfiability hypothesis is not directly usable. It took us a lot of time to figure out a coherent reasoning, writing a lot of later unused and removed helper lemmas.

Assuming a formula p to be satisfiable means a valuation v exists such that `interp` v p is `true`. However, it is in no way guaranteed that this is the valuation that will be found by `find_valuation`. The problem has its roots in valuations being total maps. For instance, let v satisfy the formula $x \vee y$. It is perfectly possible for v to map an identifier z not contained in the formula to `true` as this does not influence the formula's interpretation. In fact, our solver will not consider this exact v, as all uncontained identifiers are mapped to `false`. Consequently, we first have to deduce our solver will consider some *equivalent* v'. This requires a formalization of the notion of equivalence first:

Lemma 5. *For all valuations v, v', and formulas p, if v and v' map all identifiers contained in p to the same value, then the interpretation of p under the context of these two valuations is identical.*

The main difficulty for this lemma does not lie in its proof, which can be performed by induction on p, but in the careful translation of this statement in Coq. One needs to notice that

```
forall (v v' : valuation) (p : form) (x : id),
  (id_in_form x p = true → v x = v' x) → interp v p = interp v' p
```

only means v and v' correspond in a single identifier, which naturally does not imply their interpretations to be identical. We rather need to nest the universal quantifier for x:

```
forall (v v' : valuation) (p : form),
  (forall (x : id), id_in_form x p = true → v x = v' x) → interp v p = interp v' p
```

However, this nesting will later cause our reasoning not to be formalizable in Coq.

Now, we want to prove that `find_valuation` will truly consider a valuation v' equivalent to the known satisfying valuation v:

Lemma 6. *For all valuations v and formulas p, there exists a valuation v' such that v and v' map all identifiers contained in p to the same value and v' is in* `collect_vals (collect_ids p))`.

To formally show this in Coq, we first have to prove that `collect_vals` preserves valuations when performing the union of identifier sets, i.e., that if a valuation is contained in `collect_vals` S, then it is also contained in `collect_vals (id_set_union` S S'`)` and in `collect_vals (id_set_union` S' S`)` for some second set S'. Additionally, we also show that arbitrary combinations of valuations $v \in S$ and $v' \in S'$ are contained in `collect_vals (id_set_union` S S'`)`. These lemmas are shown by induction on the second set applied to `id_set_union` as this is the argument it pattern matches on. We also have to be careful and keep our induction hypotheses as general as possible and not introduce valuations too early. Therefore, we use `generalize dependent` before using the `induction` tactic.

We then tried proving lemma 6 by induction on p. The problem we face for the cases $p = q_1 \wedge q_2$, $p = q_1 \vee q_2$, and $p = q_1 \rightarrow q_2$ is that to know which of the two induction hypotheses to apply, we would have to introduce an arbitrary but fixed identifier x and distinguish if it is contained in q_1 or q_2. This cannot be done in Coq as the universal quantifier for x is nested. Consequently, x cannot be introduced without explicitly providing a witness for v' before, but the proper choice, in turn, depends on the case distinction. We tried different options to try and circumvent the problem, such as making explicit assertions. Using the `eexists` tactic also did not have the wished effect since it uses the first found witness in all subsequent cases, where it does not work anymore. As there is no such concept as dependent witnesses, our final attempt was to shift the case distinction inside the valuation by providing

```
fun x ⇒ if id_in_form x q1 then v1 x else if id_in_form x q2 then v2 x else empty_valuation x
```

as a witness and making use of one of our helper lemmas. Yet, Coq cannot unify our goal with the lemma, probably because the valuation function's parameter is not part of the context. Replacing the arbitrary booleans in the lemma with the specific condition makes the lemma unprovable. Therefore, we have seen ourselves forced to `admit` the concerned parts of the formal proof.

Admitting lemma 6 to hold, all prerequisites are united to prove the completeness of `solver` and subsequently that `solver` is a decision procedure for SAT:

Lemma 7. *For all formulas p, if p is satisfiable, then* `solver` *p returns* `true`*.*

Proof. Assume p to be satisfiable, i.e., that there exists a valuation v such that `interp` v p is `true`. By lemmas 5 and 6, we know that there exists a valuation v' in `collect_vals (collect_ids` p`)` with the same interpretation as v. Consequently, `interp` v' p is true, and, as we know our optimizer to be correct, `interp` v' `(optim` p`)` is `true` as well. By induction on `collect_vals (collect_ids (optim` p`))`, it is easy to show `solver` p returns `true`. □

Theorem 3. *For all formulas p,* `solver` *p returns* `true` *if and only if p is satisfiable, i.e.,* `solver` *is a decision procedure for the boolean satisfiability problem.*

Proof. The theorem is a direct consequence of the soundness and completeness of `solver`. □

2.5 Miscellaneous

To close off the discussion of our project, we provide an alternative implementation strategy for our valuation collection function and describe further challenges.

```
Fixpoint collect_vals (ids : set id) : list (list (id * bool)) :=
  match ids with
  | []  => [[]]
  | x:: xs => let vs := collect_vals xs in map (cons (x, true)) vs ++ map (cons (x, false)) vs
  end.
```

```
Fixpoint idbools_to_val (l : list (id * bool)) : valuation :=
  match l with
  | []  => empty_valuation
  | v:: vs => override (idbools_to_val vs) (fst v) (snd v)
  end.
```

List. 4. Alternative valuation collection function and a valuation converter.

Instead of explicitly collecting a list of all possible valuations, one could compute a list of lists of pairs of identifiers and truth values where pairs represent mappings from identifiers to their values. Then in `check_vals`, one could not directly plug in valuations but would need to convert lists of pairs to valuations. Listing 4 presents possible implementations of the described changes. One hope is that these changes would counteract the problems encountered when proving the completeness of `solver`: one would still have difficulties with dependent witnesses but could maybe find a witness taking into account the structure of `collect_vals` and working in all cases, as one would not have to show function equality. A supposed accurate witness is

```
l1 ++ filter (fun v2 => negb (existsb (fun v1 => eqb_id (fst v1) (fst v2)) l1)) l2
```

for lists `l1`, `l2`, even though at first sight, the inclusion of `filter` seems problematic since the standard library does not contain appropriate lemmas. Unfortunately, we did not have the time to investigate this alternative implementation strategy further.

Another observation we made is that because of the great amount of lemmas we have proven, it is easy to lose track, especially for later parts of the project where we added and then discarded many propositions. As a result, some proof scripts are more lengthy and complex than necessary. For example, we first proved the lemma stating that the result of `collect_vals` is never the empty list using induction but later realized that the lemma is a corollary from an already proven lemma, namely that the empty valuation is contained in every resulting list.

The final challenge we want to elaborate on is dealing with Coq's module management system, in particular the `Require`, `Import`, and `Export` commands. In our project's draft, we used the provided library file[2]. Even though, in the end, we merged the whole project into a single file since we only used a fraction of the provided definitions of the library file, during the project's development, we encountered a multitude of unexpected behaviors, e.g., the `app` function spuriously rejecting its provided arguments. Hence, we tried broadening our knowledge on the matter by reading excerpts of Coq's platform-docs[3].

3 Conclusion and Reflection

This section concludes the report and offers a reflection of our personal experience working with the interactive theorem prover Coq.

3.1 Conclusion

In this project, we implemented and formalized a small brute-force SAT solver and encountered some difficulties dealing with the intricacies of Coq. Our concrete syntax using `Notation`s caused conflicts with lists, as well as initially having a separate library file. At first, Coq's requirement for strictly decreasing arguments for recursive functions also made the implementation of our optimizer harder but later forced us to implement it in a bottom-up manner, which proved the better approach anyway. On multiple occasions, we had to weigh the advantages and downsides of boolean functions compared to inductive propositions for certain properties. When implementing our solver, we were faced with difficulties proving propositions about functions using accumulators. We also found it challenging to find fitting data structures in the

[2] `https://www.cs.au.dk/~spitters/project_lib.v`
[3] `https://coq.inria.fr/platform-docs/RequireImportTutorial.html`

standard library, and to automate large parts of our proofs, even though they had a quite similar structure, especially as existential quantification requires the explicit introduction of witnesses. Finally, we found out that nested quantifications make proofs significantly harder, as do functions in propositions since to prove their equality, one usually needs the functional extensionality axiom.

3.2 Reflection

We personally found it quite intuitive to write programs and proofs in Coq. The syntactic style closely resembles OCaml's, which we already had some multi-year experience with. Tactic scripts are a sensible simulation of informal reasoning, and bullets structure proofs well. In general, writing programs and proving their properties feel closely intertwined, which leads to a pleasant workflow. Additionally, features such as a compiler are useful tools. Coq mostly returns comprehensible error messages. We also appreciate the ease of documentation offered by `coqdoc`, which one can quickly pick up.

In contrast, we also experienced the limitations of Coq's constructive logic, making some propositions harder or even impossible to prove. This has to be carefully considered when developing programs, which limits the range of implementation choices. The Isabelle/HOL proof assistant, which we have previously been shortly introduced to, felt less intuitive and harder to internalize, but supports different proof styles more natively and follows mathematical reasoning more closely, thus ultimately seeming to be more reasonable to learn in the long run. Moreover, the level of detail required by Coq can become frustrating, e.g., having to use tactics like `symmetry` or `reflexivity` frequently. Automation potential seems limited compared to Isabelle/HOL, at the cost of one having to deal with nitty-gritty details.

Narrowed down, the choice of proof assistant to us is related to its intended use. Focusing on Coq and Isabelle/HOL, we observe a trade-off between better readability and comprehensibility of proofs offered by the former and more powerful and time-saving automation, sometimes at the cost of understanding what is actually happening in the background, by the latter. We mostly see Coq as a great choice to formally check proofs we mostly thought through already, whereas Isabelle is more about finding intermediate goals that it can automatically reach, especially considering the presence of power tools such as `sledgehammer` or `nitpick`.

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.

[4] *The Coq Reference Manual. Release 8.18.0.* Institut National de Recherche en Informatique et en Automatique. 2023. URL: `https://coq.inria.fr/doc/v8.18/refman/`.