Functional Programming Project

Mikkel Milo, 201505317 (au543733)
Aarhus University

December 2018

1 Introduction

This report documents the development of a formalization of a SAT-Solver in Coq. I chose the project among a list of suggested projects on `http://www.cs.au.dk/~spitters/fun18projects.pdf`. The Coq development includes

- a formalization of boolean formulae
- an interpreter of boolean formulae (given a valuation)
- a simple SAT-solver
- soundness proof of SAT-solver
- a negational-normal-form (nnf) converter for boolean formulae
- soundness and completeness of said converter
- soundness of solver using nnf-converter
- examples and sanity checks of most of these

Furthermore, I have extended the base project with necessary QuickChick generators and shrinkers to conveniently test (a weaker notion of) completeness of the SAT-solver. In particular, I have solved all mandatory parts, including one of the extensional exercises (nnf converter). Some of the less interesting mandatory exercises (such as providing X example formulae) only exist in the attached coq development file, and are not described in this report.

I will describe all these developments – some in more detail than others – with focus on the latter parts, and discuss some of the problems I have encountered, and possible alternative solutions.

2 Boolean Formulae

In this section I will briefly describe how I have defined boolean formulae in Coq and their corresponding interpreting function. A boolean formula consists either of the `true` or `false` values, or an identifier/variable, or a composition of other boolean formulae with composing functions: $\wedge, \vee, \Rightarrow$ and $\neg$. It is thus simple to define an inductive datatype `form` that represents boolean formulae:

```
Inductive form :=
  | var : id -> form
  | ftrue : form
  | ffalse : form
  | and : form -> form -> form
  | or : form -> form -> form
  | imp : form -> form -> form
  | neg : form -> form.
```

Of course this is simply a datatype definition, so it contains no semantics (ie. when a formula is true or false). We define the semantics using the usual truth table semantics by defining the following interpreting function (although in principle we have defined it using an operational semantics that just so happens to be equivalent to the truth table semantics – or at least so we hope!), where a `valuation` is a function from identifiers to booleans.

```
Fixpoint interp (V : valuation) (p : form) : bool :=
  match p with
  | ffalse => false
  | ftrue => true
  | var id => V id
  | and f1 f2 => (interp V f1) && (interp V f2)
  | or f1 f2 => (interp V f1) || (interp V f2)
  | imp f1 f2 => if (interp V f1) then (interp V f2) else true
  | neg f => negb (interp V f)
  end.
```

The above function is just a straight-forward implementation of the semantics of the usual boolean operators e.g. a boolean formula `X` $\wedge$ `Y` is true for a valuation `V` if and only if both `X` and `Y` are. From this we can define what it means for `form` to be *satisfiable*.

```
Definition satisfiable (p : form) : Prop :=
  exists V : valuation , interp V p = true.
```

In the following sections we use the following notations for `and`, `or`, `imp`, and `neg`, respectively: $\wedge, \vee$, `-->`, and `!`. The latter two were chosen such that they do not conflict with existing Coq syntax.

3 SAT-solver

In this section I describe the implementation of the SAT-solver and the corresponding soundness proof. I discuss possible alternative solutions and their consequences.

A SAT-solver decides if a boolean formula is satisfiable or not. This is a well-known NP-Complete problem, so assuming $\mathbf{P} \neq \mathbf{NP}$ we should not expect a worst-case running time better than 2^n, where n is the number of variables in the formula. My goal was to implement a SAT-solver that is relatively easy to verify soundness on, rather than an efficient SAT-solver. The most simple solver one can think of is one that simply tries all 2^n valuations until it finds one where `interp` evaluates to true, or until it has exhausted the search space. This is the kind of solver I have chosen. To implement this, three issues must first be solved: finding the set of variables in a given formula, a suitable representation of valuations such that the "next" valuation to try is defined unambiguously, and finally some way of guaranteeing (read: convincing Coq) that the function terminates. The last one should ideally be implied by the second one but, as we shall see, Coq is not always (understandably) so smart.

The first issue is rather straightforward. It is simply a function that recurses on the structure of the given formula, and adds all identifiers not seen before to a result list. See `form_ids`, `merge_id_lists`, and `contains_id` in the Coq development file if you are interested in the specifics. There is probably a shorter and more concise implementation for this problem, but here it doesn't matter because the following proofs will not rely on this implementation at all.

For the second issue we can simply interpret an assignment of variables to boolean values as a binary number, and then define a function that performs a "binary increment" on the assignment. We represent an assignment with the type `list (id * bool)`. This way there is also an immediate conversion from assignments to valuations. See `next_assignment` in the Coq code for the implementation of this "binary increment" function.

With these first two issues in order, we arrive at a seemingly obvious auxilliary function for `find_valuation`:

```
Fixpoint find_valuation_aux
  (p : form) (a : assignment) : option valuation :=
  let V' := assignment_to_valuation a in
  if is_all_true_assignment a
    then if interp V' p
        then Some V'
        else None
  else if interp V' p
    then Some V'
    else find_valuation_aux p (next_assignment a).
```

Here, `find_valuation_aux` would initially be called with the all-false assignment (representing 000...), and it would terminate either when it finds a satisfiable valuation, or when `a` is the all-true assignment (representing 111...). However, Coq will not accept this fixpoint because it cannot guess the decreasing argument. Indeed, there is no structurally decreasing argument in the above fixpoint, however we know that, if `next_assignment` does indeed behave as described, then there is a always finite number of steps until the `is_all_true_assignment` predicate is true, and so the function must always terminate. Of course we can't expect Coq to derive this fact by itself. One possible way of handling this is to instead explicitly enumerate all 2^n assignments in a list and remove an assignment once we have tried it. I dislike this idea because it uses too much space. Instead I have opted for the Coq-idiomatic solution using a "fuel" parameter, that bounds the maximum recursion depth. The final implementation is shown below.

```
Fixpoint find_valuation_aux
  (p : form) (a : assignment) (fuel : nat) : option valuation :=
  let V' := assignment_to_valuation a in
  match fuel with
  | 0 => if interp V' p
      then Some V'
      else None
  | S n => if interp V' p
    then Some V'
    else find_valuation_aux p (next_assignment a) n
  end.
```

Coq happily accepts this. As a side note, the downside is that it becomes harder to prove completeness. Luckily, in this particular case, we know exactly how to choose `fuel` because the recusion depth is bounded by exactly 2^n. This is not significant for our purposes though, since I will only be proving soundness.

I now give a high level proof of soundness for the solver. The `solver` definition is simply a wrapper on `find_valuation` that returns `true` when `find_valuation` returns `Some V` and otherwise returns `false`. Thus the main theorem we wish to prove is `forall p V, solver p = true -> satisfiable p`, however this is a direct consequence of proving

```
forall p V,
    find_valuation_aux p (empty_assignment p) (2^(length (form_ids p)))
    = Some V -> satisfiable p
```

This was the proposition I initially tried to prove, however I ran into problems because of the `fuel` argument. I realized it was better to generalize the theorem, and instead prove the above proposition for all assignments and all values of `fuel`. This turned out to be quite easy to prove.

```
Lemma find_valuation_aux_sound : forall p A V n,
  find_valuation_aux p A n = Some V -> satisfiable p.
```

The proof uses induction on n. In the base case our assumption simplifies to

```
(if interp (assignment_to_valuation A) p
     then Some (assignment_to_valuation A)
     else None) = Some V
```

where we simply case on the guard. In the first case the result follows directly, and in the second case we conclude a contradition. In the inductive case our assumption simplifies to

```
(if interp (assignment_to_valuation A) p
     then Some (assignment_to_valuation A)
     else find_valuation_aux p (next_assignment A) n
```

Again we case on the guard. The first case follows immediately, and the second case follows by our induction hypothesis. This concludes the proof. Note that the above proof does not use structural induction on `form`. This may seem odd, but looking at the implementation of the solver it becomes clear: the solver is indifferent to the structure of the given formula. It simply tries all possible valuations.

4 Negational-Normal-Form

In this section I describe my nnf converter, and proof outlines of the associated proofs.

A boolean formula is said to be in negational-normal-form if negations only occur in front of variables. I have implemented the converter as a structurally recursive function on the given formula. It makes convenient use of De Morgan's Laws to convert `!(a ∧ b)` into `!a ∨ !b`, `!(a ∨ b)` into `!a ∧ !b`, and `!(a --> b)`, which is equivalent to `!(!a ∨ b)` by the semantics of $\Rightarrow$, into `a ∨ !b`. The remaining cases are rather straightfoward. Below is the implementation. Note that the fuel pattern is also necessary here since the three cases mentioned above perform recursive calls on non-strictly decreasing arguments. Also note that although there is only used one pattern match on `p`, interally Coq will expand this to two nested pattern matches. This manifests in the Coq proof.

```
Fixpoint nnf_aux (p : form) (fuel : nat) : form :=
  match fuel with
  | 0 => p
  | S n => match p with
    | !ffalse => ftrue
    | !ftrue => ffalse
    | !var id => p
    | !(f1 /\ f2) => (nnf_aux (neg f1) n) \/ (nnf_aux (neg f2) n)
    | !(f1 \/ f2) => (nnf_aux (neg f1) n) /\ (nnf_aux (neg f2) n)
    | !(f1 --> f2) => (nnf_aux f1 n) /\ (nnf_aux (neg f2) n)
    | !!f => nnf_aux f n
    | f1 /\ f2 => (nnf_aux f1 n) /\ (nnf_aux f2 n)
    | f1 \/ f2 => (nnf_aux f1 n) \/ (nnf_aux f2 n)
    | f1 --> f2 => (nnf_aux f1 n) --> (nnf_aux f2 n)
    | _ => p
    end
  end.
```

The main theorem we must then prove is

```
Theorem nnf_solver_sound : forall p,
  nnf_solver p = true -> satisfiable p.
```

where `nnf_solver` is the same as `solver`, except it starts by converting the formula to an nnf formula. I first tried to show this by showing `satisfiable (nnf p) -> satisfiable p`, however I found that sometimes my hypothesis in an induction step sometimes seemed too weak (or at least too clumsy) to prove what was necessary. I instead proved both soundness and completeness of `nnf_aux` simultaneously, which gives a stronger induction hypothesis, as we shall see. The lemma is shown below.

```
Lemma nnf_aux_sound_and_complete : forall p n V,
  interp V (nnf_aux p n) = true <-> interp V p = true.
```

This intuitively says that we may use `p` and `(nnf p)` interchangably. The proof is roughly 90 lines of Coq code, so I won't go into too much detail. Luckily, large parts of it are simply context manipulations to reach a contradiction, which are easily explained intuitively.

The proof proceeds by induction on n. In the base case both directions follow immediately after inserting $n = 0$ and simplifying the terms. In the inductive case we consider all possible constructions on `p`. The `!ffalse`, `!ftrue`, and `!var id` cases are all either trivial or follow directly from the hypotheses. The `f1 ∧ f2`, `f1 ∨ f2`, and `f1 --> f2` cases follow from the distributivity of `interp` on those operators, e.g. `interp (f1 ∧ f2)` $\iff$ `(interp f1) ∧ (interp f2)`.

For the case of `p = !f` I will only present the $\Rightarrow$ direction, since the other direction is fairly uninteresting because it is simply a sequence of the usual rules on boolean operators paired with applications of the induction hypothesis.

Given `interp V (nnf_aux (! p) (S n)) = true` and the induction hypothesis `forall p : form, interp V (nnf_aux p n) = true <-> interp V p = true` we must show `(interp V !p) = true`. In the Coq proof this requires quite a bit of structural manipulation, since we must now also case on the construction of `f`. For our purposes, though, it is sufficient to look at our first assumption and see that, intuitively, if `interp` evaluated `(nnf_aux !p)` to false in the $n+1$th step, then it must also have evaluated it to false in the nth step. Applying the induction hypothesis, we reach our goal. This completes the proof outline.

5 QuickChick on `forms` and (partial) Completeness Testing of `solver`

In this section I describe the necessary developments for convenient automatic generation of `form` terms for use in automatic testing of properties on `forms` using QuickChick. I was inspired to do this development by one of the other optional exercises on proving completeness of the solver. Instead I wanted to try to QuickChick this property[1]. Ideally, the property we want to check is `satisfiable p ==> solver p`, however recall that `satisfiable p` was defined as `exists V, interp V p = true`. QuickCheck requires that all predicates it uses to generate data implement the `Decidable` typeclass, but since `satisfiable` quantifies over the set of valuations, which is infinite, it will not accept this property. However, since we know there is only a finite number of unique valuations for a given formula (albeit exponential in the size of the formula), I believe it should technically be possible to manually implement the `Deciable` typeclass by proving completeness of interp (I have only proved soundness). This could be a very demanding task, so I have instead opted for a weaker notion of completeness:

```
Definition solver_pseudo_complete (p : form) (V : valuation) :=
  (interp V p) ==> (solver p).
```

which QuickCheck accepts (specifically, `(interp V p)` is decidable because it is a function, and Coq requires all functions to be terminating). In order to test this property we must first construct generators of valid `forms` and `valuations`. There is no way to construct "invalid" instances of `forms`, so we could just have the QuickChick library derive a generator automatically using `Derive Arbitrary term`. A sample formulae generated by this generator is shown below

```
[
  (ffalse /\ ffalse /\ ffalse) /\
    ((ftrue /\ ffalse) /\ ffalse \/ ftrue -->
      vId 5 \/ vId 9 \/ ftrue);
  ftrue --> vId 4;
  vId 3;
  !(! (ftrue /\ ffalse \/ ffalse));
  !(ftrue /\ (ftrue \/ ftrue) /\ ffalse --> ftrue);
  vId 2;
  !(! (ftrue --> ffalse)) -->
    ((vId 1 \/ vId 2 --> ftrue) \/ ffalse);
  ffalse --> ftrue \/ ffalse;
]
```

We see that the derived generator generates a lot of uninteresting formulae with many ffalse and ftrue constructors, which can always be removed by rewriting

[1]there is a also hint of irony in *testing* completeness that I find particularly amusing.

the formula. Instead, we want identifiers to occurs more often, since these are what dictates the "complexity" of the formula. I therefore made a better generator, shown below. The generator is very similar to the ones we've seen in the QuickChick chapter of Software Foundations. Note that `ffalse` and `ftrue` are not among the possible choices of constructors in the successor case. I use the `liftM` and `liftM2` monad combinators to easily apply the constructors inside the resulting generators.

```
Fixpoint genFormSizedBetter (n : nat) : G form :=
  match n with
  | 0 => liftM var (arbitrarySized n)
  | S n' => freq [
    (1, liftM var (arbitrarySized n));
    (4, liftM2 and (genFormSizedBetter n') (genFormSizedBetter n'));
    (4, liftM2 or (genFormSizedBetter n') (genFormSizedBetter n'));
    (4, liftM2 imp (genFormSizedBetter n') (genFormSizedBetter n'));
    (4, liftM neg (genFormSizedBetter n'))
    ]
  end.
```

For convenience I have also implemented a shrinker for `forms`, but since it is not strictly necessary for our purposes of testing (partial) completeness I will not describe it here. See the Coq development file if you are interested in the implementation. As a side note, the shrinker also partly solves the second optional exercise which asks for an optimizer that eliminates `ftrue` and `ffalse` from `forms`.

Next I describe how to generate `valuations`. These seem quite odd to generate, since they are just functions from identifiers to booleans. It is more apparent how to generate a data structure such as a list. Indeed, a simpler choice is to generate `assignment` datatypes (which are just of type `list (id * bool)`), and then convert these to `valuation` instances using the existing converter from the development of the solver. A generator for assignments is simple: given a list of identifiers simply assign a uniformly randomly chosen boolean value to each. The function below implements this behavior.

```
Fixpoint genAssignment (ids : list id) : G assignment :=
  let genIdBoolPair :=
    (fun (i : id) => oneOf [
      ret (i, false) ;
      ret (i, true)
      ])
  in
  match ids with
  | [] => ret []
  | i :: ids' => liftM2 cons (genIdBoolPair i) (genAssignment ids')
  end.
```

With these generators we can test the desired property. QuickChick gives us the following result:

```
QuickChecking
  (forAllShrink (genFormSizedBetter 4) shrink
    (fun p => forAll
      (genAssignment (form_ids p))
      (fun A => solver_pseudo_complete p (assignment_to_valuation A))
    )
  )
+++ Passed 10000 tests (7198 discards)
```

All tests passed, so it would be reasonable to think `solver` is (partially) complete. We see that QuickChick discards many testcases because the formula is not satisfiable. We could try to construct a generator only for satisfiable formulae, however this problem is NP-hard (because then we would be able to solve the satisfiability problem efficiently as well), so I would expect QuickChicking using such a generator to be too slow for any non-trivially-sized formulae. With my current generator i already experienced memory issues when the size argument for `genFormSizedBetter` was set to 10. I leave these investigations as future work to keep the project within a reasonable scope.

6 Conclusion

In this project I developed a formalization of boolean formulae in Coq, and proved soundness of a SAT-solver for these formulae. As suggested by the given exercises, I implemented the solver in a very naïve fashion where it just exhaustively tries all possible valuations for the given formula. As a consequence of this simplicity, the corresponding soundness proof was also very short and simple.

Next I implemented function for converting formulae into their negational-normal-form. I proved soundness and completeness of this converter, which turned out to be a rather large proof, although my experience is that most of the proof is simply low-level structural manipulations and equivalences Coq terms containing boolean operators. I imagine the proof could be simplified significantly by exploiting reflection between many common propositions using boolean operators.

Finally, I extended the project with QuickChicking capabilities for `form`s and tested (a slightly weaker notion of) completeness, with success. Constructing generators and working with QuickChick was a rather straightforward and enjoyable experience, given the experience I built from doing exercises during the course. The `form` generator produced many formulae which were discarded in testing completeness because most generated formulae were not satisfiable. Implementing a generator of only satisfiable formulae could be very tricky, and in particular very inefficient, because this problem is NP-Complete. Investigating this further could be part of a future work.

Overall I found that generalizing propositions as much as possible helped simplify the corresponding proofs, which is also what we have been taught from Software Foundations.

Most of the design choices occurred during development of the solver. Here I needed to find a suitable datastructure for representing `valuations`, which were given as the type `id -> bool`. I chose to represent them as `assignments` with type `list (id * bool)`. I found this representation easy to work with, although I suspect it could complicate a completeness proof of the solver. Here I suspect a more direct, functional approach might solve this problem, but to keep the scope at a reasonable level I chose not to explore these questions.