Lab 3: A sheep SAT solver

15-414: Automated program verification

Lab goals

The goal of this homework is to specify and implement a simple SAT solver that takes a formula in conjunctive normal form as an input and decides whether or not it is satisfiable by enumerating every possible valuation of its variables. In the next lab, we will make it more efficient by implementing and proving the unit propagation rule.

For this lab and the next one, we allow pair programming. If you choose to work with a classmate, be sure to submit only one solution with your two names on it.

Lab instructions

We show Figure 2 an extract of the template for this lab where we define a type for formulae in conjunctive normal form (CNF). An example of how a CNF is represented using this type is provided Figure 1. Besides, a valuation (also called *interpretation* in some lecture notes) is represented as an array of booleans whose i^{th} component is the value of x_i.

```
{ nvars = 4;
  clauses = [
    [ {var=3; value=false} ];
    [ {var=0; value=true}; {var=2; value=false}; {var=3; value=true} ];
    [ {var=1; value=false}; {var=2; value=true} ] ] }
```

Figure 1: Representation of the formula $\neg x_3 \wedge (x_0 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2)$.

1. Specify and implement a function `eval_clause` that takes a valuation ρ and a clause c as its arguments and returns `true` if c is true for valuation ρ and `false` otherwise.
2. Specify and implement a function `eval_cnf` that takes a valuation ρ and a formula c in conjunctive normal form as its arguments and returns `true` if c is true for valuation ρ and `false` otherwise.
3. Specify and implement a function `sat` that takes a formula c in conjunctive normal form and returns `None` if c is unsatisfiable and `Some` ρ otherwise, with ρ a valuation that makes c true.

Tips on completing this lab You are free to implement your Sheep SAT solver however you'd like, and are encouraged to think about a few different ways you might go about it before starting to implement. However, you will probably find this lab easier to complete if you keep your implementation simple, and avoid going for efficiency at this point. A fully-verified, brute-force recursive implementation will earn full points. The next lab will ask you to implement unit propagation, which will significantly improve the performance of your solver, and the final (optional) lab will have you implement an optimization of your choice, if you choose to do it. For this lab, you are advised to **keep things simple and focus on correctness.**

If you have not already encountered exceptions in Why3 when doing independent reading for the previous labs, then ou may find them useful here depending on your approach. To learn more about exceptions, and how they can be helpful in verifying search algorithms, consult Section 2.5 of the WhyML manual.

Resources You might find the following resources helpful as you complete the lab.

- The Why3 reference manual is available at `http://why3.lri.fr/manual.pdf`, and has recently been updated to cover the most recent version that you are using. In particular, Sections 2 and 6 cover the language features and give examples to help you understand their use.
- The Why3 standard library is available at `http://why3.lri.fr/stdlib`. Consult this if you aren't sure which operations are supported by imported modules, or how Why3 formalizes them with axioms.

- Toccata hosts a collection of verified programs, most of them in Why3, at `http://toccata.lri.fr/gallery`. You are free to study these examples, and adapt their proof techniques to your implementation. However, do not copy fragments directly from Toccata and use them in your solution. If in doubt about whether you might run afoul of academic integrity guidelines, consult with the course staff and add a comment to your solution linking to the solution that helped you complete that part of your implementation.

Remark on grading There is little work on specification in this lab and it should be much easier on the provers than the previous one[1]. Therefore, fully verified solutions will be rewarded more than usual.

Reminder As mentioned in Lab 1, you should **not** use the equality operator (`=`) to compare arrays. In order to express the fact that two arrays `t` and `t’` are equal, you should write:

```
length t = length t’ /\ forall i. 0 <= i < length t -> t[i] = t’[i]
```

or use the predicate `array_eq` that is defined in `array.ArrayEq`.

[1] We solved it using *Alt-Ergo* only.

```
type var = int

type lit = { var : var ; value : bool }

type clause = list lit

predicate vars_in_range (n : int) (c : clause) =
  forall l:lit. mem l c -> 0 <= l.var < n

type cnf = { clauses : array clause ; nvars : int }
  invariant { self.nvars >= 0 }
  invariant { forall i:int. 0 <= i < length self.clauses ->
              vars_in_range self.nvars self.clauses[i] }

type valuation = array bool

predicate valid_valuation (rho : valuation) (cnf : cnf) =
  length rho = cnf.nvars

predicate clause_sat_with (rho : valuation) (c : clause) =
  exists l:lit. mem l c && rho[l.var] = l.value

predicate sat_with (rho : valuation) (cnf : cnf) =
  forall i:int. 0 <= i < length cnf.clauses ->
  clause_sat_with rho cnf.clauses[i]

predicate unsat (cnf : cnf) =
  forall rho:valuation. valid_valuation rho cnf ->
  not (sat_with rho cnf)
```

Figure 2: Types and predicates for this lab

What to hand back

Do not forget to save the current proof session when exiting Why3 IDE. Before you do, though, use the "Clean" command of the IDE on the topmost node of your session tree in order to remove unsuccessful proof attempts. Then, generate a HTML summary of your proof session using the following command:

```
why3 session html simple-sat.mlw
```

This should create a HTML file in your session folder called "why3session.html". Open it and make sure that every goal you proved appears in green in the leftmost column. Finally, hand back an archive containing:

1. The completed `simple-sat.mlw` file
2. The session folder generated by Why3 IDE, including the HTML summary.
3. If appropriate, an ASCII text file `ReadMe.txt` containing any comment you may want to share with us

Additional remarks

- Having all your proof goals checked does **not** necessarily mean that you will get a perfect grade on your homework. Indeed, you also have to make sure that your specifications are correct and complete.
- We will manually inspect your code, so please make sure it is readable and leave comments.