# Lab 4 - Part II: verifying a SAT solver with unit propagation and improvements

15-414: Automated program verification

## Lab goals

In this lab, you will continue your work on verifying a SAT solver with unit propagation. You will also improve your SAT solver with **at least one** of the techniques described in this lab handout.

## Lab instructions

This lab is due on Tuesday, 4th December. You should now have a functional correct version of your SAT solver with unit propagation. However, to guarantee that your SAT solver is fully correct, you must augment your code with **correct and complete specifications** and prove their validity using Why3. Your SAT solver can be further improved by implementing **at least one** of the following techniques:

- Pure literal rule;
- Failed literal rule;
- Probing;
- Adding clauses via resolution;
- Variable elimination;
- Adjacency data structures.

These techniques are briefly described in this handout. [1] You can choose any of these techniques [2] and you are not restricted to the template provided in this lab handout (i.e., you can change the signature of the functions). Since fully verifying these additional improvements can be challenging, we will accept partially verified solutions. If you implement and verify more than one improvement or are able to fully verify your code than you will receive bonus points. The final submission of this lab will be used in the SAT verified competition where the authors of the best SAT solvers will receive Amazon vouchers.

# 1 Lab 4 - Part II

Every subsection below corresponds to a section in the template file. In this lab handout we only describe the sections for improving your SAT solver. We assume that this will be an extension to your existing code and you will reuse your fully verified SAT solver with unit propagation. Please refer to the previous lab handout for the description of the functions for unit propagation. Note that you do only need to implement **at least one** of these improvements. If you choose of the of the suggestions from Sections 1.1 to 1.5 than you should read Section 1.6 to see how you can integrate these techniques with your current SAT solver.

[1] We refer to the lecture notes for more details on each technique. Available at `https://www.cs.cmu.edu/~15414/lectures/20-sat-techniques.pdf`.

[2] If you want to implement a technique not in this list (e.g., clause learning, lazy data structures) then contact the instructors and we will provide you with further assistance.

### 1.1 Pure literal rule

Any atom that only appears in either positive or negative literals is called *pure*, and their corresponding atoms must always be assigned in a way that makes the literal *true*. Thus, they do not constrain the problem in a meaningful way, and can be assigned without making a choice. This is called *pure literal elimination* and is one type of simplification that can be applied to CNF formulas. Consider the following CNF formula:

$$\underbrace{(x_1 \vee x_2)}_{C_0} \wedge \underbrace{(\neg x_1 \vee x_2)}_{C_1} \wedge \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(\neg x_1 \vee x_2 \vee x_3)}_{C_3}$$

Notice that $x_3$ appears only as a positive literal in this formula. Hence, we can assign $x_3$ to *true* and satisfy the literal. This procedure will simplify the above formula into:

$$\begin{aligned} & (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_1 \vee x_3) \\ \leftrightarrow\ & (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \top) \wedge (\neg x_1 \vee x_1 \vee \top) \\ \leftrightarrow\ & (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \end{aligned}$$

**Task** Implement the `pure_literal_rule` function, which is specified as follows:

```
let pure_literal_rule (l: lit) (cnf: cnf) : bool =
```

This function should return true if the literal `l` is a pure literal in the CNF formula `cnf` and false otherwise. Notice that if a literal is pure than you can add it to the formula as a unit clause. You can use this function to check if there exists any pure literals in `cnf`. The pure literal rule is one of the simplest improvements that can be done and verified. However, oftentimes there are no pure literals in a formula and there may be no performance gains to `sat` solver.

### 1.2 Failed literal rule

BCP can also be used as a simplification technique. Let $\varphi$ be a propositional formula, and $l_i$ ($\neg l_i$) a literal to be propagated. If propagating $l_i$ ($\neg l_i$) in $\varphi$ leads to a conflict, than we

can conclude that $l_i$ ($\neg l_i$) must be assigned to *false* (*true*) in all interpretations I of $\varphi$. We call this procedure the *failed literal rule*.

Consider the following CNF formula:

$$\underbrace{(x_2 \vee x_3)}_{C_0} \wedge \underbrace{(\neg x_1 \vee \neg x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(x_0 \vee x_1 \vee \neg x_3)}_{C_3} \wedge \underbrace{(\neg x_0 \vee x_1 \vee x_3)}_{C_4}$$

Performing BCP with the interpretation $I = \{x_1\}$ leads to:

$$\begin{aligned}
&(x_2 \vee x_3) \wedge (\bot \vee \neg x_3) \wedge (\bot \vee \neg x_2 \vee x_3) \wedge (x_0 \vee \top \vee \neg x_3) \wedge (\neg x_0 \vee \top \vee x_3) \\
\leftrightarrow\ &(x_2 \vee x_3) \wedge (\neg x_3) \wedge (\neg x_2 \vee x_3) \\
\leftrightarrow\ &(x_2 \vee \bot) \wedge (\top) \wedge (\neg x_2 \vee \bot) \\
\leftrightarrow\ &(x_2) \wedge (\neg x_2)
\end{aligned}$$

Since propagating $x_1$ leads to a conflict and no other decision has been made, then it means that $x_1$ needs to be assigned to *false* in all interpretations of $\varphi$, i.e. we can add the unit clause $(\neg x_1)$ to the formula.

**Task** Implement the `failed_literal_rule` function, which is specified as follows:

```
let failed_literal_rule (l: lit) (cnf: cnf) : bool =
```

This function should return true if the literal `l` is a failed literal and false otherwise. Notice that you should take advantage of your `set_and_propagate` function to find out if a literal `l` is a failed literal or not. You can use this function to check if there exists any failed literals in `cnf`. Notice that if $l$ is a failed literal then you can add $(\neg l)$ as a unit clause to the formula.

### 1.3 Probing

BCP can also be used for *probing*. The key idea behind probing is to propagate $l_i$ and $\neg l_i$ and see if any literal $l_j$ is implied by both propagations. If this is the case than we can conclude that $l_j$ must be assigned to *true* in all interpretations of $\varphi$.

Consider the following CNF formula:

$$\underbrace{(x_1 \vee x_2)}_{C_0} \wedge \underbrace{(\neg x_1 \vee x_2)}_{C_1} \wedge \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(\neg x_1 \vee x_2 \vee x_3)}_{C_3}$$

When we propagate $x_1$ and $\neg x_1$, we have the following:

- $\text{BCP}(\varphi, x_1) \mapsto \{x_2\}$
- $\text{BCP}(\varphi, \neg x_1) \mapsto \{x_2\}$

Therefore, we can conclude that $x_2$ must be assigned to *true* in all interpretations of the formula.

**Task** Implement the `probing` function, which is specified as follows:

```
let probing (v: var) (cnf: cnf) : list lit =
```

This function should use `set_and_propagate` to propagate `v` with values `true` and `false`. The output should be the list of literals that are propagates with both values. The list should be empty if no such literal exists. These literals can then be added to the formula as unit clauses.

### 1.4 Adding clauses via resolution

We refer to the lecture notes of Lecture 20 on how to use resolution to derive resolvents:

`https://www.cs.cmu.edu/~15414/lectures/20-sat-techniques.pdf`

**Task** Implement the `resolve` function, which is specified as follows:

```
let resolve (a: clause) (b: clause) (l: lit) : option clause =
```

This function should resolve this two clauses and return its resolvent. If the clauses cannot be resolved (i.e., the literal `l` does not appear with different values in `a` and `b`) then the function should return `None`.

It may also be helpful to check if the resolved clause is a *tautology*. A clause is a tautology if it contains a literal `l` with both `true` and `false` values.

**Task** Implement the `tautology` function, which is specified as follows:

```
let tautology (a: clause) : bool =
```

This function returns true if the clause `a` is a tautology and false otherwise.

### 1.5 Variable elimination

We refer to the lecture notes of Lecture 20 on how to use resolution to perform variable elimination:

`https://www.cs.cmu.edu/~15414/lectures/20-sat-techniques.pdf`

**Task** Implement the `eliminate` function, which is specified as follows:

```
let eliminate (v: var) (cnf: cnf) : cnf =
```

This function will eliminate variable `v` from the `cnf` formula and return a new formula with one less variable and with the resolvents of all clauses that contained the literal with `var=v` and `value=true/false`.

You may want to declare an auxiliary function that returns the list of clauses of all resolvents:

**Task** Implement the `eliminate_clauses` function, which is specified as follows:

```
let eliminate_clauses (v: var) (cnf: cnf) : list clause =
```

This function will return the resolvents of all clauses that contained the literal with `var=v` and `value=true` with the clauses that contained the this literal with `value=false`. See the lecture notes for an example of variable elimination. Note that in practice, you may want to perform bounded variable elimination, i.e. if the number of clauses that you eliminated from the formula is smaller or equal to the number of clauses that you are adding to the formula than you eliminate the variable; otherwise you don't. Eliminating all variables will incur an exponential blow up in memory and is not advisable for practical SAT solving. Variable elimination is one of the most important simplification techniques used by SAT solvers. However, its implementation and verification can be very *challenging*.

### 1.6 Putting all pieces together

All the techniques described up to this point can be seen as transforming a formula $\varphi$ into an equivalent or equisatisfiable formula $\phi$. Note that two formulas $\varphi$ and $\phi$ are equisatisfiable iff when $\varphi$ is satisfiable then $\phi$ is also satisfiable. However, the interpretation that satisfies $\varphi$ does not need to be the same as the one that satisfies $\phi$. Equisatisfiability is a weaker specification than equivalence [3] but will be accepted for this lab. When proving formula equivalence, you would need to prove that if a given interpretation $I$ satisfies $\varphi$ then the same interpretation will satisfy $\phi$.

**Task** Implement the `transform` function, which is specified as follows:

```
let transform (cnf : cnf) : cnf =
```

You can use either a specification for formula equivalence:

```
  ensures  { forall rho:valuation. sat_with rho cnf <-> sat_with rho result }
  ensures  { unsat cnf <-> unsat result }
```

Or a specification for equisatisfiability:

```
  ensures  { forall rho:valuation. sat_with rho cnf ->
                                   exists rho':valuation. sat_with rho' result }
  ensures  { forall rho':valuation. sat_with rho' result ->
                                   exists rho:valuation. sat_with rho result }
  ensures  { unsat cnf <-> unsat result }
```

**Challenge** Can you verify either equisatisfiability or equivalence of your transformation?

You can then integrate the transform function inside the sat function by calling transform before solving the CNF formula:

```
let sat (cnf : cnf) : option valuation =
  ensures  { forall rho:valuation. result = Some rho -> sat_with rho cnf }
```

[3] From the techniques presented in this handout only variable elimination does not preserve equivalence.

```
ensures  { result = None -> unsat cnf }
...
let cnf' = transform cnf in
```

**Challenge** Can you still verify your `sat` function when transforming the formula?

### 1.7 Adjacency data structures

Contrary to the previous optimizations, changing the underlying data structure will not modify the CNF formula in any way. One optimization that we discuss in the lecture notes is to associate each clause with 2 counters. One counter will count the number of current satisfied literals, whereas the other counter will count the number of unsatisfied literals.

**Task** Create new types that will map each clause to 2 counters. You will also need a map between a literal and a list of clauses than contain that literal.

**Task** Update your code to take advantage of this information:

- `partial_eval_clause` can be defined without recursion just by checking the counters
- `backtrack` needs to update the counters by decreasing the corresponding counters of clauses that contains the literals that are unassigned
- `set_and_propagate` needs to update the counters by increasing the corresponding counters of clauses that contains the literals that are assigned

**Challenge** Can you make all your functions verify with these changes? Note that modifying the above functions to use new types will have a cascade effect and may require *significant effort* to be verified. The main advantage of this technique is that avoids the recursive procedure of `partial_eval_clause` and will significantly speedup your unit propagation procedure.

## 2 What to hand back

Since we will use different criteria for the specifications that guarantee the correctness of your SAT solver with unit propagation (functions from Lab 4 - Part I) and for the functions describing the improvements (where we will accept partially verified solutions), you must submit two files, `unit-sat.mlw` and `final-sat.mlw`.

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 sessions using the following command:

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

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

This should create a HTML file in your session folder. 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 `unit-sat.mlw` file
2. The completed `final-sat.mlw` file (this will be mostly a copy of `unit-sat.mlw` with some additional improvements). This will be the file used for the verified SAT competition
3. The session folder generated by Why3 IDE for both files, including a HTML summary
4. If appropriate, an ASCII text file `ReadMe.txt` containing any comment you may want to share with us