Boolean Satisfiability and the DPLL Algorithm

The goal of this project is to implement and verify a tiny SAT solver using the Davis–Putnam–Logemann–Loveland (DPLL) algorithm[1].

This project is to be carried out using the Why3 tool (version 1.8.2), in combination with automated provers (Alt-Ergo 2.6.0, CVC5 1.1.2 and Z3 4.13.3 or higher). You can use other automatic provers or versions if you want, if they are freely available and recognized by Why3. You may use Rocq for discharging particular proof obligations.

The project must be done individually: team work is not allowed. In order to obtain a grade for the project, you must send an e-mail to `andrei.paskevich@lmf.cnrs.fr` and `jean-marie.madiot@inria.fr`, no later than **Thursday, February 5, 2026** at 23:00 UTC+1. This e-mail should be entitled "MPRI project PROGPROOFS", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- A Why3 file `dpll.mlw` containing your solution.
- The content of the sub-directory `dpll` generated by Why3. In particular, this directory should contain session files `why3session.xml` and `why3shapes.gz`, and Rocq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work. The contents of this report counts for at least half of your grade for the project.

The report must be written in French or English, and should typically consist of 3 to 6 pages. The structure should follow the sections and the questions of the present document. For each question, detail your approach, focusing in particular on the design choices that you made regarding the implementations and specifications. In particular, loop invariants and assertions that you added should be explained in your report: what they mean and how they help to complete the proof.

A typical answer to a question or step would be: *"For this function, I propose the following implementation: [give code or pseudo-code]. The contract of this function is [give a copy-paste of the contract]. It captures the fact that [rephrase the contract in natural language]. To prove this code correct, I need to add extra annotations [give the loop invariants, etc.] capturing that [rephrase the annotations in English]. This invariant is initially true because [explain]. It is preserved at each iteration because [explain]. The post-condition then follows because [explain]."*

The reader of your report should be convinced at each step that the contracts are the right ones, and should be able to understand why your program is correct, e.g., why a loop invariant is initially true, why it is preserved, and why it suffices to establish the post-condition. You can quote parts of your Why3 code in the report, yet you should only copy the most relevant parts, not all of your code. In case you are not able to fully complete a definition or a proof, you should carefully describe which parts are missing and explain the problems that you faced.

In addition, your report should contain a conclusion, providing general feedback about your work: how easy or how hard it was, what were the major difficulties, was there any unexpected result, and any other information that you think is important to consider for the evaluation of the work you did.

1 SAT Solver

We consider propositional logic formulas over n variables noted $x_1, x_2, \ldots, x_n$. Each of these variables can take either the value **true** (noted $\top$) or **false** (noted $\bot$). A **literal**, noted ℓ, is a variable x_i or its negation noted $\neg x_i$, the negation being defined by $\neg\top = \bot$ and $\neg\bot = \top$.

[1] https://en.wikipedia.org/wiki/DPLL_algorithm

A **clause** is a disjunction of literals, such as $x_1 \vee \neg x_2 \vee x_3$. It is true if at least one of its literals is true. The empty clause, which contains no literal, is not true. In the following, we assume that each clause contains **at most one occurrence of each variable**, either as x_i or as $\neg x_i$ but not both.

Finally, a formula in **conjunctive normal form** (CNF) is a conjunction of clauses. Here is an example of a CNF with four clauses:

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \tag{1}$$

A formula is true is all its clauses are true. The empty formula, with no clause at all, is true. In the following, we only consider formulas in conjunctive normal form.

We say that a formula is **satisfiable** if it is possible to assign a truth value (true or false) to its variables in a way that makes the formula true. For instance, the formula (1) above is satisfiable, since the assignment $\{x_1 = \bot,\ x_2 = \bot,\ x_3 = \top\}$ makes each of the four clauses true. On the contrary, the formula

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \tag{2}$$

is **unsatisfiable**, because none of the eight ways to assign a truth value to the three variables x_1, x_2, x_3 is able to satisfy the formula. Our goal is to implement and verify a program to determine the satisfiability of a given formula. Such a program is called a **SAT solver**.

The DIMACS Format. A CNF formula is simply represented in a text file using the DIMACS format. A literal x_i is represented by the integer i and the literal $\neg x_i$ by the integer $-i$. A clause is a single line of text containing space-separated literals, and terminated by the integer 0. A line starting with the character `c` is a comment. Last, the clauses is preceded by a line starting with `p cnf` giving the number of variables and the number of clauses. Here is a DIMACS file for the formula (1):

```
c example of satisfiable formula
p cnf 3 4
1 -2 0
2 3 0
-1 -3 0
-1 -2 3 0
```

A set of formulas in DIMACS format is given. The subdirectory `sat` contains satisfiable formulas and the subdirectory `unsat` contains unsatisfiable formulas. Some OCaml code is provided (in file `test_dpll.ml`) to parse a DIMACS file, run your SAT solver over it, and print either `SAT` or `UNSAT` on the standard output. A bash script (see file `check`) is also provided to test the resulting program on all the given DIMACS files.

3-SAT. To make your project slightly simpler, we only consider CNF formulas where each clause has **exactly three literals**. The OCaml DIMACS parser is taking care of turning the input formula into this restricted format. For this purpose:

- A clause that contains only one or two literals is extended to three literals with the literal x_0, where the variable x_0 is considered to be **always false**.
- A clause that contains more than three literals is split into several clauses, using fresh variables. For instance, the clause $a \vee b \vee c \vee d$ can be translated to two clauses $a \vee b \vee x_i$ and $\neg x_i \vee c \vee d$, where i is a fresh variable.

2 DPLL Algorithm

We are going to determine the satisfiability of a CNF formula using Davis–Putnam–Logemann–Loveland algorithm, known as DPLL. It uses the technique of *backtracking*. This algorithm is described later.

We set the following Why3 types:

```
type var = int
type lit = int (* -n .. n *)
type cls = (lit, lit, lit)
type assignment = array int
```

If n is the number of variables, a variable (type `var`) is an integer between 0 and n (remember that 0 is our fake variable for $\bot$). A literal (type `lit`) is an integer: the value 0 stands for the literal $x_0 = \bot$; the value $i > 0$ stands for the literal x_i; and the value $i < 0$ stands for the literal $\neg x_{-i}$. A clause (type `cls`) is a triple of literals, and we store our clause set in an array.

A truth assignment (type `assignment`) is an array of size $n + 1$, whose first cell (at index 0) is always zero. For a non-zero index $1 \leq i \leq n$, the cell i contains the current assignment to variable i, which can be either i or $-i$. If $x_i = \top$, then the cell i contains the value $-i$. If $x_i = \bot$, then the cell i contains the value i. This way, the assignment $\{x_1 = \bot,\ x_2 = \bot,\ x_3 = \top\}$ is represented by the following array:

0	1	2	-3

Note that the truth of any literal ℓ between $-n$ and n, including 0, is equivalent to the property "the value in the cell $|\ell|$ differs from ℓ".

Throughout our solution, we will use the following identifiers:

`cl` — an array of clauses that contains the initial clause set, possibly in a different order.

`mm` — the current variable assignment.

`nv` — the length of the array `mm`, equal to $n + 1$.

`na` — an integer between 1 and n, indicating the first variable that has not yet received an assignment.

`nc` — an integer between 0 and the length of `cl`, indicating the number of "active" clauses in `cl`, i.e., those that are not satisfied by the current assignment `mm`.

DPLL Algorithm. This is a recursive algorithm, that uses the technique of backtracking. As input, it takes a CNF formula F and a partial assignment A of its variables, that is **consistent** with F: no clause of F has all its three literals unsatisfied by A. As output, we have a Boolean value that is `true` if and only if it is possible to complete A so that F is satisfied. We proceed as follows:

1. We choose a literal ℓ not yet assigned in A.
2. We add ℓ to A. If all the clauses of F are now satisfied, we return `true`. Otherwise, we call DPLL recursively. In case of success, we return `true`.
3. Otherwise, we rather add the negation of ℓ to A and we do the same thing. In case of a new failure, we return `false`.

To implement this algorithm, we are going to remove clauses from F as soon as they are satisfied, so that we do not reconsider them in the future. This means we have to **restore** such clauses in case of failure in point 2 or 3 above. This is where we use the `nc` parameter. It indicates that only the first `nc` clauses in the array `cl` are to be considered; the remaining clauses (i.e., from index `nc` and beyond) are already satisfied by the partial assignment. Initially, `nc` is equal to the number of clauses. To remove the clause at index i, we simply have to decrement `nc` and swap the two clauses at indices i and `nc`. And to restore the removed clauses, it is enough to give `nc` its old value. **We do not have to redo the swaps!**

You must implement and verify a SAT solver as a following WhyML function:

```
let sat (mm: assignment) (cl: array cls) : bool
```

The returned value must be **`true`** if and if the set of causes is satisfiable. In that case, the array `mm` must contain an assignment that satisfies the formula. (When the function returns **`false`**, the contents of `mm` is meaningless.)

The draft file `dpll.mlw` contains several definitions and prototypes that may help you in your task. Read its contents attentively.

Note. A traditional implementation of DPLL also includes two other ideas: propagation of unit clauses (as soon as a clause is reduced to a single literal, the value of that literal is imposed) and detection of pure literals (if a variable appear only positively or only negatively in all clauses, it can be removed everywhere).

3 Conclusion

Don't forget to end your report with a conclusion that summarizes your achievements, explain the issues you couldn't solve if any, and comment about what you learned when doing this project.