Lab 4: verifying a SAT solver with unit propagation

15-414: Automated program verification

Lab goals

In the previous lab, you wrote a provably correct brute-force search SAT solver. This lab is about making it more efficient by writing and verifying a minimalistic implementation of the DPLL algorithm, which forms the basis of most modern SAT-solvers.

Lab instructions

This lab will have two parts. The first part is due on Monday, April 20th. In the first part, you must submit a working version of all methods in this lab. Note that you do not need to fully verify them at this stage. The goal of the first part is for you to get feedback on the performance of your SAT solver and on the parts of your code that are still unverified. To assess if your code is performing as expected, we will provide some SAT formulae and instructions on how to compile your Why3 code to OCaml and solve those formulae. Your algorithm must answer satisfiable/unsatisfiable correctly for each formula.

Since we want to guarantee the correctness of your SAT solver, you must submit the second part of this lab on Friday, May 1 with all your methods verified. For extra credit, you may also improve your current SAT solver with a technique from the note for Lecture 19.

1 Introduction

1.1 A reminder on DPLL and unit propagation

We define a *partial valuation* as a partial function from variable identifiers to booleans. A variable which is not mapped to a value is said to be *unassigned*. Besides, a literal x_i or $\neg x_i$ is said to be unassigned if and only if x_i is unassigned. Given a partial valuation, a clause is said to be

- *satisfied* if one or more of its literals are satisfied
- *conflicting* if all its literals are assigned but not satisfied
- *unit* if it is not satisfied and all but one of its literals are assigned
- *unresolved* otherwise.

The DPLL algorithm enhances a naive backtracking search algorithm by implementing an optimization called *unit propagation*: if a clause becomes unit during the search process, it can only be satisfied by making its unique unassigned literal true and so no branching is necessary. In practice, this rule often applies in cascade, which can reduce the search space greatly. An example run of the DPLL algorithm is shown Figure 1.

Other optimizations are sometimes associated with the DPLL algorithm, most notably *pure literals propagation*[1] and *conflict clauses learning*. You are **not** expected to implement them, although they would make for natural extensions of this lab.

[1] A literal is said to be *pure* if its negation does not occur in any clause that is not yet satisfied.

$$F = \overbrace{(x_2 \vee x_3)}^{C_0} \wedge \overbrace{(\neg x_1 \vee \neg x_3)}^{C_1} \wedge \overbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}^{C_2} \wedge \overbrace{(x_0 \vee x_1 \vee \neg x_3)}^{C_3} \wedge \overbrace{(\neg x_0 \vee x_1 \vee x_3)}^{C_4}$$

Step	**Partial valuation**
Start with an empty partial valuation.	$\{\}$
Decide x_0.	$\{x_0 \mapsto \texttt{true}\}$
Decide x_1.	$\{x_0 \mapsto \texttt{true}, x_1 \mapsto \texttt{true}\}$
Propagate $\neg x_3$ from unit clause C_1.	$\{x_0 \mapsto \texttt{true}, x_1 \mapsto \texttt{true}, x_3 \mapsto \texttt{false}\}$
Propagate x_2 from C_0.	$\{x_0 \mapsto \texttt{true}, x_1 \mapsto \texttt{true}, x_3 \mapsto \texttt{false}, x_2 \mapsto \texttt{true}\}$
Clause C_2 is conflicting. Backtracking.	$\{x_0 \mapsto \texttt{true}\}$
Decide $\neg x_1$.	$\{x_0 \mapsto \texttt{true}, x_1 \mapsto \texttt{false}\}$
Propagate x_3 from C_4.	$\{x_0 \mapsto \texttt{true}, x_1 \mapsto \texttt{false}, x_3 \mapsto \texttt{true}\}$
Every clause is satisfied: F is satisfiable.	$\{x_0 \mapsto \texttt{true}, x_1 \mapsto \texttt{false}, x_3 \mapsto \texttt{true}, x_2 \mapsto *\}$

Figure 1: Unit propagation in action

1.2 General remarks and advice

- Your SAT-solver only has to be proven correct in the sense that it should be guaranteed to always terminate without runtime errors and with the right result. This does not imply anything about its efficiency or about unit propagation being implemented properly. In particular, your code must respect the following properties, although we expect no formal guarantees for them:
  - branching must never happen in the presence of unit clauses
  - partial valuations must not be copied in an abusive way (see section 2.5)

  If your code checks and we can convince ourselves that the two properties above hold, you will (likely) get full credit.

- In order to make verification and debugging easier, you should keep your code as simple, concise and readable as possible. In particular,

– no single function should be more than 40 or 50 lines[2], specification included

– if an `ensures` or `requires` statement takes more than two lines to write, you may consider introducing some auxiliary predicates to make it simpler.

[2]Also, no line should be longer than 80 characters.

2 Lab 4

Every subsection below corresponds to a section in the template file.

2.1 A useful lemma

The template opens with a lemma that you may find useful for proving some termination properties as you progress in the lab. This lemma states the following: when you modify an array by updating a single cell from a value v to a different value, the number of occurences of v in this array decreases by one. Obvious, isn't it ?

In order to formalize this statement, we use the `numof` function from the `array.NumOfEq` module of the standard library. If t is an array, v is a value and i, j are nonnegative integers, `numof` t v i j denotes the number of occurences of v in t between indices i (included) and j (excluded). For example, if $t = [0, 1, 0, 2, 0, 0]$, then `numof` t 0 1 5 $= 2$.

Remarks

- We define the following useful shortcut:
  `function total_numof (t : array 'a) (v : 'a) : int = numof t v 0 (length t)`
- Because `numof` is defined by a set of axioms (like `fib` in Lab 1), `numof` and `total_numof` cannot be used in code and must only appear in annotations.

With these definitions, our lemma of interest can be expressed as follows:

```
lemma numof_decreases: forall v : 'a, t t' : array 'a, i : int.
  length t = length t' ->
  (0 <= i < length t /\ t[i] = v /\ t'[i] <> v) ->
  (forall j:int. 0 <= j < length t -> j <> i -> t[j] = t'[j]) ->
  total_numof t' v = total_numof t v - 1
```

This is all fine, except that no automated prover is able to prove this without help. Indeed, proving this from the axioms of `numof` requires a proof by induction featuring a non-trivial case disjunction. How can we give provers a hint about this?

Consider the following lemma:

```
lemma option_inversion:
    forall opt : option 'a. (opt = None) \/ (exists x : 'a. opt = Some x)
```

A bit of thought should convince us that proving this lemma is equivalent to proving the correctness of any function with the following specification:

```
let option_inversion_f (opt : option 'a) : unit
    ensures { (opt = None) \/ (exists x : 'a. opt = Some x) }
```

Then, it is possible to use the body of such a function to hint the structure of a proof for `option_inversion`. In this case, we can force the prover into doing a case disjunction on `opt` by providing the following body for `option_inversion_f`:

```
let option_inversion_f (opt : option 'a) : unit =
    ensures { (opt = None) \/ (exists x : 'a. opt = Some x) }
    match opt with
      | None   -> ()
      | Some _ -> ()
    end
```

Why3 supports writing and proving lemmas this way. Indeed, a pure function with return type `unit` can be automatically turned into a lemma by adding the keyword `lemma` in its declaration. Such a function is called a *function-lemma*, or sometimes a *let-lemma*.

Task Figure 2 shows the template of a function-lemma for `numof_decreases`. Complete the four blanks in it to make it check (each blank should be filled with a small expression). Observe how the structure of this lemma-function mimics the structure of a proof for `numof_decreases`.

```
let lemma numof_decreases (v : option bool) (t t' : array (option bool)) (i : int) =
  requires { length t = length t'  }
  requires { 0 <= i < length t /\ t[i] = v /\ t'[i] <> v }
  requires { forall j:int. 0 <= j < length t -> j <> i -> t[j] = t'[j] }
  ensures  { total_numof t' v = total_numof t v - 1 }

  let rec aux (j : int) : int =
    variant  { j }
    requires { 0 <= j <= length t }
    ensures  { result = numof t v 0 j }
    ensures  { j <= i -> numof t' v 0 j = result }
    ensures  { j > i  -> numof t' v 0 j = result - 1 }

    if j = 0 then ...
    else if j = i+1 then ...
    else if t[j-1] == v then ...
    else ...

  in let _ = aux (length t) in ()
```

Figure 2: Template for the `numof_decreases` function-lemma

2.2 Basic types and predicates

This section of the template, which defines formulas in conjunctive normal forms and satisfiability, is copied from Lab 3. Indeed, we expect your new SAT-solver to have the exact same signature and specification it had in the previous lab:

```
let sat (cnf : cnf) : option valuation =
  ensures  { forall rho:valuation. result = Some rho -> sat_with rho cnf }
  ensures  { result = None -> unsat cnf }
```

Only its implementation should change.

2.3 Partial valuations

In order to implement unit-propagation, we need a type to represent partial valuations. Not surprisingly, it is defined as follows in the template:

```
type pval = array (option bool)
```

This is to be contrasted with the type definition for valuations:

```
type valuation = array bool
```

The template also provides a few useful predicates on partial valuations. The most important one is certainly the `compatible` predicate, which is defined as follows:

```
predicate compatible (pval : pval) (rho : valuation) =
    forall i:int, b:bool. 0 <= i < length pval ->
    pval[i] = Some b -> rho[i] = b
```

More informally, a partial valuation p is said to be *compatible* with a valuation ρ if both agree on every variable which is assigned by p. In particular, an empty partial valuation is compatible with any valuation.

2.4 Partial evaluation of clauses and formulas

We want to define a function `partial_eval_clause` that takes a partial valuation p along with a clause C as its arguments and returns:

- [`Satisfied`] if and only if p satisfies C
- [`Conflicting`] if and only if p and C are conflicting
- [`Unit` l] if C is a unit clause with unassigned literal l (for partial valuation p)
- [`Unresolved`] in every other case.

This corresponds to the following type definition:

```
type clause_status = Satisfied | Conflicting | Unit lit | Unresolved
```

Similarly, we want to define a function `partial_eval_cnf` that takes a partial valuation p along with a CNF formula F as its arguments and returns:

- [`Sat`] if and only if p satisfies every clause of F. In this case, F is true for every valuation that is compatible with p and the search can stop.
- [`Conflict`] if p is conflicting with at least one clause of F. In this case, F is false for every valuation that is compatible with p and backtracking is needed.
- [`Unit_clause` l] only if F admits a unit clause whose unassigned literal is l. If F admits more than one unit clause, which one is featured in the argument of `Unit_clause` is unspecified.[3]
- [`Other`] in every other case.

This corresponds to the following type definition:

```
type cnf_status = Sat | Conflict | Unit_clause lit | Other
```

[3] An alternative that would also enable a slightly faster implementation of unit propagation would be for `Unit_clause` to take the list of **every** unit literal as an argument. This would introduce some additional complexity though and we do not recommend taking this path.

A direct way to turn these definitions into a specification for the `partial_eval_cnf` function is to define the `cnf_satisfied`, `cnf_conflicting` and `cnf_has_unit_clause` predicates as in Figure 3 and use them as follows:

```
let partial_eval_cnf (pval : pval) (cnf : cnf) : cnf_status =
  requires { valid_pval pval cnf }
  ensures { result = Sat -> cnf_satisfied pval cnf }
  ensures { result = Conflict -> cnf_conflicting pval cnf }
  ensures { forall l. result = Unit_clause l -> cnf_has_unit_clause pval cnf }
  ensures { ... }
```

We do **not** recommend this approach though. Indeed, the specification of `sat` relies mostly on the `sat_with` predicate and so you would have to write many auxiliary lemmas to help the provers understand the connection between `sat_with` and the definitions from Figure 3. Besides, the validity of the unit propagation rule does not follow trivially from the definition of `cnf_has_unit_clause` that is given Figure 3. A better and maybe more elegant approach is to use `sat_with` directly to specify `partial_eval_cnf`. For example, the first `ensures` statement above could be replaced by:

```
result = Sat -> forall rho:valuation. compatible pval rho -> sat_with rho cnf
```

Task Specify and implement the `partial_eval_clause` and `partial_eval_cnf` functions. Keep in mind that you do not need to write complete specifications but only specifications that are strong enough for proving the correctness of the code that depends on these functions.

```
predicate lit_is_true (pval : pval) (l : lit) = pval[l.var] =
  Some l.value

predicate lit_is_false (pval : pval) (l : lit) = pval[l.var] =
  Some (not l.value)

predicate clause_satisfied (pval : pval) (c : clause) =
  exists l:lit. mem l c /\ lit_is_true pval l

predicate clause_conflicting (pval : pval) (c : clause) =
  forall l:lit. mem l c -> lit_is_false pval l

predicate clause_is_unit (l : lit) (pval : pval) (c : clause) =
  (mem l c /\ pval[l.var] = None) /\
  (forall l':lit. mem l' c -> l <> l' -> lit_is_false pval l')

predicate cnf_satisfied (pval : pval) (cnf : cnf) =
  forall i. 0 <= i < length cnf.clauses ->
  clause_satisfied pval cnf.clauses[i]

predicate cnf_conflicting (pval : pval) (cnf : cnf) =
  exists i. 0 <= i < length cnf.clauses /\
  clause_conflicting pval cnf.clauses[i]

predicate cnf_has_unit_clause (l : lit) (pval : pval) (cnf : cnf) =
  exists i. 0 <= i < length cnf.clauses /\
  clause_is_unit l pval cnf.clauses[i]
```

Figure 3: A naive way to define the status of clauses and CNFs (do **not** do this)

2.5 Backtracking

In the DPLL algorithm, when a conflict arises during search, one has to backtrack before the last decision point. A naive way to do so would be to create a full copy of the current partial valuation every time a choice is made but this would be terribly inefficient. A better alternative is to maintain a list of every variable that has been assigned since the the last decision point and to use this list as a reference for backtracking.

Let p and p' two partial valuations and l a list of variables. We say that l is a *delta* from p to p' if p and p' agree outside of l and the variables of l are unassigned in p. This can be formalized in Why3 as follows:

```
predicate delta (diff : list var) (pval pval' : pval) =
  (length pval = length pval') /\
  (forall v:var. mem v diff -> 0 <= v < length pval /\ not (assigned pval v)) /\
  (forall v:var. 0 <= v < length pval -> not (mem v diff) -> pval[v] = pval'[v])
```

Then, we can define a function `backtrack` that restores an older version of a partial valuation given a delta from this older version to the current one:

```
let rec backtrack (diff : list var) (pval : pval) (ghost old_pval : pval) =
  requires { delta diff old_pval pval }
  ensures  { pval_eq old_pval pval }
```

Note that `old_pval` is a *ghost argument*, which means that it will be eliminated during compilation. Therefore, it cannot be used in the body of `backtrack` but only in its specification. However, as opposed to `diff` and `pval`, it can be instantiated with ghost code. You can find Figure 4 an example of how to use the `backtrack` function. Pay attention to how ghost code is used in this example. In particular, no array is copied at runtime!

Task Implement the `backtrack` function.

```
let backtrack_example (pval : pval) : unit =
  requires { length pval >= 2 }
  requires { not (assigned pval 0) /\ not (assigned pval 1) }
  ensures  { pval_eq (old pval) pval }

  let ghost old_pval = Array.copy pval in
  pval[0] <- Some true ;
  pval[1] <- Some false ;
  let diff = Cons 0 (Cons 1 Nil) in
  backtrack diff pval old_pval
```

Figure 4: An example of using the `backtrack` function

2.6 Unit propagation

In this section, we want to implement a function `set_and_propagate` with the following signature:

```
let rec set_and_propagate (l : lit) (pval : pval) (cnf : cnf) : (bool, list var)
```

This functions takes as its arguments an unassigned literal l and the current partial valuation p. It updates p by setting literal l to true and then recursively performing unit propagation until a conflict is reached or no unit clause remains. Besides:

- It raises a `Sat_found` exception in case the CNF becomes satisfied.
- It returns a tuple whose first component is a boolean that is `true` if and only if a conflict was reached and whose second component is the delta of p (the list of every variable that was assigned during the call to `set_and_propagate`).

To go back to the example of Figure 1, calling `set_and_propagate` for literal x_1 and with `pval` $= \{x_0 \mapsto$ `true`$\}$ updates `pval` to $\{x_0 \mapsto$ `true`, $x_1 \mapsto$ `true`, $x_3 \mapsto$ `false`, $x_2 \mapsto$ `true`$\}$ and returns the tuple (`true`, $[2, 3, 1]$).

Task Specify and implement the `set_and_propagate` function. Once again, the specification does not have to be complete but only strong enough to prove the functional correctness of `sat`.

2.7 Putting all the pieces together

Now, it is times for you to put all the pieces together!

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

```
let sat (cnf : cnf) : option valuation =
  ensures  { forall rho:valuation. result = Some rho -> sat_with rho cnf }
  ensures  { result = None -> unsat cnf }
```

3 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 unit-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 session folder generated by Why3 IDE, including a 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.
- All your archives will be inspected manually, so please make sure it is readable and leave comments.