Mini-Project 2
Decision Procedures

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Friday, April 5, 2024 (checkpoint)
23:59pm, Friday, April 19, 2024 (final)
150 pts

You may, but are not required to, do this assignment with a partner.
The mini-projects have two due dates:

- Checkpoint at 23:59pm, Fri Apr 5, 2024 (70 pts)
- Final projects at 23:59pm, Fri Apr 19 2024 (80 pts)

No late days may be used on the checkpoint portion of the project. You may recover up to 40 of the points you lost at the checkpoint if you revise the first part with your final submission. For the checkpoint, having a fully verified version may be challenging but you can recover the points regarding verification. However, your implementation should be working and we provide a set of CNF formulas for you to test your '`sat`' function.

The mini-project must be submitted electronically on Gradescope. Please carefully read the policies on collaboration and credit on the course web pages at `http://www.cs.cmu.edu/~15414//assignments.html`.

If you are working with a partner, only one of the two of you needs to submit to each Gradescope assignment. Once you have uploaded a submission, you should select the option to add group members on the bottom of the screen, and add your partner to your submission. Your partner should then make sure that they, too, can see the submission.

> As before, we give the advice that: **Elegance is not optional!** For writing verified code, this applies to both: the specification and the implementation.

The Code

In each problem, we provide some suggested module outlines, but your submitted modules may be different. For example, where we say '`let`' it may actually be '`let rec`', or '`predicate`', etc. However, you **cannot make any of the main functions for each task pure** (i.e., you cannot use '`function`'). You may modify the order of the functions or provide auxiliary types and functions. You may also change the type definitions or types of the function, but in this case, you should justify the change in your writeup. We recommended your functions to raise exceptions for convenience since this would make it easier to stop once the condition was met. However, depending on your implementation you may want to return a value instead of raising an exception. We leave this choice up to you. **Note that you should not write axioms for this assignment.**

The Writeup (20 pts)

The writeup should consist of the following sections:

1. **Executive Summary.** Which problem did you solve? Did you manage to write and verify all functions? If not, where did the code or verification fall short? Which were the key decisions you had to make? What ended up being the most difficult and the easiest parts? What did you find were the best provers for your problem? What did you learn from the effort?
2. **Code Walk.** Explain the relevant or nontrivial parts of the specification or code. Point out issues or alternatives, taken or abandoned. Quoting some code is helpful, but avoid "core dumps." Basically, put yourself into the shoes of a professor or TA wanting to understand your submission (and, incidentally, grade it).
3. **Recommendations.** What would you change in the assignment?

Depending on how much code is quoted, we expect the writeup to consist of about 3-5 pages in the lecture notes style.

What To Hand In

You should hand in the following files on Gradescope:

- Submit the file `mp2.zip` to MP2 Checkpoint (Code) for the checkpoint and to MP2 Final (Code) for the final handin. We will be looking for files `sat.mlw`. Use `make handin` to create the handin file.
- Submit a PDF containing your final writeup to MP2 Final (Written). There is no checkpoint for the written portion of the assingment. You may use the file `mp2-sol.tex` as a template and submit `mp2-sol.pdf`. Use `make sol` to create the writeup file.

  Make sure your session directories and your PDF solution files are up to date before you create the handin file.

Using LaTeX

We prefer the writeup to be typeset in LaTeX, but as long as you hand in a readable PDF with your solutions it is not a requirement. We package the assignment source `mp2.tex` and a solution template `mp2-sol.tex` in the handout to get you started on this.

1 SAT Solver

A *SAT solver* uses a decision procedure to establish the satisfiability of a propositional formula. The goal of this project is to implement a SAT solver based on DPLL and unit propagation 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.

A reminder on DPLL and unit propagation. We define a *partial valuation* as a partial function from variable identifiers to booleans. A variable that 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.

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

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

More details about the DPLL algorithm and unit propagation are available in Lecture 16 notes.

1.1 SAT solver with partial valuations (Checkpoint: 70 pts)

In Assignment 5, you specified and implemented some simple operations that can be performed over formulas in CNF. In that assignment you considered complete valuations, however, in practice a SAT solver uses partial valuations. In this project, we will start by considering the same types as before. You may reuse any code from Assignment 5. All code that you write for the checkpoint should be in the module `Sat`.

```
type var = int
type lit = { var : var ; sign : bool }
type clause = list lit
type cnf = { clauses : array clause ; nvars : int }
type valuation = array bool
```

To make it easier for this assignment, we provide in the code template the data structure invariants for the type `cnf` as well as basic predicates (`valid_valuation`, `clause_sat_with`, `sat_with`, and `unsat`). We recommend using these predicates for your specifications.

Partial valuations. A variable in a partial valuation can take values *True* or *False* if it is assigned a value, or *None* if is unassigned. A complete valuations relates a with partial valuation as follows. A partial valuation 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.

```
type pval = array (option bool)

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

Task 1 (10 pts). A partial valuation that satisfies a CNF formula can be extended to a complete valuation by assigning the unassigned variables to any truth value. Implement, specify and verify a function `extract_sat_valuation` that given a partial valuation `pval` that satisfies the formula `cnf` returns a complete valuation that also satisfies the formula `cnf`.

```
let extract_sat_valuation (pval : pval) (ghost cnf : cnf) : valuation
```

Task 2 (10 pts). Implement, specify and verify 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
- [`Unresolved`] in every other case.

This corresponds to the following type and function definition:

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

let rec partial_eval_clause (p : pval) (c : clause) : clause_status
```

Note that your specification only needs to prove implications and not equivalences. For instance, you only need to prove that if the result is something then that implies something else. For instance:

```
ensures  { result = Satisfied -> ... }
```

To make writing the specification easier for the `Unresolved` case, you can write a weaker specification that does not need to be as precise as the definition. In particular, you can just ensure that when you return `Unresolved` the clause contains an unassigned literal. Note that this simplification could lead to an implementation that would mark a clause as `Unresolved` when it is already `Satisfied`. However, this would not be problematic for the correctness of `sat` since eventually the clause would be marked as `Satisfied`. This happens in practice since SAT solvers do not keep track of the status of a clause and only track if a clause is conflicting (requires backtracking) or unit (requires propagation).

Task 3 (10 pts). Implement, specify and verify a function `partial_eval_cnf` that takes a partial valuation p along with a CNF formula cnf as its arguments and returns:

- [`Sat`] if and only if p satisfies every clause of cnf. In this case, cnf 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 cnf. In this case, cnf is false for every valuation that is compatible with p and backtracking is needed.
- [`Other`] in every other case.

Your `partial_eval_cnf` function should raise an exception `Conflict_found` when a conflict is found. You do not need to find all conflicts and can return an exception in the first conflict you find. This corresponds to the following type and function definition:

```
exception Conflict_found

type cnf_status =
  | Sat
  | Conflict
  | Other

let partial_eval_cnf (p : pval) (cnf : cnf) : cnf_status
```

Similarly to Task 2, your specification only needs to prove implications and not equivalences.

Task 4 (5 pts). Implement, specify and verify a `backtrack` function. Recall that 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 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 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)
```

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.

Task 5 (5 pts). Implement, specify, and verify a function `set_value` that takes as its arguments an unassigned literal l and the current partial valuation p. It updates p by setting literal l to true. 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 (in this case since only one variable is assigned the delta will correspond to the variable `l.var`).

```
exception Sat_found

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

Note that `set_value` returns a `list var` but this list will only contain one element. However, we suggest this signature so that it will be easier to change your code from the checkpoint to the final submission. Similarly to the other tasks, you only need to prove implications in your contracts.

Task 6 (30). Implement, specify, and verify a function `sat` that uses partial valuations and puts all the previous pieces together to prove the satisfiability of a propositional formula. In particular, this function should satisfy the following contract.

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

Hints: Since this project is harder to fully verify, we provide here some hints that may be helpful for you.

When writing your specification about a formula being satisfiable, you will need to relate a partial valuation with a formula being satisfied. The following predicate (or something similar) may be useful for your tasks:

```
predicate sat_with_pval (pval : pval) (cnf : cnf) =
  forall rho:valuation. compatible pval rho -> sat_with rho cnf
```

When writing the specifications for the `partial_eval_cnf` function we **do not recommend** to take the definitions and transform them directly into predicates as below.

```
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
    ]}
...
```

Instead, you should write these predicates using the `sat_with` predicate (or similar). Note that the specification of `sat` relies on the predicate `sat_with`. If you write your other definitions without using this predicate then you would need to write many auxiliary lemmas to help the provers understand the connection between `sat_with` and those definitions.

1.2 SAT solver with unit propagation (Final Submission, 60 pts)

We now extend the previous implementation of the SAT solver with unit propagation. This will allow your solver to be more efficient since it can *backtrack* earlier because it may find conflicts earlier when propagating unit literals. All code that you write from this point forward should be in the module `UnitSat`. You can copy the previous functions before doing the modifications that are required below.

Task 7 (5 pts). To perform unit propagation, we need to capture the notion of *unit clause*. Modify and verify the function `partial_eval_clause` when considering an extension of the type `clause_status` that includes `Unit lit`, i.e. that returns:

- [`Unit` l] if c is a unit clause with unassigned literal l (for partial valuation p)

The updated type of `clause_status` is:

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

Task 8 (5 pts). Modify and verify the function `partial_eval_cnf` to consider unit clauses, i.e.:

- [`Unit_clause` l] only if cnf admits a unit clause whose unassigned literal is l. If cnf admits more than one unit clause, which one is featured in the argument of `Unit_clause` is unspecified.

Your `partial_eval_cnf` function should raise an exception `Unit_found` when a unit clause is found. You do not need to find all unit clauses and can return an exception in the first unit clause you find (even though there may be conflicting clauses in the formula). The updated type for `cnf_status` is:

```
exception Conflict_found
exception Unit_found lit

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

The Other case is not very interesting since it will not affect the correctness of the algorithm as long as you ensure that when the valuation is complete the result can only be either `Sat` or `Conflict`. Therefore, we allow you to weaken the specification of `Other` and write whatever you think it is suitable.

Task 9 (40 pts). Specify, implement and verify a function `set_and_propagate` with the the following signature:

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

This function 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. Even though your implementation must run this procedure until fix point, you do not need to write a specification that guarantees this fix point, i.e. your specification does not need to prove that when you terminate there are no more unit clauses. 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` $= \{x_0 \mapsto \texttt{true}\}$ updates `pval` to $\{x_0 \mapsto \texttt{true}, x_1 \mapsto \texttt{true}, x_3 \mapsto \texttt{false}, x_2 \mapsto \texttt{true}\}$ and returns the tuple $(\texttt{true}, [2, 3, 1])$.

Proving termination. In the template, you will find a lemma `numof_decreases` that may be useful for proving termination of the unit propagation procedure. This lemma states that when you modify an array by updating a single cell from a value v to a different value, the number of occurrences of v in this array decreases by one. To count the number of occurrences of v in an array, you can use the provided function `total_numof`.

```
function total_numof (t : array (option bool)) (v : option bool) : int =
  numof t v 0 (length t)
```

Because `numof` is defined by a set of axioms, `numof` and `total_numof` cannot be used in code and must only appear in annotations. Note that `total_numof` is only needed to prove the termination of the `set_and_propagate` function and it is not required for the checkpoint.

Task 10 (10 pts). Modify and verify the `sat` function to call `set_and_propagate` and the modified functions above. Note that the function `set_and_propagate` will replace the previous function `set_value` in your new implementation of your SAT solver.

The signature of `sat` should remain the same as before:

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

1.3 Writeup (Final Submission, 20 pts)

Task 11 (20 pts). Writeup, to be handed in separately as file `mp2-sol.pdf`.

2 Testing

Even though you will be verifying your `sat` function, writing a correct implementation can be challenging. Therefore, you may want to test that your function is producing the correct output (sat/unsat) for your implementation.

Testing the algorithm and making up CNF formulas can be tedious in Why3. We provide a test module with 10 formulas (5 satisfiable and 5 unsatisfiable).

You can execute the test module for the checkpoint (module Sat) as follows:

```
why3 -L . execute test.mlw --use="TestSat" 'all()'
```

A similar command can also be executed for the final submission (module UnitSat):

```
why3 -L . execute test.mlw --use="TestUnitSat" 'all()'
```

These commands print the number of "correct" answers. The default implementation in the template always returns unsatisfiable and if you run it you should get the following output:

```
result: int = 5
globals: <none>
```

After you implemented the `sat` function, you should **expect that number to be 10** if your implementation is correct:

```
result: int = 10
globals: <none>
```

Running all test cases can take a few seconds to run (e.g., the Sat version takes around 10 seconds on my local machine and the UnitSat around 4 seconds). Note that just because you can pass these 10 test cases does not mean the implementation is correct and that is why we want to verify the code so that we can guarantee that for any CNF formula our SAT solver will return the correct answer.