Mini-Project 2
Decision Procedures

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Tuesday, May 4, 2021 (checkpoint)
23:59pm, Thursday, May 13, 2021 (final)
250 pts

You should **pick one of the following two alternative mini-projects**. You may, but are not required to, do this assignment with a partner.

The mini-projects have two due dates:

- Checkpoint at 23:59pm, Tue May 4, 2021 (150 pts)
- Final projects at 23:59pm, Thu May 13 2021 (100 pts)

No late days may be used neither on the checkpoint nor on the final portion of the project. You may recover up to 60% of the points you lost at the checkpoint if you revise the first part with your final submission.

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/S21/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 '`function`', or '`predicate`', etc. You may also 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.

The Writeup (30 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 if we were going to reuse it again next year?

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 `unit-sat.mlw` (SAT option) and `cong-bare.mlw` and `cong-path.mlw` (Congruence Closure option). 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`.

  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.

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

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 12 notes.

1.1 SAT solver with partial valuations (Checkpoint:,150 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 ; polarity : bool }
type clause = list lit
type cnf = { clauses : array clause ; nvars : int }
type valuation = array bool
```

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) (cnf : cnf) : valuation
```

Task 2 (30 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
```

Task 3 (30 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 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
```

Task 4 (10 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 (10 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)
```

Task 6 (60). 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 }
```

1.2 SAT solver with unit propagation (Final Submission, 70 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. The updated type for `cnf_status` is:

```
exception Conflict_found
exception Unit_found lit

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

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. 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`, $[2, 3, 1]$).

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.

Task 10 (20 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, 30 pts)

Task 11 (30 pts). Writeup, to be handed in separately as file `mp2-sol.pdf`.

2 A Proof-Producing Decision Procedure

Decision procedures and theorem provers are complex software artifacts and, even with the best technology, verifying their correctness may be infeasible. Unsurprisingly, many implementations have been shown to have bugs which may erode confidence in their use during software verification. To restore this trust there is an interesting intermediate point: we can instrument a decision procedure to produce not just a *yes* or *no* answer, but also a *proof object* in case a given formula is valid. This proof object should be easy to check for correctness with a simple program, so we can trust at least a *yes* answer if we trust only the small checker. In general terms, this is an example of *result checking* [Blum & Kannan 1989]; in theorem proving it is closely intertwined with the origins of ML [Gordon et al. 1978].

In this mini-project, we explore a variation on this theme. We first implement a decision procedure for *congruence closure* and then instrument it with contracts that verify that it *can* produce a proof of any equality it affirms. As such, you prove a specific form of partial correctness.

2.1 Implementing Congruence Closure (Checkpoint, 150 pts)

You may want to review the description of *congruence closure* in Lecture 17 or other online information you find helpful. We will implement *incremental congruence closure* in which equations are asserted one by one and equality can be checked at any time. So at the high level we would have the following interface:

```
type eqn
type cc
let cc_new (n : int) : cc
let merge (cc : cc) (e : eqn) : unit
let check_eq (cc : cc) (e : eqn) : bool
```

where cc is the type of the data structure maintaining the congruence closure, and cc_new n creates a new data structure over constants $0, \ldots, n-1$ where each element is only equal to itself.

merge cc e updates cc to incorporate the equation e, and check_eq cc e returns *true* if the equation e follows from the equations asserted so far and the standard inference rules in the theory of equality with uninterpreted function symbols (namely: reflexivity, symmetry, transitivity, and monotonicity).

2.1.1 Representation of Terms

We reuse the implementation of union-find from Assignment 7. This means it is convenient to represent all constants as integers $0, \ldots, n-1$. For a maximally streamlined implementation we represent all terms in *Curried* form.

```
type const = int
type term = Const const | App term term
```

Here are some examples, using $a = 1$, $b = 2$, etc.

Term	Curried	WhyML
c	c	Const 3
$f(a)$	$(f\ a)$	(App (Const 6) (Const 1))
$f(g(a), b)$	$((f\ (g\ a))\ b)$	(App (App (Const 6) (App (Const 7) (Const 1))) (Const 2))

During congruence closure and other operations we need to consider equality between subterms of the input. In order to support this in a simple and efficient way we translate terms to so-called *flat terms* using new constants that act as names for the subterms. For example, the term $f(g(a), b)$ (or $((f\ (g\, a))\, b)$ in Curried form) might have the name c_3 with the definitions

$$\begin{aligned} c_1 &= g\, a \\ c_2 &= f\, c_1 \\ c_3 &= c_2\, b \end{aligned}$$

This representation means we only have to consider two kinds of equations in our algorithm, $c = (\mathsf{App}\, a\, b)$ for constants a and b and $a = b$.

```
  type const = int
  type eqn =
  | Defn const const const  (* c = App a b *)
  | Eqn const const         (* a = b *)
```

2.1.2 The Incremental Congruence Closure Algorithm

In order to accommodate the definitions above, we slightly modify the interface.

```
module CongBare

  use ...

  type const = int
  type eqn =
  | Defn const const const  (* c = app a b *)
  | Eqn const const         (* c = a *)

  use UnionFindBare as U

  type cc = { size : int ;
              uf : U.uf ;
              mutable eqns : list eqn }

  let cc_new (n : int) : cc
  let merge (cc : cc) (e : eqn) : unit
  let check_eq (cc : cc) (a : const) (b : const) : bool

end
```

Here, UnionFindBare is your bare implementation from Assignment 7. You may make minor modifications and extensions to its interface for the purposes of this mini-project.

The field cc.uf should be a union-find data structure over the constants $0 \leq c <$ cc.size and cc.eqns should be a list of the equations you need for the computation of your algorithm.

At a high level, merge $cc\, e$ should assert the equation e. This proceeds in two phases. In the first phase, we suitably update cc.uf and cc.eqns to join equivalence classes. In the second phase, we repeatedly *propagate* the equality to create a representation of the *congruence closure*.

The function check_eq $cc\, a\, b$ should just consult the union-find data structure to see if a and b are in the same equivalence class.

Your implementation does not need to be particularly efficient, but it should be polynomial. Furthermore, we constrain it to use union-find to maintain equivalence classes so that further standard improvements would be straightforward to make. Such further improvements are generally related to *indexing* to avoid searching through lists.

Your contracts should be sufficient for *safety* of all array accesses, but do not otherwise have to express correctness. Furthermore, you do not need to ensure termination.

As a consequence, you will need to test your implementation, and we will do so as well while grading. In order to facilitate our testing harness, you must adhere to the significant parts of the interface (namely, types const and eqn, and the types of the functions cc_new, merge, and check_eq). You may, however, modify or add fields to the cc structure, since testing will not rely on these internals.

Task 12 (130 pts). Implement and verify the safety the CongBare module as specified above.

Task 13 (20 pts). Test your implementation on several examples that exercise multiple aspects of the implementation, including the iterative nature of the propagation phase of the algorithm. You should have at least 5 distinct examples.

You should hand in file `cong-bare.mlw` with modules UnionFindBare, CongBare, and Test.

2.2 Instrumenting Congruence Closure (Final Submission, 45 pts)

For the final submission you will have to produce and verify the correctness of proofs of equality. We reuse here the abstract type of path in the union-find data structure, extended with two new constructors: hyp e and mono $p\,q\,e\,e'$ to represent hypotheses (assumptions) and the rule of monotonicity.

hyp (Eqn $a\,b$) is a path from a to b. This will be used if the client asserts an equation $a = b$ by calling merge cc (Eqn $a\,b$).

mono $p\,q$ (Defn $c\,a\,b$) (Defn $c'\,a'\,b'$) is a path from c to c', if p is a path from a to a' and q is a path from b to b'. This will be used if the algorithm uses monotonicity to conclude App $a\,b$ = App $a'\,b'$ from the equalities $a = a'$ and $b = b'$.

Note that any equation used as an argument to hyp and mono should be one directly passed into merge. This could be enforced in a complicated manner with an additional layer of abstraction, but we forego this complication since the client can still check separately that all uses of hyp and mono in a path rely only on equations it asserted.

```
module CongPath

  use ...

  type const = int

  type eqn =
  | Defn const const const   (* c = app a b *)
  | Eqn const const          (* c = a *)

  use UnionFindPath as U

  function hyp (e : eqn) : U.path
  axiom c_hyp : forall a b.
    U.connects (hyp (Eqn a b)) a b

  function mono (p : U.path) (q : U.path) (e : eqn) (e' : eqn) : U.path
  axiom c_mono : forall p q a a' b b' c c'.
    U.connects p a a' -> U.connects q b b' -> U.connects (mono p q (Defn c a b)
         (Defn c' a' b')) c c'

  type cc = { size : int ;
              uf : U.uf ;
              mutable eqns : list eqn }

  let cc_new (n : int) : cc
  let merge (cc : cc) (e : eqn) : unit
  let check_eq (cc : cc) (a : const) (b : const) : (bool, ghost (option U.path)
     )

end
```

We do not supply a path to merge since the merge function itself can construct it, as explained above.

For this instrumentation you may arbitrarily change your bare implementation, except that you should use your UnionFindPath from Assignment 7 (again, modified as needed).

Note that your contracts should guarantee two things: (1) safety (as before) and (2) the path provided with the result of check_eq $cc\,a\,b$ when a and b are in fact equal, must go from a to b.

Task 14 (45 pts). Add paths to serve as proof objects to your implementation as specified above.

2.3 Simplifying Paths (Final Submission, 25 pts)

The paths that represent proofs of equality constructed by union-find can be rather large. However, using "smart constructors" we can eliminate redundancies to create more compact paths. Obviously, this would only be relevant if we wanted to check paths for correctness externally, which is one way to use proof-producing decision procedures. In this task you will write some constructors to eliminate redundancies. We will not integrate this into our decision procedure, although it should be clear how this could be done.

For simplicity, in this task we are only concerned with with proofs of equalities (functions refl, sym, and trans) but *not* monotonicity (function mono). A key observation is that these constructors form a *group* over the equality assumptions hyp e for equations e. We have the following interpretation

Paths	Groups	
Hyp (Eqn a b)	x_{ab}	element
Refl a	1	unit element
Sym p	p^{-1}	inverse
Trans $p\,b\,q$	$p * q$	multiplication

with the laws

$$p * 1 = p = 1 * p$$
$$p * p^{-1} = 1 = p^{-1} * p$$
$$(p * q) * r = p * (q * r)$$

We would like to transform paths into a standard form where

1. Refl appears only at the top level, and

2. Sym p is allowed only for hypotheses p.

Task 15 (25 pts). Complete the following definitions

```
module Path

  type elem = int
  type eqn = Eqn elem elem
  type path =
  | Hyp eqn                 (* = x_ab *)
  | Refl elem               (* = 1 *)
  | Sym path                (* = p^-1 *)
  | Trans path elem path    (* = p * q *)

  predicate std (p : path)

  let refl a =
  ensures { std result }

  let trans p b q =
  requires { std p /\ std q }
  ensures { std result }

  let sym p =
  requires { std p }
  ensures { std result }

end
```

Your functions should also be such that refl corresponds to Refl, sym to Sym, and trans to Trans. For example, if trans $p\,b\,q$ where p is a path from some a to b and q is a path from b to some c, then the result should be a path from a to c. However, we won't bother verifying this, only that certain redundancies are avoided. We could then implement another pass that cancels $x_{ab} * x_{ab}^{-1}$ and $x_{ab}^{-1} * x_{ab}$ if we are interested in shortest possible paths.

You should hand in file `cong-path.mlw` with modules UnionFindPath, CongPath, and Path. Since `why3 execute` does not allow (most?) ghost expressions, you would have to use `why3 extract` and the OCaml compiler to test your instrumented code, which we do not require.

2.4 Writeup (Final Submission, 30 pts)

Task 16 (30 pts). Writeup, to be handed in separately as file `mp2-sol.pdf`.