MiniProject 2
Verification with Certificates

15-414: Bug Catching: Automated Program Verification

Due Friday, April 21, 2023 (checkpoint)
Friday, April 28, 2023 (final)

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

> WhyML implementations of the data structures below that have been verified in Why3 may exist online. While you can examine Why3 reference materials, tutorials, and examples, **you may not read or use Why3 implementations of the data structures we ask you to code**. However, you may study or use implementations in other languages (with appropriate citations), and you can freely use anything in the Why3 standard library. In addition, the Toccata gallery of verified Why3 program may provide some insight.

The mini-projects have two due dates:

- Checkpoint on Fri Apr 22 2022, 2022 (50 pts)
- Final projects on Fri Apr 29 2022 (100 pts)
  Up to 20 pts you lost on the checkpoint may be recovered on your final submission if you fix the problems that were noted. You are strongly encouraged to look at our feedback even if you received a full score.

The mini-projects 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/s22/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.

> Our main piece of advice is this: **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 functions **except for externally visible ones we use for testing purposes**. They are marked in the starter code as `DO NOT CHANGE`.

You can find starter code in the `sat-starter/` and `cong-starter/` directories.

The Writeup

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 3-4 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. Make sure you submit both the code and completed session folder in the zip. Feel free to adjust our past Makefiles for your purposes, but you are not required to create one.
- Submit a PDF containing your final writeup to MP2 Final (Written). There is no checkpoint for the written portion of the mini-project. You may use the file `mp2-final.tex` as a template and submit `mp2-final.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-final.tex` in the handout to get you started on this.

1 Certificate-Producing 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, takes a formula in conjunctive normal form as an input, and decides whether or not it is satisfiable. When the formula is satisfiable, the solver returns a satisfying assignment that it has verified correct. When the formula is not satisfiable, the solver returns a certificate that encodes a resolution proof that the original formula is unsatisfiable.

A Reminder on DPLL and Certificates

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. More details on the DPLL algorithm are available in the Lecture 12 notes.

$$F = \overbrace{(a \vee \neg b)}^{C_0} \wedge \overbrace{(\neg a \vee c \vee \neg d)}^{C_1} \wedge \overbrace{(a \vee c \vee \neg d)}^{C_2} \wedge \overbrace{(\neg c \vee \neg e)}^{C_3} \wedge \overbrace{(\neg c \vee e)}^{C_4} \wedge \overbrace{(c \vee d)}^{C_5}$$

Step	**Partial valuation**
Start with an empty partial valuation.	$\{\}$
Decide a.	$\{a \mapsto \texttt{true}\}$
Decide c.	$\{a \mapsto \texttt{true}, c \mapsto \texttt{true}\}$
Propagate $\neg e$ from unit clause C_3.	$\{a \mapsto \texttt{true}, c \mapsto \texttt{true}, e \mapsto \texttt{false}\}$
Clause C_4 is conflicting. Backtrack.	$\{a \mapsto \texttt{true}\}$
Decide $\neg c$.	$\{a \mapsto \texttt{true}, c \mapsto \texttt{false}\}$
Propagate d from C_5.	$\{a \mapsto \texttt{true}, c \mapsto \texttt{false}, d \mapsto \texttt{true}\}$
Clause C_1 is conflicting. Backtrack.	$\{\}$
Decide $\neg a$.	$\{a \mapsto \texttt{false}\}$
Propagate $\neg b$ from unit clause C_1.	$\{a \mapsto \texttt{false}, b \mapsto \texttt{false}\}$
Decide c.	$\{a \mapsto \texttt{false}, b \mapsto \texttt{false}, c \mapsto \texttt{true}\}$
Propagate $\neg e$ from unit clause C_3.	$\{a \mapsto \texttt{false}, b \mapsto \texttt{false}, c \mapsto \texttt{true}, e \mapsto \texttt{false}\}$
Clause C_4 is conflicting. Backtrack.	$\{a \mapsto \texttt{false}, b \mapsto \texttt{false}\}$
Decide $\neg c$.	$\{a \mapsto \texttt{false}, b \mapsto \texttt{false}, c \mapsto \texttt{false}\}$
Propagate d from unit clause C_5.	$\{a \mapsto \texttt{false}, b \mapsto \texttt{false}, c \mapsto \texttt{false}, d \mapsto \texttt{true}\}$
Clause C_2 is conflicting. Backtrack.	$\{\}$
Unsat	

Figure 1: Unit propagation in action

In the above example, the procedure returns unsat after first attempting to set a to `true`, encountering a conflict, then setting a to `false`, and encountering another conflict.

When DPLL returns that a formula is unsatisfiable, it can also produce a resolution proof that the formula is indeed satisfiable. Recall the resolution rule from Lecture 13. Below, C has no copy of p and D has no copy of $\neg p$.

$$\frac{p \vee C \quad \neg p \vee D}{C \vee D}\ \text{resolution}$$

We say that $C \vee D$ is the *resolvent* obtained by removing the complementary literals p and $\neg p$ from $p \vee C$ and $\neg p \vee D$, respectively. A resolution proof that ends with the empty clause, equivalent to $\bot$, is called a *refutation*, as it demonstrates the unsatisfiability of the formula.

A representative encoding of a refutation obtained by the DPLL trace above might look as as follows below on the left; the corresponding proof tree is on the right, with the complementary literal listed to the right of each step. In the encoding on the left, literals appearing in the formula are referred to by integers, with a corresponding to `0`, b to `1`, $\neg c$ to -2, and so forth. Each line begins with a unique identifier, and contains either an assumption or a resolution step. Lines with `assume` list a clause from the formula, e.g., `[-2 4]` for $\neg c \vee e$. Lines with `resolve` list the identifiers for two previous lines, followed by a variable, followed by the resolvent obtained by applying resolution to the clauses on the identified lines, on the complementary literals specified by the variable. So `resolve 0 1 4 [-2]` states that $\neg c$ is the resolvent obtained from steps 0 (`assume [-2 4]`) and 1 (`assume [-2 -4]`), on variable e (`4`).

```
0  assume  [-2 4]
1  assume  [-2 -4]
2  resolve 0 1 4 [-2]
3  assume  [2 3]
4  assume  [-0 2 -3]
5  resolve 3 4 3 [-0 2]
6  resolve 2 5 2 [-0]
7  assume  [0 2 -3]
8  resolve 7 3 3 [0 2]
9  resolve 2 8 2 [0]
10 resolve 6 9 0 []
```

$$\dfrac{\dfrac{\dfrac{\neg c \vee e \quad \neg c \vee \neg e}{\neg c}\, e \quad \dfrac{c \vee d \quad \neg a \vee c \vee \neg d}{\neg a \vee c}\, d}{\neg a}\, c \qquad \dfrac{\dfrac{\dfrac{\neg c \vee e \quad \neg c \vee \neg e}{\neg c}\, e \quad \dfrac{a \vee c \vee \neg d \quad c \vee d}{a \vee c}\, d}{a}\, c}{\bot}\, a$$

The procedure for obtaining this proof is as follows.

1. Whenever a conflict is encountered on clause C,
   - If the conflict occurred because literal l propagated from unit clause C', return the resolvent of C and C' on l.
   - Otherwise, return C.
2. Whenever both decisions l and $\neg l$ result in conflicts, apply resolution to the clauses obtained from Step 1, on complementary literals l and $\neg l$.
3. When applying resolution in Steps 1 and 2, if one clause does not involve the literal l, then skip resolution and return that clause instead.

1.1 "Baby" SAT (Checkpoint, 50 points)

In Assignment 5, you specified and implemented some simple operations that can be performed over formulas in CNF.

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

We will pick up where we left off, making use of the same types and functionality that you implemented before.

Certificate checking

However, in its current form this solver does not produce certificates, and will always raise an exception when it encounters an unsatisfiable formula. But in order to produce a resolution certificate, it needs (at least) a definition of the type `certificate`, and more importantly a definition for `valid_refutation` and a procedure to ensure that a given certificate meets this definition.

Task 1 (5 points). Define a type `certificate` that will encode resolution proofs of unsatisfiability. You may choose whatever representation you like, i.e. proofs can be trees with leaves corresponding to clauses from the formula, or flat sequences with fields that identify antecedents at other positions in the sequence. Before proceeding, you should think carefully about which encoding to use, and consider looking ahead to future tasks to inform your choice. Note that you may define other types that are referenced by `certificate`, e.g., if you want to define a type for each individual step in a resolution proof.

Task 2 (15 points). Define a predicate `valid_refutation` over a certificate `p` and formula `cnf`. It should the set of `certificate` objects which correspond to correct resolution proofs of unsatisfiability. In other words, the predicate should be true if and only if:

- Each step encoded in a `certificate` is either a correct application of the resolution rule, or an assumption of a clause appearing in the original CNF formula.
- The final step of the proof encoded in the certificate is the empty clause.

As in the previous task, you may write several "helper" predicates or functions that are used to define `valid_refutation`, and are encouraged to do so to make your specification more readable.

Task 3 (30 points). Implement the function `check_refutation`, which accepts a certificate `p` and CNF formula `cnf`, and returns true if and only if the certificate is a valid refutation for the formula. If you need to change the signature to make it recursive, that is fine, but you should provide a variant to ensure that this procedure terminates. As before, you are encouraged to implement helper functions that accomplish much of the work needed by `check_refutation`, to make your code easier to understand.

```
let check_refutation (p : certificate) (cnf : cnf) : bool
  requires { true } (* Any preconditions that you find necessary *)
  ensures { result <-> valid_refutation p }
```

As you complete this task, we encourage you to make use of the helper functions provided in the starter code. There are functions for performing basic tests on clauses and literals that may prove useful when implementing your certificate checker.

The only verification that you are required to perform is the contract given in the listing of `check_refutation` above (note that `diverges` is not part of its specification, so you should prove that it terminates). You may need to write other contracts and invariants that, for example, ensure that array accesses are in bounds.

What not to change. To allow us to check your work effectively, do not change the signature of `sat`, or any of the type definitions listed at the beginning of this section: `var`, `lit`, `clause`, `cnf`, and `valuation`.

1.2 DPLL (Final submission, 80 points)

Below is a basic SAT solver that builds on the types we have defined, using exhaustive search to find a satisfying assignment. According to its specification, if it does not find a satisfying assignment, then it should return a certificate that amounts to a valid refutation for the original formula.

```
let sat (cnf : cnf) : solver_result =
  ensures { forall t. result = Assignment t -> sat_with t cnf }
  ensures { forall p. result = Proof p -> valid_refutation p cnf }
  raises  { InvalidCert }
  let t = make cnf.nvars false in
  let rec split i =
    (* Invariants in solver.mlw *)
    if i = cnf.nvars then
      if eval_cnf t cnf then raise Sat
    else begin
      t[i] <- false ; split (i + 1) ;
      t[i] <- true  ; split (i + 1) ;
    end
  in try split 0 ; raise InvalidCert with Sat -> Assignment t end
```

Note that the return type for this solver is `solver_result`:

```
type solver_result =
  | Assignment valuation
  | Proof certificate
```

Most importantly, although this implementation satisfies its contract, it will *always* raise `InvalidCert` when it encounters un unsatisfiable result. Your goal will be to adapt this basic solver in the following ways:

- Use *partial* assignments so that it is not necessary to assign all variables before checking for conflicts.
- Apply unit propagation immediately after splitting on a variable, to avoid wasting time on unnecessary case splits.

- Construct resolution certificates using your types from the checkpoint, and call `check_refutation` before returning a proof.

In all of the tasks below, your primary concern will be implementing correct functionality. You do not need to provide complete specifications to facilitate a full verification of your code, beyond what is necessary to ultimately ensure that the top-level postconditions on `sat` can be verified. You can annotate any function with `diverges` to avoid needing to prove termination.

Note: It is advisable to read through all of the tasks before starting. The following tasks are listed in an order determined by dependency, i.e., the functionality in Task 7 depends on the functionality described in Tasks 4, 5, and 6. However, you may benefit from first writing prototypes and contracts for some of the functions from earlier tasks, and attempting to implement Tasks 7 and 8 first, and your approach to implementing the solver (Task 8) may change the way that you do unit propagation (Task 7), which may in turn impact choices that you make for earlier functions.

Essential Primitives

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 4 (15 pts). Implement 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 and function definition:

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

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

Task 5 (15 pts). Implement 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.
- [`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.
- [`Other`] in every other case.

Your `partial_eval_cnf` function should raise an exception `Conflict_found` when a conflict is found. Note that you do not need to find *all* of the conflicting clauses—it is fine to return an exception with the first conflict found. Likewise, it should raise `Unit_found` when a unit clause is found, and may return the first unit clause found. This corresponds to the following type and function definition:

```
exception Conflict_found
exception Unit_found lit

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

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

Managing Solver State

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 is 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.

Task 6 (5 pts). Implement a `backtrack` function that unassigns the variables on a given list in a given partial valuation:

```
let rec backtrack (diff : list var) (pval : pval)
```

You are not required to use this signature, and if you find it useful to add additional arguments, you are free to do so.

The core of DPLL is unit propagation, which you should implement in a function `set_sign` that changes a variable in a partial valuation from `None` to `Some b`, for a value `b`, and propagates any resulting unit literals until there are no further opportunities to do so. Because the information needed to construct a resoultion certificate is generated during unit propagation, it is important to think carefully about how you will implement this functionality. You are free to write the signature of `set_sign` however you find appropriate, but one suggestion is to use:

```
exception Sat_found
let rec set_sign
  (l : lit) (pval : pval) (cnf : cnf) : (bool, option clause, list var, list (
      option (clause, var)))
```

With this signature, `set_sign` behaves as follows.

- It raises a `Sat_found` exception in case the CNF becomes satisfied.
- If it reaches a conflict, then the first component of its return value is `true`, and its second component is `Some c`, where `c` is either:
  - The clause on which it encountered a conflict, if there was a conflict found before any unit propagation was performed.
  - If the conflict was encountered in clause C_1 after unit-propagating literal l from unit clause C_2, and l is in C_2, then the resolvent of C_1 and C_2 on literal l.
- If it does not reach a conflict, and the CNF is not satisfied after exhausting unit propagation, then the first component is `false`, and the second component is `None`.
- Its third component is a list of all of the variables that it assigned in `pval` prior to returning, which the solver can use for backtracking.
- The last component, of type `list (option (clause, var))`, is a list of the unit clauses and corresponding literals encountered during propagation.

The benefit of returning this information from each variable assignment and propagation is, of course, that it can be used to construct a resolution certificate. Whenever a conflict is detected, then the clause returned by `set_sign` can be used along with the list of unit clauses and literals (returned in the fourth component) to generate a sequence of steps to add to the certificate.

Task 7 (20 pts). Implement a function `set_sign` that implements the functionality described above.

Updating the Solver

Task 8 (25 pts). Make necessary changes to the `sat` function listed at the beginning of this section to make use of partial valuations, putting all the previous pieces together to either prove the satisfiability of a propositional formula, or provide a resolution certificate in cases of unsatisfiability. The updated function should satisfy the contract that it is given in the starter code (and shown in the handout), but you do not need to prove termination. You should change the type of `solver_result` to present partial valuations rather than complete assignments.

```
type solver_result =
  | Assignment pval
  | Proof certificate
```

What not to change. To allow us to check your work effectively, do not change the signature of `sat`, or any of the type definitions listed at the beginning of this section or the previous: `var`, `lit`, `clause`, `cnf`, `valuation`, `clause_status`, `cnf_status`, and `pval`.

1.3 Writeup (Final Submission, 20 pts)

Task 9 (20 pts). Writeup, to be handed in separately as file `mp2-final.pdf`.

2 Congruence Closure

At the core of decision procedures or theorem provers for a variety of theories are algorithms to compute the *congruence closure* of some equations including uninterpreted function symbols. Even more fundamentally, congruence closure itself relies on computing and maintaining *equivalence classes* of terms. An efficient data structure for this purpose is called *union-find*. You may read, for example, the Wikipedia article on Disjoint-Set Data Structure. Union-find also has other applications, such as in Kruskal's algorithm for minimum spanning trees.

For the checkpoint, you will implement union-find and partially prove it correct and also produce checkable certificates. For the final submission you will use your union-find algorithm to implement congruence closure, which will also produce a certificate.

2.1 Bare Union-Find (Checkpoint, 25 pts)

All *elements* that are to be divided into equivalence classes are represented as integers $0 \leq x <$ size. In a separate data structure maintained by a client, these could be mapped, for example, to terms.

Throughout the algorithm, each equivalence class maintains a unique *representative element* which we visualize as the root of a tree. In addition, each element has a *parent*, with the representative of a class functioning as its own parent. We call such representatives *roots*.

To determine if two elements x and y are in the same equivalence class we ascend the tree to find the representative of the classes for x and y, say, $\hat{x} = \mathsf{find}\, x$ and $\hat{y} = \mathsf{find}\, y$. If $\hat{x} = \hat{y}$ then x and y are in the same class; otherwise they are not.

Initially, all elements are in their own (singleton) equivalence class and we call union to merge equivalence classes. The operation $\mathsf{union}\, x\, y$ should merge the equivalence classes for x and y. We do this by calculating the representatives $\hat{x} = \mathsf{find}\, x$ and $\hat{y} = \mathsf{find}\, y$. If these are equal we are done. Otherwise, we set the parent of $\hat{x}$ to be $\hat{y}$ or the parent of $\hat{y}$ to be $\hat{x}$.

To decide between these two alternatives we maintain a *rank* for each root z that is a bound on the longest chain of parent pointers for the tree below z. We set the parent of $\hat{x}$ to $\hat{y}$ if $\hat{x}$ has strictly smaller rank than $\hat{y}$ and vice versa. If the ranks are equal, the choice is arbitrary, and we also have to increase the rank of the resulting root by one.

Task 1 (25 pts). Implement the bare union-find data structure with the following types:

```
type elem = int

type uf = { size : int ;
            parent : array elem ;
            rank : array int }
```

Here, $\mathsf{parent}[x]$ is the parent of element x, and x itself for the root. $\mathsf{rank}[x]$ is the rank of x (only relevant if x is a root). Implement the following predicates and functions, following the informal description and other sources as you see fit.

```
predicate is_root (uf : uf) (x : elem)
let uf_new (n : int) : uf
let find (uf : uf) (x : elem) : elem
let union (uf : uf) (x : elem) (y : elem) : unit
```

- is_root $uf\ x$ is true iff x is a root in uf.

- uf_new $n = uf$ returns a new union-find structure over elements $0 \leq x < n$, with each element a root.
- find $uf\ x = \hat{x}$ returns the root $\hat{x}$ representing the equivalence class containing x.
- union $uf\ x\ y$ modifies uf by merging the classes containing x and y.

Your contracts should be strong enough to verify that all array accesses are in bounds and that the result of find is a root. **You do not need to verify termination (use `diverges` instead) or any other correctness properties of your functions.**

Because the contracts essentially only specify *safety* and not correctness, it is your responsibility to make sure your code properly implements the union-find data structure. You do **not need to implement the so-called *path compression*** during the find operation (which further improves the already excellent bound of $n \log(n)$ for n successive union-find operations). We recommend writing test cases, but we do not require them as part of your submission.

2.2 Producing Certificates (Checkpoint, 25 pts)

In many practical scenarios where decision procedures or theorem provers are used, it is impractical to formally prove their correctness. That is unfortunate, as we want to be able to rely on the results. To close this gap, we can extend the algorithm so it produces a certificate, or even verify that it *could* produce a certificate when it gives a positive answer.

Applying this to union-find means we would like to instrument the code so that it can produce a *certificate showing* that any element is equivalent to the representative of the equivalence class it is in. We call a certificate that x and y belong to the same equivalence class a *path from x to y*. We have the following constructors for paths, derived from the axioms for equivalence relations:

- refl x is a path from x to x.
- sym p is a path from y to x if p is a path from x to y.
- trans $p\ y\ q$ is a path from x to z if p is a path from x to y and q is a path from y to z.

Whenever union $x\ y$ is called, the client of the data structure must provide a path from x to y which somehow justifies the equivalence. For example, if $x = a + 1$ and $y = 1 + a$, the client might provide a path explaining that x and y are equivalent due to the commutativity of addition. The implementation of union-find takes these on faith (they are the client's responsibility, after all) but can apply refl, sym, and trans to build longer paths from those that are given.

We keep the type of path abstract so that the implementation of union-find cannot "fake" any paths. The properties listed above are summarized using the axioms below.

```
type path  (* abstract *)
function refl (x : elem) : path
function sym (p : path) : path
function trans (p1 : path) (x : elem) (p2 : path) : path

predicate connects (p : path) (x : elem) (y : elem)
axiom c_refl : forall x. connects (refl x) x x
axiom c_sym : forall p x y. connects p x y -> connects (sym p) y x
axiom c_trans : forall x y z p q.
  connects p x y -> connects q y z -> connects (trans p y q) x z
```

The union-find data structure now maintains a ghost array path of paths, where for every element x, path$[x]$ is a path connecting x to parent$[x]$. **This property should be guaranteed by the data structure invariants.** The information is sufficient to produce a path from x to the representative $\hat{x}$ of its equivalence class.

Task 2 (25 pts). We update the interface as follows:

```
type uf = { size : int ;
            parent : array elem ;
            rank : array int ;
            ghost path : array path }

let uf_new (n : int) : uf
let find (uf : uf) (x : elem) : (elem, ghost path)
let union (uf : uf) (x : elem) (y : elem) (ghost pxy : path) : unit
```

with the specifications

- find $uf\ x = (\hat{x}, p)$ should ensure that p is a path from x to $\hat{x}$. This path should be constructed while traversing the data structure. **Your postcondition should enforce that p is indeed a path from x to $\hat{x}$.**
- union $uf\ x\ y\ p$ requires that p is a path from x to y. This means the client has to supply the evidence for the equality x and y. Since union modifies uf by merging the classes of x and y, it will need to update the path field to maintain the data structure invariants.

Your code should include sufficient data structure invariants and contracts to guarantee these properties for find and union. **Your contracts still do not need to express, for example, that** union **really represents a union. It therefore remains your responsibility that the code is correct.**

2.3 Implementing Congruence Closure (Final Submission, 40 pts)

You may want to review the description of *congruence closure* in Lecture 18 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).

2.3.1 Representation of Terms

It is convenient to represent all constants as integers $0, \ldots, n-1$, as in the implementation of union-find. 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 *)
```

2.3.2 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 the checkpoint. You may make minor modifications and extensions to its interface for the purposes of the final submission.

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 3 (40 pts). Implement and verify the safety the CongBare module as specified above.

We recommend you test your implementation but we do not formally require it. You should hand in file `cong-bare.mlw` with modules UnionFindBare and CongBare. The code quoted in this handout is in the `cong-starter/` directory.

2.4 Instrumenting Congruence Closure (Final Submission, 40 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.

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 4 (40 pts). Add paths to serve as certificates to your bare implementation as specified above.

We recommend you test your implementation but we do not formally require it. You should hand in file `cong-path.mlw` with modules UnionFindPath and CongPath. The code quoted in this handout is in the `cong-starter/` directory.

2.5 Writeup (Final Submission, 20 pts)

Task 5 (20 pts). Writeup, to be handed in separately as file `mp2-final.pdf`.