Mini-Project 2
Decision Procedures

15-414: Bug Catching: Automated Program Verification

Due Tuesday, April 8, 2025 (checkpoint)
Friday, April 18, 2025 (final)
150 pts

You should **pick one of the following two alternative mini-projects**. You are encouraged 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. The Toccata gallery of verified Why3 program may provide some insight.

The mini-projects have two due dates:

- Checkpoint on Tue Apr 8 2025
- Final projects on Fri Apr 18 2025
  Up to 50% of the points 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 site.

If you are working with a partner, only one of the two of you should 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.

A note on verification and grading. The verification component of this project is more challenging than previous assignments. Whether your solution fully verifies is just one component of our grading, and we will consider the student's effort in verifying their code. If you have spent time to strengthening your specifications, testing your code to ensure it is correct, and trying different approaches to specifying and proving correctness, but still were not able to verify everything, you will receive a significant portion of the available points for verification.

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`.

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). **Importantly**, if parts of your code did not verify and you spent time trying to fix it, explain what you tried, why you think it may not have worked, and why you believe that your solution is correct.
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 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 Building blocks of DPLL (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 (15 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
- [`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
```

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 (15 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.
- [`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. You do not need to find all conflicts and can return an exception in the first conflict you find. Likewise, it should raise `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). 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
```

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

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 (20 pts). Implement 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.

- 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 $(\texttt{true}, [2, 3, 1])$.

For the checkpoint, you do not need to write contracts for `set_and_propagate`, or verify it. For the final solution, you will need to write an appropriate specification and prove it when you implement the SAT solver. However, you should start thinking about what the specification should look like. In particular, you should think about how to specify the relationship between the returned delta and the partial valuation that is passed as an argument. If you want to provide a conjectured specification as comments, we may be able to provide feedback on it that could help you in the final submission.

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

Now your task is to use the building blocks developed for the checkpoint to produce a working implementation of DPLL. This will allow your solver to be more efficient since it can *backtrack* earlier because it may find conflicts earlier when propagating unit literals. Continue working in the module `Sat`. Before continuing, you should review the notes for Lecture 16 to remind yourself how these pieces can fit together in a working implementation of DPLL.

Task 6 (60 pts). Implement, specify, and verify a function `sat` that uses partial valuations and calls `set_and_propagate` and `backtrack`, putting 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 }
```

Note that you are not required to prove termination. If you do not, you can annotate the above function, and any of its dependencies, with `diverges`.

Rubric. The 60 points available for this task will be broken down as follows:

- 20 pts for the implementation of `sat` and any auxiliary functions apart from those from the checkpoint (`set_and_propagate`, `backtrack`).
- 20 pts for the specification of auxiliary functions needed by `sat` (including `set_and_propagate`, `backtrack`, and any others you devise).
- 20 pts for the verification of `sat` and any auxiliary functions needed by `sat` (including `set_and_propagate`, `backtrack`, and any others you devise).

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.

1.3 Writeup (Final Submission, 20 pts)

Task 7 (20 pts). Writeup, to be handed in separately as file `mp2-sol.pdf`.

Testing your implementation

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 20 formulas in `test-sat.mlw`.

You can execute the test module as follows:

```
why3 -L . execute test-sat.mlw --use="TestSat" '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>
```

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.

Background: Union Find In the starter code `cong-bare.mlw`, you will find an implementation of the *bare* union-find data structure. All *elements* that are 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 $\mathsf{union}\, x\, y$ accomplishes this by calculating the representatives $\hat{x} = \mathsf{find}\, x$ and $\hat{y} = \mathsf{find}\, y$. If these are equal, we are done. Otherwise, it sets 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.

The implementation is provided in the starter code as follows:

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

2.1 Implementing Congruence Closure (Checkpoint, 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).

2.1.1 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 *)
```

2.1.2 The Incremental Congruence Closure Algorithm

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

```
module CongBare

  use ...
```

```
4
5   type const = int
6   type eqn =
7   | Defn const const const   (* c = app a b *)
8   | Eqn const const          (* c = a *)
9
10  use UnionFindBare as U
11
12  type cc = { size : int ;
13              uf : U.uf ;
14              mutable eqns : list eqn }
15
16  let cc_new (n : int) : cc
17  let merge (cc : cc) (e : eqn) : unit
18  let check_eq (cc : cc) (a : const) (b : const) : bool
19
20 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 1 (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.

2.2 Producing Certificates (Checkpoint, 20 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$[x]$. 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
```

For the checkpoint, you do not need to write contracts or verify them, but you should aim for the following functionality (which you will need to verify for the final submission):

- find $uf\ x = (\hat{x}, p)$ should construct a path from x to $\hat{x}$ while traversing the data structure. You want to make sure that p is indeed a correct path from x to $\hat{x}$, so that you will be able to verify it for the final submission.
- union $uf\ x\ y\ p$ may assume 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.

2.3 Specify and Verify Union-Find (Final Submission, 20 pts)

For the final submission you will update your Congruence Closure algorithm to produce and verify the correctness of proofs of equality. To lay the groundwork for this, you will need to specify and verify key properties of your implementation of union-find with paths.

Task 3 (20 pts). Update `UnionFindPath` to ensure that the following properties hold:

- A data structure invariant for `uf` which ensures that for every element x, path$[x]$ is a path connecting x to parent$[x]$.
- 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 $uf\ x\ y\ p$ can formally require that p is a path from x to y. In terms of postconditions, there are no specifically required properties that you must prove to receive credit, but any postconditions you find useful should be verified, and any verification conditions arising from data structure invariants must also be verified.

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.** You do not need to prove termination, but where ever your code accesses arrays, it should prove that the accesses are safe.

2.4 Instrumenting Congruence Closure (Final Submission, 50 pts)

In order for the Congruence Closure algorithm to provide proof certificates, we reuse 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.

```
1  module CongPath
2
3    use ...
4
5    type const = int
6
7    type eqn =
8    | Defn const const const   (* c = app a b *)
9    | Eqn const const          (* c = a *)
10
```

```
11  use UnionFindPath as U
12
13  function hyp (e : eqn) : U.path
14  axiom c_hyp : forall a b.
15    U.connects (hyp (Eqn a b)) a b
16
17  function mono (p : U.path) (q : U.path) (e : eqn) (e' : eqn) : U.path
18  axiom c_mono : forall p q a a' b b' c c'.
19    U.connects p a a' -> U.connects q b b' ->
20    U.connects (mono p q (Defn c a b) (Defn c' a' b')) c c'
21
22  type cc = { size : int ;
23              uf : U.uf ;
24              mutable eqns : list eqn }
25
26  let cc_new (n : int) : cc
27  let merge (cc : cc) (e : eqn) : unit
28  let check_eq (cc : cc) (a : const) (b : const) : (bool, ghost (option U.path)
        )
29
30 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.

2.5 Writeup (Final Submission, 20 pts)

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