Mini-Project 2
Verification with Certificates

15-414: Bug Catching: Automated Program Verification

Due Friday, Nov 14, 2025 (checkpoint)
Tuesday, Nov 25, 2025 (final)
150 pts

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 Nov 14 2022, 2025 (50 pts)
- Final projects on Tue Nov 25 2025 (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 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.

1.1 Bare Union-Find (Checkpoint, 25 pts)

All *elements* that are to be divided into equivalence classes are represented as integers $0 \leq x < \mathsf{size}$. 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.

1.2 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.**

1.3 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).

1.3.1 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 *)
```

1.3.2 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.

1.4 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.

1.5 Writeup (Final Submission, 20 pts)

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

1.6 Testing

We provide harnesses and tests for both the checkpoint (union–find only) and the final (congruence closure). Before proceeding:

1. Ensure the included `why3d` script is executable and runnable from a shell.
2. Install a recent OCaml toolchain (e.g., via ocaml.org).

Running the tests

From `cong-starter`:

```
# Checkpoint: union{find only
make test-uf

# Final: congruence closure (bare)
make test-bare

# Final: congruence closure with paths (instrumented)
make test-path
```

Test case format

All identifiers are integers in the range `0 ... size-1`. A case file has a header, an `--ops--` section of assertions, a `--checks--` section of queries, and ends with `--end--`.

Union Find tests. Ten starter test cases are provided in `tests-uf`.

```
name: uf_single_union
size: 4

--ops--
union 0 1                # merge classes of 0 and 1

--checks--
check 0 1 true           # ? are in same class
check 2 3 false
--end--
```

Congruence-closure tests (final). Ten starter test cases are provided in `tests-cong`. Two kinds of assertions appear under `--ops--`:

- `eq a b` asserts the equality $a = b$.
- `def c a b` records the application $c = f_a(b)$ of an uninterpreted unary function f_a to argument b. Congruence means that equal heads and equal arguments imply equal results.

Checks may be written as `? a b = true` or `check a b true`. Example:

```
name: congruence-both-equal
size: 7

--ops--
def 2 0 3                # 2 = f_0(3)
def 4 1 5                # 4 = f_1(5)
eq  0 1                  # 0 = 1
eq  3 5                  # 3 = 5

--checks--
```

```
check 2 4 true            # by congruence
check 0 3 false           # unrelated constants remain distinct
--end--
```

Adding your own tests

You can add files with extension `.tcase` to either test case directory; they will be automatically detected by the makefile. You are encouraged to think of new test cases as you work through the project. Feel free to share your test cases with the rest of the class on Piazza as well!

2 Nested Depth-First Search

At the core of LTL model checking is the search for an *accepting lasso* in a Büchi automaton: a finite *prefix* from an initial state to some accepting state u, followed by a non-empty *loop* (from u back to a state on the prefix, typically u itself). The language of a Büchi automaton is non-empty iff such a lasso exists. A standard way to decide non-emptiness is the *Nested Depth-First Search* algorithm. It runs a DFS from every initial state (*outer* DFS). Whenever the outer DFS discovers an accepting state u, it launches a second DFS (*inner* DFS) to search for a path from u back to any state on the current outer-DFS stack. If such a path exists, we have found an accepting cycle, and thus a witness lasso.

We will implement nested DFS over an explicit Büchi automaton and *produce a certificate* for positive answers. Your implementation will return a trace t that forms a valid accepting lasso prefix when a counterexample is found; we will require a postcondition on your top-level checker that expresses precisely this fact.

In LTL model checking one typically constructs the product of a system with the Büchi automaton for the negation of the property and then checks the product for an accepting lasso; a lasso corresponds to a counterexample execution. In this mini-project we work directly with a Büchi automaton, focusing on the nested DFS emptiness check and its certificate. Refer to lecture 18 for a review of Büchi automata and the nested DFS algorithm.

2.1 Büchi Automaton Data Structure (Checkpoint, 10 pts)

To give you flexibility, we do not fix the internal representation of Büchi automata. It is your job to specify it and the associated invariants that guarantee safety. The testing harness will only rely on a small, fixed interface to construct automata, using `create` and `add_transition`.

Task 1 (10 pts). Define a module `Buchi` that exports the following **externally visible** items.

```
module Buchi
  use int.Int
  use list.List

  type state = int          (* DO NOT CHANGE *)
  type trace = list int     (* DO NOT CHANGE *)

  type buchi                (* your representation; abstract to clients *)
                            (* define any invariants needed for safety *)

  (* Build an automaton with given number of states, initial states, and
     accepting states. *)
  let create (n : int) (initial : list state) (final : list state) : buchi

  (* Add a directed transition u -> v. *)
  let add_transition : (b : buchi) (u v : state) : unit
end
```

Note that we fix the type `trace` to be `list int`, so that our test harness can interface with your code. However, you may define an "internal" trace type that is different, e.g. as an array, if you find it convenient. Remember that you will need to translate between the `list int` traces in the two locations that our test harness will call your code if you do.

You are responsible for the data structure invariants that guarantee all array accesses are in bounds and that only valid states appear in the initial, accepting, and successor sets. You may add further functions and predicates as you see fit, but the harness will only call `create`, `add_transition`, and later pass the constructed `buchi` to the nested DFS implementation.

Beyond what is necessary to verify safety for the data structure and functions for this task, we do not require any additional contracts or proof of correctness. You are free to provide them if you wish, but any contract in your code must verify to receive full credit.

Before committing to a particular type, you should consider the pros and cons of using mutable versus immutable data types for its components, and any metadata (or ghost data) that you may want to include. It may help to read through to the end of this project description before completing this task.

2.2 Accepting Traces (Specification) (Checkpoint, 20 pts)

We will certify positive answers by returning a finite trace that matches a lasso shape. To express what it means for such a trace to be valid for a given automaton, define a logical predicate `accepting_trace` and any supporting predicates you find helpful.

Task 2 (20 pts). In the `Buchi` module, define the predicate `accepting_trace`.

```
predicate accepting_trace (g: buchi) (t: trace)
```

Your specification should model acceptance via a finite prefix that forms a *lasso*. Concretely, the trace can be decomposed into an initial `prefix` and subsequent `loop` where (1) `prefix` is a valid trace whose first state lies in the set of initial states; (2) `loop` is a non-empty valid trace with a transition from the final state of `prefix` to the first state of `loop`; and (3) `loop` contains an accepting state, and begins and ends with the same state.

Your predicate may be computational; if so, you are free to change the definition to use `let` as follows.

```
let predicate accepting_trace (g: buchi) (t: trace)
```

Whether it makes sense to define it computationally will depend on the choices you made in your `buchi` type. But keep in mind that if you do make it computational, then you can use it as the basis for Task 3 by calling it directly.

We strongly encourage you define supporting helper predicates and functions as needed, rather than trying to define everything in one complex predicate. We do not require any specific set of helpers, but you might consider some of the following:

- a predicate that says whether a state `v` is a successor of state `u`
- a predicate that says a finite sequence `t` is a valid trace in `g`
- transitive reachability from some initial state to a given state
- a predicate that determines whether a given trace is a cycle
- a predicate that says whether a given trace `t` is a lasso with prefix `pref` and loop `loop`

Only implement what you need to ultimately provide the logical spec `accepting_trace`, striving for elegance and readability.

2.3 Accepting Traces (Computation) (Checkpoint, 20 pts)

In addition to providing a logical specification of trace acceptance, you will implement and verify correct a *computational* acceptance checker for a given finite trace.

Task 3 (15 pts). Implement `check_accepting` within the `Buchi` module. Do not change the interface, as part of your grade will depend on the outcome of automated test cases.

```
(* DO NOT CHANGE *)
let check_accepting (b : buchi) (t : trace) : bool
  ensures { result <-> accepting_trace g t }
```

That is, if `validate_trace g t` returns `true`, then `t` is a valid accepting witness according to your logical predicate. As discussed in Task 2, if your specification of `accepting_trace` is already computational, then you may call it from `check_accepting`. Whether this is convenient depends on your design choices so far.

You should prove that `check_accepting` satisfies the given postcondition, and that it terminates. You are free to add additional preconditions as needed for safety, as any verification condition resulting from your implementation must verify in order to recieve full credit.

2.4 Inner DFS and Cycle Detection (Final Submission, 35 pts)

With the Büchi data structure and formalized notion of trace acceptance in place, we move to implementing the nested DFS search itself.

The algorithm is composed of an inner and outer depth-first search. When the outer search encounters a state from the accepting set, it starts the inner search and attempts to reach a state that currently appears on the outer stack. Reaching such a state t closes a cycle against the path recorded by the outer search and is the key to building a finite counterexample trace whose loop is non-empty and contains an accepting state.

Task 4 (35 pts). Implement `dfs2` as described, and any helper functions and data types needed to support it. You do **not** need to prove functional correctness for `dfs2` or any of its helpers. However, you **must prove safety and termination**. Name your outer-search function `dfs2`; the harness will not call it directly, so its parameters and return type are your choice.

On success, return enough information for the outer search to reconstruct the loop (for example, a concrete path $u \rightsquigarrow t$ together with the identity of t); otherwise, report failure. You are free to introduce helper data types, functions, and predicates to keep the bookkeeping clear. Name your inner-search function `dfs2`; its exact signature is up to you and will not be called by the harness directly.

There are numerous ways to implement the inner search of nested DFS. We encourage you to read sources online, and references from the lecture notes, to familiarize yourself with several of them. Two important variations are:

- Rather than reporting a cycle when the state's successor is on the outer stack, the inner DFS takes an additional `seed` argument to record the state from which the inner search was launched. Cycle detection is postponed until the search returns exactly to `seed`. This makes the soundness of the inner search straightforward, and in practice may expose your implementation to fewer potential bugs at a modest price in efficiency.

- Rather than explicitly maintaining an inner and outer stack, maintain a *color* for each state. Blue states have been fully visited by the outer search, red states have been visited by the inner search, and cyan states are on the current path of the outer search. The logic for determining when to recurse and detect cycles is adjusted accordingly in both inner and outer searches.

Each of these alternatives comes with a different way of maintaining the state of the search, and tradeoffs in implementation complexity and efficiency. You are free to choose whichever approach you believe you can implement correctly and elegantly.

2.5 Outer DFS and Counterexample Assembly (Final Submission, 35 pts)

The outer search starts exploring from a given initial state, searching for reachable accepting states from which to launch the inner search. Before launching the inner search, it recursively calls outer search on all successors to explore the reachable state space, in order to prevent calling the inner search redundantly.

Task 5 (35 pts). Implement the outer depth-first search in a function called `dfs1` with the behavior described above. You do **not** need to prove functional correctness for `dfs1` or any of its helpers. However, you **must prove safety and termination**. Name your outer-search function `dfs1`; the harness will not call it directly, so its parameters and return type are your choice.

If the inner search succeeds, the outer search should return what is needed to assemble an accepting trace t composed of a prefix that leads from an initial state to u and a non-empty loop that closes back on the outer stack, traversing. an accepting state along the way. Ultimately, you will want to check that this trace satisfies `accepting_trace`, so that you can verify that your top-level solver returns correct results.

As with the inner search, there are several ways to implement `dfs1`. To a large extent, the approach that you take will be determined by the approach you take for the inner search, as the nested searches need to share information. Although we are not grading on the efficiency of your algorithm, we do consider it an incorrect implementation of the nested DFS algorithm if the outer search eagerly invokes the inner search *before* fully exploring its successors recursively. Regardless of which approach you take to implementing the search, you should ensure that

We advise that you review the fuel-based termination arguments discussed in Lecture 14, and look for ways to use it in this application.

2.6 Top-Level Checker and Contract (Final Submission, 10 pts)

The checker coordinates the search and packages the result for the harness. It should iterate the outer search from all initial states and either return a counterexample trace that your specification deems accepting, or report that no accepting cycle was found.

Task 6 (10 pts). Provide a module that exports the following externally visible result type and checker:

```
type result = Trace trace | NonAccepting          (* DO NOT CHANGE *)

let check_buchi (g : buchi) : result              (* DO NOT CHANGE *)
  ensures { match result with
            | Trace t -> accepting_trace g t
```

```
6               | _         -> true
7               end }
```

Your `check_buchi` should have the effect of launching `dfs1` from each initial state in turn and return `Trace t` if an accepting trace is found, or `NonAccepting` otherwise.

Do not modify the contract above:

- If an accepting trace is returned, then you must be able to prove that it satisfies `accepting_trace`.
- In the case that no accepting trace is found, no postcondition is required.

A separate termination proof for `check_buchi` is not required if all recursion is delegated to your already-terminating searches, but its safety obligations must verify. Regardless, you should not mark `check_buchi diverges`.

2.7 Writeup (Final Submission, 20 pts)

Task 7 (20 pts). Writeup, to be handed in separately as file `mp2-final.pdf`. In addition to the general guidelines at the beginning of the handout, please include:

- your chosen representation of Büchi automata and why it was convenient;
- a brief description of your witness format and checker;
- any invariants or helper lemmas that were critical to proving the postcondition on `check_buchi`.

2.8 Testing

We have provided ten test cases for the checkpoint and final submission to get you started testing your solution. Before proceeding:

1. Make sure that the `why3d` file in the handout is marked executable, and that you have access to a bash shell or similar.
2. If you haven't already, install OCaml locally on your system.

Install graph-easy (optional)

The test harness will print a visual display of each Buchi automaton it tests if you have `graph-easy` installed on your system. If you don't, the harness will still run, and just not print the graphical representations. If you would like to install `graph-easy`, follow the instructions below. Note that your system configuration may be different than the course staff, and our ability to get `graph-easy` installed on your system may be limited. Because it is optional, we may ask you to proceed without it if installation proves difficult.

macOS

```
# Install cpanminus, then graph-easy
brew install cpanminus
cpanm App::GraphEasy
graph-easy --version
# If you use MacPorts instead:
sudo port install p5-graph-easy
```

Linux

```
# Debian/Ubuntu
sudo apt-get update
sudo apt-get install -y libgraph-easy-perl

# Fedora
sudo dnf install -y perl-Graph-Easy

# Arch Linux (via CPAN)
sudo pacman -S perl-cpanminus
cpanm App::GraphEasy
```

Windows Install Strawberry Perl (https://strawberryperl.com/), then in a `cmd` or PowerShell window:

```
cpan App::GraphEasy
# or, with cpanminus installed
cpanm App::GraphEasy
```

Running the tests

Two harnesses are provided:

- Accepting trace checker (checkpoint)
- Nested DFS (final)

In the starter directory `F25/assignments/mp2/nested-dfs-starter`:

```
# Run all accepting-trace tests
make test-acc-all

# Run all nested-DFS tests
make test-all
```

Test case format

Each test case file has a header of key–value pairs, followed by a graph description between `--graph--` and `--end--`. Nodes are written as `[i]` and edges as `[u] -> [v]`. Accepting states are shown in the diagram by adding a double border attribute.

Accepting-trace tests Starter directory: `tests-accepting`.

```
name: chain_to_selfloop_accept
initial: 0
accept: 2
trace: 0 1 2 2 2      # the concrete trace to check
expect: true

--graph--
[2] { border: double }
[0] -> [1]
[1] -> [2]
[2] -> [2]
--end--
```

Nested DFS tests Starter directory: `tests-nested-dfs`. (In the full solution directory, the nested-DFS tests live in `tests`.)

```
name: accept-loop
initial: 0 2          # one or more initial states
accept: 1 3           # accepting states
expect: true          # whether an accepting lasso is expected

--graph--
[0]
[1] { border: double }   # accepting states use double border for the diagram
[0] -> [1]
[1] -> [1]
--end--
```

Adding your own tests

You can add files with extension `.tcase` to `tests-nested-dfs` and `tests-accepting`; the harnesses will discover all test cases in those directories. You are encouraged to think of new test cases as you work through the project. Feel free to share your test cases with the rest of the class on Piazza as well!