Assignment 7
Certificates

15-414: Bug Catching: Automated Program Verification

Due Thursday, April 14, 2022
80 pts

This assignment is due on the above date and it 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.

What To Hand In

You should hand in the following files on Gradescope:

- Submit the file `asst7.zip` to Assignment 7 (Code). You can generate this file by running `make handin`. This will include your solutions `bfs.mlw`.
- Submit a PDF containing your answers to the written questions to Assignment 7 (Written). You may use the file `asst7.tex` as a template and submit `asst7.pdf`.

Compiling and running

Detailed instructions for compiling and running your implementation on Linux, macOS, Docker, and WSL are contained in `readme.md`. This file also contains basic guidelines to get you started writing tests cases for your solution.

Using LaTeX

We prefer the answer to your written questions 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 `asst7.tex` with handout to get you started on this.

1 Uninterpreted Functions

Task 1 (5 pts). **Program equivalence**. The following programs are equivalent, in that they will terminate with the same values in `e` and `f` when started in states with the same values for `a`, `b`, `c`, and `d`.

```
1
2
3 e := (a + b) * (c + d)
```

```
1 g1 := a + b;
2 g2 := c + d;
3 f := g1 * g2
```

Write a formula in the theory of equality and uninterpreted functions that demonstrates the equivalence of these programs. That is, write a formula that is *unsatisfiable* if and only if these programs assign the same values to `e` and `f` when executed on the same values of `a`, `b`, `c`, and `d`.

Task 2 (10 pts). Demonstrate the unsatisfiability of your solution to Task 1 by running the congruence closure algorithm on it. Your solution should show the steps of the algorithm with a level of detail comparable to what is demonstrated in Examples 7 and 8 of Lecture 18.

Task 3 (5 pts). Unfortunately, it is not always possible to prove that two programs are equivalent using uninterpreted functions. Write a program that is equivalent to the first (i.e., produces the same value in `f` as `e := (a + b) * (c + d)`), but for which your approach for Task 1 produces a *satisfiable* formula. We advise that you check your work by applying the congruence closure algorithm to extract a satisfying assignment, but this is not required to receive credit.

2 Checkable Certificates

Today's SAT solvers can be very complicated, with many high-level and low-level optimizations. As such, they cannot be proved correct in practice. Instead, they produce *checkable certificates* that allow a client to verify the *correctness of the outcome of the computation*. The certificate checker should be small, trustworthy, and ideally proved correct even if the solver itself can not be proved. The ultimate outcome of a computation would then be **sat** or **unsat** or **error** which means that the certificate did not pass the checker, indicating a bug in the SAT solver. In this way we can restore our trust in the answer of the SAT solver even if it is quite complicated.

In this problem we explore checkable certificates for the problem of finding the shortest path from a source to a sink in a directed graph. To keep it as simple as possible, we assume all distances between vertices are 1. Under these assumptions, breadth-first search through the graph will find the shortest distance. We implement breadth-first search by using a queue containing the current frontier of the search.

Dijkstra's algorithm for shortest paths (where not all edges have unit length) *can* in fact be verified, but requires induction for some of the steps. See http://toccata.lri.fr/gallery/dijkstra.en.html.

2.1 The Interface Layer

We make the representation of vertices explicit as integers so you (and we) can easily test the implementation. The top-level interface function is

```
use fmap.MapAppInt as Map

type vertex = int
type graph = Map.t (list vertex)
type path = list vertex

type outcome = NoPath | SomePath path | Error

let shortest_path (vs:list vertex) (g:graph) (v:vertex) (u:vertex) : outcome
```

with the following interpretation:

vs	the list of vertices in the graph g
g	the graph g as a map from vertices to a list of adjacent vertices
v	the starting vertex of the potential path
u	the ending vertex of the potential path

The $outcome$ has the following straightforward interpretation

NoPath	there is no path from v to u
SomePath us	a path from v to u as a list of vertices, not including v
Error	the certificate could not be verified

Note that the path us must be a shortest path (which is not necessarily unique). The empty path Nil represents the zero-length path from v to itself, so then $v = u$.

2.2 The Inner Layer

The algorithm will take a graph and a starting vertex v and return a $visited$ map that records for each reachable vertex u a pair (pu, du) where pu is predecessor of u on a shortest path from v to u and du is the distance from v.

```
type visited = Map.t (vertex , dist)

let bfs (g:graph) (v:vertex) : visited
```

The function bfs should maintain a queue representing the frontier of the exploration of reachable vertices, starting with the single node v at distance 0. You may consult various sources for such an algorithm, but note that since all edges have distance 1 you do not need a priority queue as you do for Dijkstra's algorithm, just a simple queue. The starter code supplies the implementation from Lecture 3, augmented so it can also function as a stack (see below).

All the functions you write may be marked as **diverges** and you do not need to prove their correctness. You may nevertheless need some pre- and post-conditions for your functions to make sure that result checking in the outer layer can run correctly.

2.3 The Outer Layer

In the outer layer we can easily compute the outcome from the $visited$ map computed by breadth-first search. Recall:

```
type path = list vertex
type outcome = NoPath | SomePath path | Error

let bfs (g:graph) (v:vertex) : visited
let shortest_path (vs:list vertex) (g:graph) (v:vertex) (u:vertex) : outcome
```

Then, if u is not in the domain of *visited* then there is no path from v to u. If *visited* maps u to (pu, du) then we can follow the predecessors all the way back to v, assembling a path in the process.

2.4 Certificate Checking

We use the *visited* map to check correctness of the answer as follows:

1. If u is not in the domain of *visited* we should check that the *visited* map is *saturated*, that is, when u has been visited, then every neighbor of u has also been visited. This should be checked straightforwardly by going through all vertices in the graph or the domain of *visited*.

2. If u is in the domain of *visited*, we should check three properties:

   (a) The path computed by tracing back through predecessors is indeed a valid path in the graph.

   (b) The *visited* map should record minimal distances from v for every reachable node u. We check this by considering every a and b in the visited map with an edge from a to b. It must be the case the $db \leq da + 1$, that is, there could be a shorter path to b than through a but not a longer one. This could underestimate the distance (if there is a bug in the code and a shorter path does not actually exist), so we also need to check that the computed path from v to u has the length predicted by du.

2.5 Testing

We need to test the inner layer (algorithm) as well as the outer layer (certificate checker). This means there needs to be a possibility of introducing a bug into your implementation. We achieve this via the function *insert*:

```
use Queue as Q

let ref depth_first = false

let insert (x:'a) (q:Q.queue 'a) : Q.queue 'a =
if depth_first then Q.push x q else Q.enq x q
```

You should use this function to insert a vertex and its distance into the frontier. When *depth_first* is **false** then your algorithm should behave like breadth-first search and therefore obtain the shortest path. When *depth_first* is **true** then your algorithm should behave like depth-first search and therefore does not always compute the shortest path.

In order to test the implementation you should include a module BfsTest with at least 5 test cases *test1* through *test5*, at least one of which should use depth-first search and return an incorrect path that is then rejected by the result checker.

The autograder will test your code on a number of examples, both with breadth-first and depth-first search, expecting the certificate checker to fail on some depth-first examples so that the overall outcome will be the value Error. If the outcome is incorrect, the test function should raise the **exception** Incorrect.

2.6 Contracts and Verification

Since this is an exercise is certificate checking, you should ideally verify that the certificate checkers are correct (if not the core search algorithm). However, since even correctness of the checker requires induction, and termination is also not entirely straightforward, you are relieved of the responsibility to verify them. Instead, we rely on the simplicity of the certificate checker.

You may still need or want to have contracts to guarantee that operations such as looking up a vertex in a map are well-defined. If you have such contracts, they need to be verified and the proof should properly replay.

2.7 Grading

Your code, including the test cases, should verify in Why3 and replay should work. You only need contracts to the extent

We intend to distribute points as follows:

1. **[20 pts]** A correct implementation breadth-first search (function bfs)
2. **[20 pts]** A correct implementation of the certificate checker (function $shortest\_path$)
3. **[10 pts]** Test cases (functions $test1, \ldots, test5$)
4. **[10 pts]** Verification of contracts to the extent they are present