Assignment 7
Finding Unions

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Thursday, April 22, 2021
90 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/assignments.html.

What To Hand In

You should hand in the following files on Gradescope:

- Submit the file `asst7.zip` to Assignment 7. You can generate this file by running `make handin`. This will include your files `missingP1.smt2`, `missingP2.smt2`, and `missingP12.smt2`. This will also include your solutions `unionfind.mlw`, and the proof sessions in `unionfind/`.

  Make sure your session directories and your solution files are up to date before you create the handin file.

1 Program Equivalence (40 pts)

Consider the two programs C programs `missingP1` and `missingP2`. Assume that the array "`a`" has size n and contains unique integers ranging from 1 to $n+1$. One of the integers is missing in this array and the programs `missingP1` and `missingP2` find the missing element with different strategies. For instance, if the array is initialized with elements $\{1, 3, 4\}$, then both `missingP1` and `missingP2` will return the missing value 2. Show that these programs are equivalent for arrays of size 3 (with integers ranging from 1 to 4) by encoding these programs to SMT using static single assignment (SSA) and unrolling the loops. You should submit the `.smt2` files `missingP1.smt2` (Task 1), `missingP2.smt2` (Task 2), and `missingP12.smt2` (Task 3). Note that if you want to write comments in `.smt2` files, you can do this by using the symbol "`;`".

```c
int missingP1(int a[], int n)
{
    int i, total;
    total = (n + 1) * (n + 2) / 2;
    for (i = 0; i < n; i++)
        total -= a[i];
    return total;
}

int missingP2(int a[], int n)
{
    int i;
    int x1 = a[0];
    int x2 = 1;

    for (i = 1; i < n; i++)
        x1 = x1 ^ a[i]; // xor

    for (i = 2; i <= n + 1; i++)
        x2 = x2 ^ i; // xor

    return (x1 ^ x2); // xor
}
```

Task 1 (15). Write a SMT formula in SMT-LIB format that symbolically encodes program `misssingP1`. We describe here the main aspects of the SMT-LIB language that you will need to use. For this task, you should use the theory of integer linear arithmetic to represent integers. The SMT-LIB format accepts the usual operations over integers, i.e. $=, <, >, <=, >=, *, -, +, /$. You can use negation `(not Bool Bool)` to model not equal or the negation of any Boolean operator. For instance, `(assert (not (= (x y))))`, would assert that x must be different from y.

The main operations over arrays are `select` and `store`. The operator `select` is relevant for this task and takes as input an array and an index and returns the value of the array at that index. For this task, you can initialize the first 3 elements of an unbounded array as depicted below. Complete the rest of the formula by encoding the program to SMT. Do not forget the preconditions of the problem that restrict the contents of the array.

```
(declare-fun p1n () Int)
(assert (= p1n 3))
(declare-fun p1ret () Int)
(declare-fun p1e1 () Int)
(declare-fun p1e2 () Int)
```

```
(declare-fun p1e3 () Int)
(declare-fun p1a () (Array Int Int))
(assert (= (select p1a 0) p1e1))
(assert (= (select p1a 1) p1e2))
(assert (= (select p1a 2) p1e3))
...
```

Variable `p1ret` should represent the return value of program `missingP1`. Note that after having the SMT formula, you can run `z3` on the formula and see that the solver will assign integer values to variables `p1e1`, `p1e2`, and `p1e3` that will correspond to the values in the array. The variable `p1ret` will be assigned the missing integer value. You should already have `z3` installed in your system since we use it in `Why3`. To run `z3`, simply run the command:

```
$ z3 missingP1.smt2
```

Task 2 (15). Write an SMT formula in SMT-LIB format that symbolically encodes program `missingP2`. For this task, you should use the theory of bitvectors to represent integers with bitvectors of bit-width 8. The list of operations over bitvectors that may be useful for this task are:

- binary predicate for unsigned less than or equal: `(bvule (_ BitVec m) (_ BitVec m) Bool)`
- binary predicate for unsigned greater than or equal: `(bvuge (_ BitVec m) (_ BitVec m) Bool)`
- addition modulo 2^m: `(bvadd (_ BitVec m) (_ BitVec m) (_ BitVec m))`
- multiplication modulo 2^m: `(bvmul (_ BitVec m) (_ BitVec m) (_ BitVec m))`
- unsigned division, truncating towards 0: `(bvudiv (_ BitVec m) (_ BitVec m) (_ BitVec m))`
- bitwise exclusive or: `(bvxor (_ BitVec m) (_ BitVec m) (_ BitVec m))`
- creates a bitvector from an integer: `(_ int2bv 8) m`

The full list of operators for bitvectors is available online. The array can be initialized in a similar way as in the previous task. Complete the rest of the formula by encoding the program to SMT. Do not forget the preconditions of the problem that restrict the contents of the array.

```
(declare-fun p2n () (_ BitVec 8))
(assert (= p2n ((_ int2bv 8) 3)))
(declare-fun p2ret () (_ BitVec 8))
(declare-fun p2e1 () (_ BitVec 8))
(declare-fun p2e2 () (_ BitVec 8))
(declare-fun p2e3 () (_ BitVec 8))
(declare-fun p2a () (Array Int (_ BitVec 8)))
(assert (= (select p2a 0) p2e1))
(assert (= (select p2a 1) p2e2))
(assert (= (select p2a 2) p2e3))
...
```

We also include the syntax for using `int2bv` to get the bitvector representation of an integer for the value of 3. We suggest using different variable names than the previous formula since that will be useful when writing the formula for Task 3. Variable `p2ret` should represent the return value of program `missingP2`. As before, you can run `z3` over this formula by using the command:

```
$ z3 missingP2.smt2
```

`z3` will assign integer values to variables `p2e1`, `p2e2`, and `p2e3` that will correspond to the values in the array. As before, the variable `p2ret` will be assigned the missing integer value.

Task 3 (10). Use the previous formulas to write a SMT formula called `missingP12.smt2` that encodes the equivalence between the two programs `missingP1` and `missingP2`, i.e. for the same inputs they will have the same output. You can run `z3` over this formula using the command:

```
$ z3 missingP12.smt2
```

2 Union-Find (50 pts)

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.

In this problem you will implement union-find and partially prove it correct. We explain below exactly which properties you are asked to prove.

2.1 The Bare Algorithm

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 4 (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, parent$[x]$ is the parent of element x, and x itself for the root. 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. However, 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).

The code for this task should be in a module `UnionFindBare` in the file `unionfind.mlw`.

2.2 Producing Proofs

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 write the prover so it produces a representation of a *proof* every time it is run, or even verify that it *could* produce a proof when it gives a positive answer and remain silent when it does not.

Applying this to union-find means we would like to instrument the code so that it can produce a *proof* that any element is equivalent to the representative of the equivalence class it is in. We call a proof 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 5 (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. Since this function 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.**

The code for this task should be in a module `UnionFindPath` in the file `unionfind.mlw`. We suggest you cut-and-paste some of your code for Task 4 as a starting point.

As an example of the use of paths, consider a client who wishes to check if x and y are in the same class and obtain a path as explicit evidence if they are. The client could write the following function:

```
let eq_path (uf : uf) (x : elem) (y : elem) : ghost (option path) =
requires { 0 <= x < uf.size /\ 0 <= y < uf.size }
diverges
returns { | Some p -> connects p x y
          | None -> true }
  match (find uf x, find uf y) with
    ((z, p), (z', q)) -> if z = z' then Some (trans p z (sym q)) else None
  end
```

This function obtains no information if x and y do not have the same representative. Conceivably, the union-find implementation could construct a model showing that x and y may be different, but that is beyond the scope of this assignment.

Task 6 (10 pts extra credit). Update your path-producing implementation of Task 5 of find so it performs path compression, as in the standard union-find algorithm. This must update not only parent$[y]$ but consequently also path$[y]$ for each element y between x and $\hat{x}$ in the tree.