Homework Assignment 3

Due: March 1, 2017 at 11:00pm

Total points: 100

Deliverables: hw3.pdf containing typeset solutions to Problems 1–6.
sort.dfy containing your Dafny implementation for Problem 2.
hw3.smt containing your SMT-Lib encoding for Problem 5.

1 Reasoning about Programs with Hoare Logic (30 points)

1. (30 points) Prove the validity of the following Hoare triple:

```
{$n \geq 0 \land d > 0$}
q := 0;
r := n;
while (r ≥ d) {
  q := q + 1;
  r := r - d;
}
{$n = q * d + r \land 0 \leq r < d$}
```

Your answer should take the form of a *proof outline*, which annotates the program S with FOL predicates inferred by applying the rules of Hoare logic, as seen in Lecture 9. For example, if a proof outline includes n consecutive predicates $F_1, \ldots, F_n$, then it must be that case that $F_1 \Rightarrow \ldots \Rightarrow F_n$, corresponding to the Rule of Conseqeunce. Similarly, each statement $s \in S$ must be surrounded by formulas P and Q such that $\{P\}S\{Q\}$ is a valid Hoare triple, according to the inference rule for S.

2 Verifying Programs with Dafny (30 points)

In this part of the assignment, you will use Dafny (Lecture 11) to verify a modified implementation of the insertion sort. You can either download and install Dafny or use the web interface at rise4fun. To get started, read the Dafny Guide, which describes all features of Dafny that are needed to complete the assignment.

2. (30 points) sort.dfy contains an implementation of insertion sort and a partial correctness predicate: applying the `sort` method to an array a ensures that $a[i] \leq a[j]$ for all indices $i < j$. This predicate is not quite right as written, however, and the implementation is missing all annotations except for the desired post-condition on `sort`.

   Get Dafny to verify sort.dfy by annotating it with sufficient pre/post conditions, assertions, loop invariants, and frame conditions. You may not change the implementation in any way other than by adding annotations. When the verification succeeds, Dafny will print the following message: "Dafny program verifier finished with n verified, 0 errors" (where n is a small number). Submit your annotated copy of sort.dfy.

3 Symbolic Execution (40 points)

Consider the Python programs `P0` and `P1` shown below, along with a harness procedure that tests their equivalence on a given k-bit integer:

```
def P0(x, k):
  return x & -x

def P1(x, k):
  for i in range(0, k):
    if (x & (1 << i)) != 0:
      return 1 << i
  return 0

def equiv1(x, k):
  assert P0(x, k) == P1(x, k)
```

Suppose that we apply symbolic execution to evaluate `equiv1` on a symbolic k-bit integer x and a concrete positive integer k. Let $VC_1(x, k)$ denote the set of verification conditions emitted during this symbolic execution process.

3. (2 points) How many verification conditions will be generated?

4. (3 points) Can the generated verification conditions $VC_1(x, k)$ be used to *prove* that `P1` and `P0` are equivalent for a given concrete k? Explain why or why not.

5. (20 points) Encode all verification conditions from $VC_1(x, 4)$ in SMT-LIB syntax. Your encoding should list the verification conditions *in the order in which they are generated* by basic symbolic execution (i.e., depth-first, exploring 'then' branches before 'else' branches). Each verification condition should be defined using its own **define-fun** expression. All verification conditions should be checked independently for validity using the **push** / **pop** commands (see the SMT-LIB manual or the Z3 tutorial for details). In particular, your encoding should take the following form:

   ```
   (declare-const x ...)
   ...
   (define-fun vc0 ...) ; the first VC generated by symbolic execution
   ...
   (define-fun vcn ...) ; the last VC generated by symbolic execution

   (push) ; check the validity of vc0
   ...
   (check-sat)
   (pop)
   ...
   (push) ; check the validity of vcn
   ...
   (check-sat)
   (pop)
   ```

   Use Z3 to check the validity of your $VC_1(x, 4)$ encoding. Report the result of running Z3 with the `-st` and `-smt2` options. Submit your SMT-LIB encoding in a separate `hw3.smt` file.

6. (15 points) Consider the program `P2` shown below:

   ```
   def P2(x, k):
     i = 0
     while ((x & (1 << i)) == 0 and i < k):
       i = i + 1
     assert P0(x, k) == x & (1 << i)
   ```

The **`assert`** statement checks if `P2`'s final state is equivalent to that of `P0`. Use the technique shown in Lecture 13 to transform and annotate `P2` so that it can be used to prove the equivalence of `P2` and `P0` by emitting just three verification conditions via symbolic execution. In particular, after your transformation, symbolic execution should behave as follows when applied to `P2(x, k)` with a symbolic k-bit integer x and a concrete positive integer k:

- it yields a set $VC_2(x, k)$ with three verification conditions, whose sizes are independent of k (though quantifiers with domains involving k are permitted), and
- `P2` and `P0` are equivalent if and only if the verification conditions in $VC_2(x, k)$ are valid.

Your transformed code may use the procedure `symbolic(k)` to obtain a fresh symbolic k-bit integer; it may include **`assume`** statements; and it may also use Python's **`all`** syntax in **`assert`** and **`assume`** statements as a universal quantifier (e.g., `all(x[i] for i in range(0,k))` is equivalent to $\forall\, i \in [0, k).\ x[i]$).

```
# annotated and transformed code
def P2(x, k):
  ...
```