Homework Assignment 2

Due: February 15, 2017 at 11:00pm

Total points: 100

Deliverables: `hw2.pdf` containing typeset solutions to Problems 1-13.
`tree.als` containing your Alloy encoding for Problems 7-10.
`verifier.rkt` containing your implementation for Problem 12.

1 Theory of Equality and Uninterpreted Functions (15 points)

1. (5 points) Apply the congruence closure algorithm to decide the satisfiability of the following $T_=$ formula:

$$f(g(x)) = g(f(x)) \wedge f(g(f(y))) = x \wedge f(y) = x \wedge g(f(x)) \neq x$$

Provide the level of detail as in Lecture 5. In particular, show the intermediate partitions (sets of congruence classes) after each merger or propagation step, together with a brief explanation of how the algorithm arrived at that partition (e.g., "according to the literal $f(x) = y$, merge $f(x)$ with y").

2. Consider the following program fragments, where all variables are 32-bit integers:

P_1:

```
return (x1 + y1) * (x2 + y2)
```

P_2:

```
u1 = (x1 + y1)
u2 = (x2 + y2)
return (u1 * u2)
```

(a) (5 points) Use Bounded Model Checking (BMC) to construct a formula in the theory of equality ($T_=$) that is unsatisfiable iff P_1 and P_2 are equivalent ignoring the semantics of 32-bit addition and multiplication. Use variables `r1` and `r2` to stand for the return values of P_1 and P_2, respectively.

(b) (5 points) Construct a program P_3 such that P_3 is equivalent to P_1, but the equivalence of P_1 and P_3 cannot be proven without considering some aspect of the semantics of 32-bit addition or multiplication. In particular, P_3 should be constructed by modifying **exactly one expression** in P_2. The BMC formula for checking the equivalence of P_1 and P_3 must be satisfiable in $T_=$ but unsatisfiable in the theory of bitvectors (T_{bv}); provide a brief explanation of why this is true for your P_3.

2 Combining Theories with Nelson-Oppen (35 points)

3. Consider the following formula in $T_= \cup T_R$:

$$g(x + y, z) = f(g(x, y)) \wedge x + z = y \wedge z \geq 0 \wedge x \geq y \wedge g(x, x) = z \wedge f(z) \neq g(2x, 0)$$

   (a) (5 points) Purify the formula and show the resulting $T_=$ and T_R formulas. Show the purification results using the table below. Apply purification to the (current) innermost term first. If there are several innermost terms, prefer the leftmost one. Use a_i to refer to the i^{th} auxiliary literal, starting with a_1. All occurrences of the same term should be mapped to the same auxiliary literal. You do not need to show the individual steps of the purification process, just the final result.

$T_=$	T_R
...	...

   (b) (5 points) Use the Nelson-Oppen procedure to decide the satisfiability of the purified formula. In one sentence, state which version of the procedure you are using and justify your choice. Show the equality propagation by filling out the table below. If T_i infers the j^{th} equality (or disjunction of equalities), enter it into the j^{th} row and i^{th} column only—leave the remaining column in that row empty.

$T_=$	T_R
...	...

4. (5 points) Recall that the theory of arrays $T_A = \{\mathit{read}, \mathit{write}, =\}$ is defined by the following axioms.

$$\begin{array}{l} \forall a, i, j.\ i = j \rightarrow \mathit{read}(a, i) = \mathit{read}(a, j) \\ \forall a, v, i, j.\ i = j \rightarrow \mathit{read}(\mathit{write}(a, i, v), j) = v \\ \forall a, v, i, j.\ i \neq j \rightarrow \mathit{read}(\mathit{write}(a, i, v), j) = \mathit{read}(a, j) \end{array}$$

   Prove that T_A is not convex by constructing $n \geq 3$ formulas in T_A such that $F_1 \Rightarrow (F_2 \vee \ldots \vee F_n)$ but $F_1 \not\Rightarrow F_i$ for any $i \in [2 \ldots n]$.

5. (10 points) Prove that the theory of equality $T_=$ is convex.

6. (10 points) Let F be a conjunctive formula in a non-convex theory T. Let G be a finite disjunction of equalities $\bigvee_{i=1}^{n} u_i = v_i$, also in T, such that $F \Rightarrow G$. Describe an algorithm for computing a minimal disjunction G' of the equalities in G such that $F \Rightarrow G'$. If your algorithm returns a minimal disjunction with m equalities, then it should have invoked the decision procedure for T at most $O(m \log n)$ times.

3 Finite Model Finding with Alloy (20 points)

In this part of the assignment, you will write four short Alloy specifications and check their correctness with the help of Alloy's finite model finder (Lecture 8). To start, download alloy.jar and double click on it to launch the tool. You may also want to skim Parts 1 and 2 of the Alloy tutorial.

The following questions ask you to formally define different kinds of tree data structures. We will only consider trees that have directed edges and no unconnected nodes. Such a tree is fully described by its set of edges. In Alloy, we model the edges of a tree (or, more generally, a graph) as a binary relation from nodes to nodes.

A skeleton solution can be found in tree.als. Complete the missing definitions and submit your copy of tree.als. Solutions will be automatically checked against a reference specification, so they need to be fully contained in the submitted file.

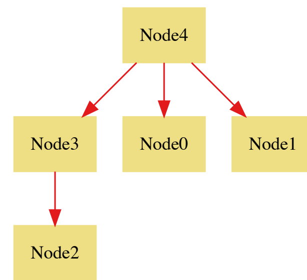

Figure 1: A tree is a binary relation between nodes.

7. (5 points) A *tree* is a graph that satisfies additional properties. What are those properties? Formalize them by completing the definition of the `tree` predicate in tree.als. Use the Alloy tool to check that your definition is correct (i.e., it rejects relations that are not trees) and non-vacuous (i.e., it admits some relations) in a universe with a small number of nodes.

8. (5 points) Formalize the properties of a *spanning tree* (of a directed graph) by completing the definition of the `spanningTree` predicate in tree.als. Check your definition for correctness and vacuity errors.

9. (5 points) Define *binary trees* in terms of their *left* and *right* relations, which map tree nodes to their left and right children (if any), respectively. Use your definition to complete the `binaryTree` predicate in tree.als. Check your definition for correctness and vacuity errors.

10. (5 points) Define *binary search trees* in terms of their *left*, *right*, and *key* relations. As above, the *left* and *right* relations map tree nodes to their left and right children (if any). The *key* relation maps tree nodes to integer keys. Use your definition to complete the `binarySearchTree` predicate in tree.als. Check your definition for correctness and vacuity errors.

4 A Verifier for Superoptimization (30 points)

Superoptimization is the task of replacing a given loop-free sequence of instructions with an equivalent sequence that is better according to some metric (e.g., shorter). Modern superoptimizers work by employing various forms of the guess-and-check strategy: given a sequence s of instructions, they guess a better replacement sequence r, and then they check that s and r are equivalent. In this problem, you will develop a simple SMT-based verifier for superoptimization. Given two loop-free sequences of 32-bit integer instructions, your verifier will either confirm that they are equivalent or, if they are not, it will produce a concrete counterexample—an input on which the two sequences produce different outputs.

The verifier will accept programs in the `BV` language, which has the following grammar:

```
Prog        :=  (define-fragment (id id*) Stmt* Ret)
Stmt        :=  (define id Expr) | (set! id Expr)
Ret         :=  (return Expr)
Expr        :=  id | const | (if Expr Expr Expr) | (unary-op Expr) |
                (binary-op Expr Expr) | (nary-op Expr+)
unary-op    :=  bvneg | bvnot
binary-op   :=  = | bvule | bvult | bvuge | bvugt | bvsle | bvslt | bvsge | bvsgt |
                bvsdiv | bvsrem | bvshl | bvlshr | bvashr | bvsub
nary-op     :=  bvor | bvand | bvxor | bvadd | bvmul
id          :=  identifier
const       :=  32-bit integer | true | false
```

Assume the following well-formedness rules for programs, which your verifier does not need to check:

1. an identifier is not used before it is defined;
2. an identifier is not defined more than once;
3. the first sub-expression of an if-expression is of type boolean, and its remaining subexpressions have the same type.

The statement `(set! id Expr)` assigns the value of `Expr` to the variable `id`; the types of `id` and `Expr` must match. The inputs to a fragment are 32-bit integers.

The operators in the `BV` language have the same semantics as the corresponding operators in T_{bv} (see the Z3 tutorial on bitvectors). For example, the following `BV` programs correspond to P_1 and P_2 from Problem 2:

```
(define-fragment (P1 x1 y1 x2 y2)
  (return (bvmul (bvadd x1 y1) (bvadd x2 y2))))

(define-fragment (P2 x1 y1 x2 y2)
  (define u1 (bvadd x1 y1))
  (define u2 (bvadd x2 y2))
  (return (bvmul u1 u2)))
```

11. (5 points) The grammar for the `BV` language is designed in such a way that you do not need to convert a `BV` program to Static Single Assignment (SSA) form before translating it to bit vector logic. Explain in a few sentences what property of this grammar allows you to avoid SSA conversion.

12. (20 points) Implement a BMC verifier for the `BV` language in Racket, using the provided solution skeleton. See the `README` file for instructions on using the skeleton with Z3.

    Your verifier (see `verifier.rkt`) should take as input two `BV` program fragments (`examples.rkt` and `bv.rkt`); produce a `QF_BV` formula that is unsatisfiable iff the programs are equivalent; invoke Z3 on the generated formula (`solver.rkt`); and decode Z3's output as follows. If the programs are equivalent, the verifier should return `'EQUIVALENT`; otherwise it should return an input, expressed as a list of integers, on which the fragments produce a different output.

Inputs to the two programs should be the only unknowns (i.e., bitvector constants) in the `QF_BV` formula produced by your verifier. This means that the verifier cannot use additional constants to represent the values of program expressions and statements. But it should also not inline the translations of individual expressions. For example, consider the following `BV` fragment:

```
(define-fragment (toy b c)
  (define a (bvmul b c))
  (return (bvadd a a)))
```

The encoding may introduce two unknowns to represent the input variables `b` and `c`. But it may not translate the first statement by emitting an SMT-LIB equality assertion such as (assert (= a (bvmul b c))), where a is a fresh unknown. Similarly, it may not translate the return statement by inlining the encoding of the first statement, i.e., (bvadd (bvmul b c) (bvmul b c)).

(**Hint:** Your encoding may use SMT-LIB definitions, introduced by `define-fun`.)

Your entire encoding should fit into the `verifier.rkt` file. In particular, the `verify-all` procedure in `tests.rkt` (see Problem 13) should be executable just by placing your `verifier.rkt` into the `simple-verifier` directory, without modifying any supporting files. Your encoding will be tested and graded automatically, so it is important for the implementation to be self-contained, and to adhere to the input/output specification given above.

13. (5 points) Run your verifier on the benchmarks in `tests.rkt` and record the outcomes in table format:

Benchmark	**Outcome**	**Time (ms)**
max1 ≡ max2	`EQUIVALENT` or counterexample (57, 42)	k
⋮	⋮	⋮

(**Note:** We will also test your code on additional benchmarks that are not included in `tests.rkt`. To make sure that your verifier works correctly, you will need to write additional tests of your own, especially for corner cases.)