Homework Assignment 2
Due: November 10, 2023 at 5:00pm

Total points:	100
Deliverables:	`hw2.pdf` containing typeset solutions to Problems 1-11. `verifier.rkt` containing your implementation for Problem 6. `uf.rkt` containing your implementation for Problems 7-8. `nnf.als` containing your Alloy encoding for Problems 9-11.
Sources:	https://gitlab.cs.washington.edu/cse507/hw23au.

1 SMT Solving (30 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 07. In particular, show the intermediate partitions (sets of congruence classes) after each merge or propagation step, together with a brief explanation of how the algorithm arrived at that partition (e.g., "by literal $f(x) = y$, merge $f(x)$ with y").

2. 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 why. 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
...	...

3. (5 points) Recall that the theory of arrays $T_A = \{read, write, =\}$ is defined by the following axioms.
$$\forall a, i, j.\ i = j \rightarrow read(a, i) = read(a, j)$$
$$\forall a, v, i, j.\ i = j \rightarrow read(write(a, i, v), j) = v$$
$$\forall a, v, i, j.\ i \neq j \rightarrow read(write(a, i, v), j) = read(a, j)$$
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 $i \in [2 \ldots n]$.

4. (10 points) Prove that the theory of equality $T_=$ is convex.

2 A Verifier for Superoptimization (25 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, these are valid `BV` programs:

```
(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)))
```

5. (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.

6. (20 points) Implement a BMC-style verifier for the `BV` language in Racket, using the solution skeleton in the `bvv` directory. Your verifier (see `verifier.rkt`) should take as input two `BV` program fragments (see `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 encoding should be entirely contained in the `verifier.rkt` file. In particular, the `verify-all` procedure in `tests.rkt` should be executable just by placing your `verifier.rkt` into the `bvv` 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. Note also that we will test your code on 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.

3 Faster Verification With(out) Uninterpreted Functions (20 points)

One common application of uninterpreted functions, shown in Lecture 06, is to accelerate verification by abstracting away the details of operations that are expensive to compute. In this part of the assignment, you will use uninterpreted functions to accelerate two verification queries in a Rosette program. Then, you will accelerate these queries without uninterpreted functions.

The program and the queries are given in `matrix.rkt`. The program is a straightforward implementation of matrix multiplication, and the queries verify that the implementation satisfies two simple properties: `query-1` checks that $A = B \implies A * B = B * A$, and `query-2` checks that $B = A^T \implies (A * A)^T = B * B$.

Both queries take a long time to solve even on small symbolic matrices, e.g., 4×4. The goal is to get each query to complete in seconds on 20×20 matrices. We will take two different approaches to reach this goal.

7. (10 points) The first approach is to abstract a set of operators of your choosing with uninterpreted functions as described in `uf.rkt`. This set should include as few operators as possible.

   Your solution must be fully contained in your copy of `uf.rkt`. No other files may be modified. Submit your copy of `uf.rkt`, along with the typeset (short) answers to the following questions.

   What is the *minimal* set of operators you need to abstract with uninterpreted functions in order to accelerate both queries? Do your abstractions depend on any properties of these operators?

8. (10 points) Next, suppose that you cannot use uninterpreted functions, but you are allowed to rewrite each query to solve an *equivalent* verification problem. Briefly describe how to rewrite `query-1` and `query-2` into logically equivalent queries that achieve the same level of performance as in Problem 7, and briefly argue why your rewriting strategy results in equivalent verification problems. (**Hint:** thinking about equalities is key to solving this problem.)

4 Finite Model Finding with Alloy (25 points)

In this part of the assignment, you will formalize the syntax and semantics of propositional logic in Alloy, and use this formalization to check the theorem about the monotonicity of negation normal form (NNF) from Problem 3 in Homework 1.

To start, download Alloy-5.0.0.1.jar and double click on it to launch the Alloy Analyzer tool. You may also want to skim Parts 1–2 of the Alloy tutorial and this quick reference on Alloy syntax and semantics.

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

9. (10 points) Start by formalizing the syntax of propositional logic and NNF.

   (a) A *formula* in propositional logic can be a variable, negation, conjunction, or disjunction. A formula has zero or more children, depending on its type (e.g., variables have no children). We represent formulas as abstract syntax graphs that allow sharing of subformulas (so two different formulas may contain the same subformula) but disallow cycles (i.e., no formula contains itself). Formalize these constraints by completing the definition of the `wellFormedFormulas` fact. Use the Alloy Analyzer to check that your definition is correct (e.g., it rejects variables with children) and non-vacuous (i.e., it admits some formulas) in a universe with a small number of formulas.

   (b) An NNF formula satisfies additional constraints on its structure. What are those constraints? Formalize them by completing the definition of the `NNF` predicate. Check your definition for correctness and vacuity bugs.

10. (10 points) Next, formalize the semantics of propositional logic.

    (a) Specify the semantics by completing the definition of the `valuation` predicate. This predicate takes as input a relation from formulas to booleans, and returns true iff the input relation is a total function that maps each formula to its meaning. (**Hint:** to specify a formula's meaning, consider when the semantics function should map each kind of formula to `True`.)

    (b) Use the `valuation` predicate to define the `satisfies` predicate that specifies what it means for a valuation to satisfy a given formula.

    Check your definitions for correctness and vacuity bugs.

11. (5 points) Finally, finish formalizing the theorem about the monotonicity of NNF from Problem 3 in Homework 1. To do so, complete the definition of the `pos` function so that it returns the positive set of a valuation with respect to a formula. Check that the theorem holds for all formulas in a small universe. To guard against vacuity, also ensure that the Alloy Analyzer can find some formula and valuations that satisfy the premise and the conclusion of the theorem.