Homework Assignment 2

Due: May 04, 2016 at 11:00pm

Total points: 100
Deliverables: `hw2.pdf` containing typeset solutions to Problems 1-7.
`hw2.smt` containing your SMT-LIB encoding for Problem 4d.
`verifier.rkt` containing your implementation for Problem 6.

1 Variations on SAT (20 points)

Consider the following variations on the propositional satisfiability (SAT) problem discussed in Lecture 3:

Partial Weighted MaxSAT Given a CNF formula $\phi_H = \bigwedge_{c \in H} c$ corresponding to a set of *hard* clauses H, and a CNF formula $\phi_S = \bigwedge_{c \in S} c$ corresponding to a set of *soft* CNF clauses S with weights $w : S \to \mathbb{Z}$, the Partial Weighted MaxSAT problem is to find an assignment A to the problem variables that satisfies all the hard clauses and that maximizes the weight of the satisfied soft clauses. That is, $A \models \bigwedge_{c \in H} c$, and if we let $C = \{c \in S | A \models c\}$, then there is no $C' \subseteq S$ such that $H \cup C'$ is satisfiable and $\sum_{c' \in C'} w(c') > \sum_{c \in C} w(c)$.

Pseudo-Boolean Optimization Let B be a set of *pseudo-boolean constraints* of the form $\sum a_{ij} x_j \geq b_i$, where x_j is a variable over $\{0, 1\}$ and a_{ij}, b_i, c_j are integer constants. The Pseudo-Boolean Optimization problem is to satisfy all constraints in B while minimizing a linear function $\sum c_j \cdot x_j$.

1. (10 points) Explain how to encode a Partial Weighted MaxSAT problem P as a Pseudo-Boolean Optimization problem P' such that (1) P' is satisfiable iff P is satisfiable, and (2) a solution to P can be extracted from a solution to P'. Show the result of your encoding (the optimization function and the pseudo-boolean constraints) on the following example, which uses the pair notation to associate weights with soft clauses:

$$\begin{aligned} H &= \{(x_1 \vee x_2 \vee \neg x_3), (\neg x_2 \vee x_3), (\neg x_1 \vee x_3)\} \\ S &= \{\langle(\neg x_3), 6\rangle, \langle(x_1 \vee x_2), 3\rangle, \langle(x_1 \vee x_3), 2\rangle\} \end{aligned}$$

2. (10 points) Explain how to encode a Pseudo-Boolean Optimization P as a Partial Weighted MaxSAT problem P' such that (1) P' is satisfiable iff P is satisfiable, and (2) a solution to P can be extracted from a solution to P'. Assume the existence of a function *toCNF* that takes as input a pseudo-boolean constraint $\sum a_{ij} x_j \geq b_i$ and encodes it as a boolean circuit in CNF form. Show the result of your encoding (the set of hard clauses, and the set of soft clauses with their weights) on the following example:

$$\begin{array}{ll} \text{minimize} & 4x_1 + 2x_2 + x_3 \\ \text{subject to} & 2x_1 + 3x_2 + 5x_3 \geq 5 \\ & -x_1 - x_2 \geq -1 \\ & x_1 + x_2 + x_3 \geq 2 \end{array}$$

2 Theory of Equality and Uninterpreted Functions (30 points)

3. (10 points) Apply the congruence closure algorithm to decide the satisfiability of the following $T_=$ formula:

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

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").

4. (20 points) 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}).

(c) (5 points) Provide a *partial interpretation* of the relevant 32-bit integer operations that is sufficient for the proof in Problem 4b to succeed in $T_=$. The partial interpretation should take the form of axiom(s)—additional (universally quantified) formulas that constrain the uninterpreted function representing an operation to capture relevant properties of that operation.

(d) (5 points) Use BMC to construct a $T_=$ formula that is unsatisfiable iff P_1 is equivalent to P_3 from Problem 4b under the partial interpretation from Problem 4c. Express this encoding in SMT-LIB syntax, run Z3 on it with the `-st` and `-smt2` options, and report the resulting output. See the Z3 tutorial for examples of formulas in SMT-LIB syntax. Submit your SMT-LIB encoding in a separate `hw2.smt` file.

3 A Verifier for Superoptimization (50 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 4:

```
(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. (10 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. (30 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 7) should be executable just by placing your `verifier.rkt` into the `src` 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.

7. (10 points) Run your verifier on the benchmarks in `tests.rkt` and record the outcomes in table format:

Benchmark	**Outcome**	**Time (ms)**
max1 $\equiv$ max2	`EQUIVALENT` or counterexample (57, 42)	k
$\vdots$	$\vdots$	$\vdots$

(**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.)