EE219C HW2: SMT

Vighnesh Iyer

1 Bit-Twiddling Hacks

(a) Are the functions `f1` and `f2` in Figure 1 equivalent?

```
int f1(int x) {
  int v0;
  if (x > 0) v0 = x;
  else v0 = -x;
  return v0;
}
```

```
int f2(int x) {
  int v1, v2;
  v1 = x >> 31;
  v2 = x ^ v1;
  return (v2 - v1);
}
```

`f1` is an absolute value function. `f2` is first isolating the sign bit in `v1` then performing a 2s complement inversion if the sign bit is 1. So these functions should be equal. I encoded this validity question using the Z3 Python API:

```
x, v0, v1, v2 = BitVecs('x v0 v1 v2', 32)
s = Solver()
s.add(v0 != v2 - v1, v0 == If(x > 0, x, -x), v1 == x >> 32, v2 == x ^ v1)
print(s.check())
print(s.sexpr())
```

The equality between the return values of `f1` and `f2` was inverted to check for validity. The results were:

```
unsat
(declare-fun v0 () (_ BitVec 32))
(declare-fun x () (_ BitVec 32))
(declare-fun v2 () (_ BitVec 32))
(declare-fun v1 () (_ BitVec 32))
(assert (distinct v0 (bvsub v2 v1)))
(assert (= v0 (ite (bvsgt x #x00000000) x (bvneg x))))
(assert (= v1 (bvashr x #x00000020)))
(assert (= v2 (bvxor x v1)))
(model-add v0
           ()
           (_ BitVec 32)
           (bvmul x (ite (bvsle x #x00000000) #xffffffff #x00000001)))
(model-add v2 () (_ BitVec 32) (bvxor x v1))
```

Showing that `f1` and `f2` are functionally equivalent.

(b) Are the functions `f3` and `f4` in Figure 1 equivalent?

```
int f3(int x, int y) {
  int v0;
  if (x >= y) v0 = x;
  else v0 = y;
  return v0;
}
```

```
int f4(int x, int y) {
  int v1, v2, v3;
  v1 = x ^ y;
  v2 = (-(x >= y));
  v3 = v1 & v2;
  return (v3 ^ y);
}
```

`f3` is a max function. I used Z3 in the same manner:

```
x, y, v0, v1, v2, v3 = BitVecs('x y v0 v1 v2 v3', 32)
s = Solver()
s.add(v0 != v3 ^ y,
      v0 == If(x >= y, x, y),
      v1 == x ^ y,
      v2 == If(x >= y, BitVecVal(-1, 32), BitVecVal(0, 32)),
      v3 == v1 & v2
     )
print(s.check())
print(s.sexpr())
```

These two functions are also equivalent:

```
unsat
(declare-fun v0 () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))
(declare-fun x () (_ BitVec 32))
(declare-fun v1 () (_ BitVec 32))
(declare-fun v2 () (_ BitVec 32))
(declare-fun v3 () (_ BitVec 32))
(assert (distinct v0 (bvxor v3 y)))
(assert (= v0 (ite (bvsge x y) x y)))
(assert (= v1 (bvxor x y)))
(assert (= v2 (ite (bvsge x y) #xffffffff #x00000000)))
(assert (= v3 (bvand v1 v2)))
(model-add v0 () (_ BitVec 32) (ite (bvsle y x) x y))
(model-add v1 () (_ BitVec 32) (bvxor x y))
(model-add v2 () (_ BitVec 32) (ite (bvsle y x) #xffffffff #x00000000))
(model-add v3
           ()
           (_ BitVec 32)
           (let ((a!1 (bvor (bvnot (bvxor x y))
                            (bvnot (ite (bvsle y x) #xffffffff #x00000000)))))
             (bvnot a!1)))
```

2 Sum-Sudoku

(a) Describe your SMT encoding and list the constraints in it. Then encode the formulation using the Z3 API by implementing `var`, `val`, and `valid` in `sumsudoku.py`.

We are working in the theory of `QF_LIA`. Declare the following variables:

$$\begin{array}{rl} x_{i,j} & \forall i\; 0 \leq i < n\,, 0 \leq j < n \text{ where } x_{i,j} \text{ represents the value of a cell} \\ c_i & \forall i\; 0 \leq i < n \text{ where } c_i \text{ represents the sum of column } i \\ r_i & \forall i\; 0 \leq i < n \text{ where } r_i \text{ represents the sum of row } i \end{array}$$

Enforce the following constraints:

$$\begin{array}{rl} x_{i,j} \neq x_{i,k} & \forall i\,, 0 \leq j < k \leq n : \text{ values in each row are distinct} \\ x_{j,i} \neq x_{k,i} & \forall i\,, 0 \leq j < k \leq n : \text{ values in each column are distinct} \\ x_{i,j} \in \{1, \ldots, m\} & \forall i\,, 0 \leq i < n\,, 0 \leq j < n : \text{ values in each cell are between 1 and m} \\ \sum_{i=1}^{n} x_{i,j} = c_i & \forall j\,, 0 \leq n : \text{ sum across cells in a column equals the column sum} \\ \sum_{j=1}^{n} x_{i,j} = r_i & \forall i\,, 0 \leq n : \text{ sum across cells in a row equals the row sum} \end{array}$$

I implemented the functions in Python:

```
def transpose(l: List[List]) -> List[List]:
    return [list(i) for i in zip(*l)]
def var(name): return z3.Int(name)
def val(v): return z3.IntVal(v)
def valid(g):
    # Ensure all rows and all columns have unique values
    def unique_across_rows():
        for row in g[2]:
            for combo in itertools.combinations(row, 2):
                yield combo[0] != combo[1]
    rows_unique = reduce(lambda a, b: z3.And(a, b), unique_across_rows())
    def unique_across_cols():
        for row in transpose(g[2]):
            for combo in itertools.combinations(row, 2):
                yield combo[0] != combo[1]
    cols_unique = reduce(lambda a, b: z3.And(a, b), unique_across_cols())
    # Ensure all values are between 1 and m
    def values_in_range():
        for row in g[2]:
            for elem in row:
                yield z3.And(elem >= 1, elem <= m)
    vals_range = reduce(lambda a, b: z3.And(a, b), values_in_range())
    # Relate the row and column sums to the grid
    def row_relation():
```

```
        for row_num in range(n):
            yield g[0][row_num] == reduce(lambda a, b: a + b, g[2][row_num])
    row_sum_rel = reduce(lambda a, b: z3.And(a, b), row_relation())
    def col_relation():
        for col_num in range(n):
            yield g[1][col_num] == reduce(lambda a, b: a + b, transpose(g[2])[col_num])
    col_sum_rel = reduce(lambda a, b: z3.And(a, b), col_relation())
    return z3.And(rows_unique, cols_unique, vals_range, row_sum_rel, col_sum_rel)
```

Z3 produced this solution:

```
    |   8 |   8 |  12
---------------------
  8 |   2 |   1 |   5
 10 |   5 |   2 |   3
 10 |   1 |   5 |   4
---------------------
```

which doesn't match the solution in the homework, because this board doesn't have a unique solution.

(b) Use the pigeonhole principle, forall numbers there is only 1 place it can go into.