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:

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

Enforce the following constraints:

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

I implemented the functions in Python:

```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 has 8 solutions.

(b) There exists an assignment to all the row and column sums such that only 1 solution is possible. We can use a set of boolean variables to encode every possible filling out of the puzzle (m^{2n}) and enforce that this set is one-hot.

```
def create_puzzle():
    g1 = gridvars(('r', 'c', 'x'), n, m)
    rs, cs = g1[0], g1[1]
    S = z3.Solver()
    def do_assn():
        assignments = itertools.product(*[range(1, m+1) for r in
        ↪  range(2*n)])
        for a in assignments:
            v = z3.Bool('%s' % a.__str__())
            S.add(z3.Implies(v == True, reduce(lambda a, b: z3.And(a, b),
            ↪  map(lambda x: x[0] == x[1], zip(a, g1[3])))))
            S.add(z3.Implies(reduce(lambda a, b: z3.Or(a, b), map(lambda x:
            ↪  x[0] != x[1], zip(a, g1[3]))), v == False))
            yield v
    assn_vars = [x for x in do_assn()]
    S.add(z3.PbLe([(x,1) for x in assn_vars], 1))
```

But this didn't work when $n = 2$ and gave this result:

```
    |   7 |   5
---------------
  5 |   3 |   2
  7 |   4 |   3
---------------
```

which has 3 solutions. When checking the model I saw that only 1 assignment was hot, even though multiple ones should have been valid for the same row and column sums. So this technique doesn't work.

We can encode a uniqueness constraint using the `ForAll` and `Exists` quantifiers. In English: there exists an assignment to the row and column sums such that for all cell fillings of the puzzle only one is valid.

$$\exists\{r'_k, c'_k\}_{k=1,\dots,n}\forall\{c_{i,j}\}_{0\leq i<n,0<j<n}$$
$$((r'_k = r_k)\wedge(c'_k = c_k)\wedge V(r_k, c_k, c_{i,j}))\vee((r'_k\neq r_k)\wedge(c'_k\neq c_k)\wedge\neg V(r_k, c_k, c_{i,j}))$$

where V is the valid function defined above. I tried to encode this in Z3, like this:

```python
def create_puzzle():
    g1 = gridvars(('r', 'c', 'x'), n, m)
    rs, cs = g1[0], g1[1]
    S = z3.Solver()
    S.add(
        z3.Exists([*rs, *cs],
            z3.ForAll([*rs, *cs, *g1[3]],
                z3.Or(
                    z3.And(g1[0] == rs, g1[1] == cs, valid(g1)),
                    z3.And(g1[0] != rs, g1[1] != cs, z3.Not(valid(g1)))
                )
            )
        )
    )
```

but I got unsat and the runtime was long (around 10 seconds for $n = 3$). I suspect there's something wrong with references in the inner functions. How do I refer to the existential variables inside the lambda? I spent way too much time on this problem and all my encodings failed to get a unique puzzle construction.

(c) Start off with a filled puzzle and remove entries as much as possible while retaining a unique solution.

The procedure is to start with a filled puzzle, generate tuples of cells indicating whether they should be 'emptied', encode those removals and ensuing restrictions as SMT constraints, solve the model and if it is UNSAT then the original puzzle solution is still unique, otherwise there is another solution. My implementation:

```python
def lock_cells(S, flags, g, grid_vals):
    assert len(flags) == len(g[3]) == len(grid_vals)
    fixed_cells = filter(lambda x: x[2], zip(g[3], grid_vals, flags))
    S.add(reduce(lambda a, b: z3.And(a, b), map(lambda x: x[0] == x[1],
    ↪   fixed_cells), True))
    avoid_prev_assn = filter(lambda x: not x[2], zip(g[3], grid_vals,
    ↪   flags))
    S.add(reduce(lambda a, b: z3.Or(a, b), map(lambda x: x[0] != x[1],
    ↪   avoid_prev_assn), False))
```

The lock_cells function adds constraints to the solver that fix the values of particular cells marked True by flags, and also constrain the ”empty” (i.e. free) cells to not repeat the previous solution. Now the main function:

```
def make_puzzle_unique(initial_grid):
    (row_vals, col_vals, grid_vals) = initial_grid
    g1 = list(gridvars(('r', 'c', 'x1'), n, m))

    S = z3.Solver()
    S.add(valid(g1))
    S.add(reduce(lambda a, b: z3.And(a, b), map(lambda x: x[0] == x[1],
    ↪  zip(g1[0], row_vals))))
    S.add(reduce(lambda a, b: z3.And(a, b), map(lambda x: x[0] == x[1],
    ↪  zip(g1[1], col_vals))))
    valid_flags = []
    for removals in range(9):
        for i in itertools.combinations(range(n*n), removals):
            # i represents cells which are 'empty'
            flags = [False if x in i else True for x in range(n*n)]
            S.push()
            lock_cells(S, flags, g1, grid_vals)
            r = S.check()
            # if there's no model, then there still exists a unique solution
            if r == z3.unsat:
                valid_flags.append(flags)
            S.pop()
            print(removals, i, r)
    flags = valid_flags[-1]
    print(flags)
    lock_cells(S, flags, g1, grid_vals, bypass=True)
```

I first fix the row and column sums to the values from the fully filled puzzle. Then brute force all the possible ways one could empty out the puzzle and choose the most empty board which still returns UNSAT. I found that there were several ways of removing 7 cells which still gave a unique solution, but removing 8 cells in any way would give always give a SAT board (no more unique solutions).

3 Bag of Chips

(a) Define your well-founded relation and why it guarantees termination.

I used this well-founded relation which was kind of modeled off the get_chip() logic. The idea is that the elements of the tuple are checked in a priority order which ensures that there is a consistent global ordering of states. The well-founded relation guarantees termination because it enforces that every subsequent state must be ’smaller’ than the previous one in less than infinite timesteps, and this fact is used in an inductive proof to guide the sequence of states to when every element in the tuple is 0.

```
def relation(p, q):
    return z3.Or(
            z3.And(q[0] != p[0], q[0] < p[0]),
            z3.And(q[0] == p[0], q[1] != p[1], q[1] < p[1]),
            z3.And(q[0] == p[0], q[1] == p[1], q[2] < p[2])
    )
```

(b) Encode the new variation of the problem in the case functions and a new relation to prove termination

Encoding the cases is straightforward:

```
def case1(color1, color2, state):
    (ycnt, bcnt, rcnt) = state
    cnd = z3.Or(
            z3.And(color1 == RED, color2 == YELLOW),
            z3.And(color2 == RED, color1 == YELLOW))
    rcntp = z3.If(cnd, rcnt+1, rcnt)
    return (ycnt, bcnt, rcntp)

def case2(color1, color2, state):
    (ycnt, bcnt, rcnt) = state
    cnd = z3.And(color1 == YELLOW, color2 == YELLOW)
    bcntp = z3.If(cnd, bcnt+5, bcnt)
    return (ycnt, bcntp, rcnt)

def case3(color1, color2, state):
    (ycnt, bcnt, rcnt) = state
    cnd = z3.Or(
            z3.And(color1 == BLUE, color2 == RED),
            z3.And(color2 == BLUE, color1 == RED))
    bcntp = z3.If(cnd, bcnt+10, bcnt)
    return (ycnt, bcntp, rcnt)
```

The new well-founded relation needs to be modified slightly to take into account the 'elimination order' of this new algorithm variant which I analyzed by inspection to be yellow first, then red, and finally blue. I adjusted the relation accordingly to swap the comparison order between the red and blue chips because red chips are eliminated first by `case3`. This maintains a strict state ordering which guarantees termination of the algorithm.

```
def relation(p, q):
    return z3.Or(
            z3.And(q[0] != p[0], q[0] < p[0]),
            z3.And(q[0] == p[0], q[2] != p[2], q[2] < p[2]),
            z3.And(q[0] == p[0], q[2] == p[2], q[1] < p[1])
    )
```

(c) Pitfalls of showing the inverse of an implication is UNSAT? Say that A_1 was incorrectly encoded such that it was always false. The SMT formula that is checked is still going to be UNSAT if the other statements were always false under any variable assignment. So we could get a 'proof' of the initial implication without knowing that A_1 is improperly encoded. This can be mitigated by adding additional assertions to check the validity of the final SMT formula with each A_i dropped out, to check that each term has an impact on the final solution.