Problem 1. Pointer Analysis

Perform pointer analysis on the following Java snippet, and answer the following questions.

```java
public class Foo {
    public Foo a;
    public Foo b;
    public static main(String [] args) {
        Foo x = new Foo(); // h1
        Foo y = new Foo(); // h2
        bar(x, y);
        y.a = x;
        bar(y, x);
    }
    public static void bar(Foo p, Foo q) {
        if(p.a != q.b) return;
        p.b = new Foo(); // h3
        baz(p, q);
        qux(p, q);
    }
    public static void baz(Foo c, Foo d) {
        c.a = d;
    }
```
1. What are the hP tuples inferred from this code in a context insensitive analysis? To get you started here’s one: hP(h1, a, h2).

   x points to h1, y points to h2, all other variables point to either h1 or h2. Given that, the list of hP is:

   hP(h1, a, h1)
hP(h1, a, h2)
hP(h2, a, h1)
hP(h2, a, h2)
hP(h1, b, h1)
hP(h1, b, h2)
hP(h2, b, h1)
hP(h2, b, h2)
hP(h1, b, h3)
hP(h2, b, h3)

   Observe that hP(h1, a, h1) and hP(h2, a, h2) are in the list, even though they are impossible from the graph. This is due to context insensitivity in the graph.

2. Draw the call-graph of the original program and the call-graph after running the cloning-based algorithm for context-sensitive analysis. Do not make clones for recursive calls.

   Context-insensitive call graph:
Notice that the SCC “bar-baz-qux” is cloned as whole. An alternative solution would be to clone baz as well inside the SCC. This results in a bigger set of contexts, and it is not clear it would help in general, but it is not incorrect, and therefore will not result in lost points (provided it is consistent with the next answers).

3. How many copies of each procedure are created (for bar, baz and qux)?
   Two copies each.

4. What are the hP tuples inferred from this code in a context sensitive analysis?
   Same as before, x points to h1, y points to h2, and all other variables point to either h1 or h2, except now the situation when p points to h1 and q to h2 is differentiated from the one where p points to h2 and q to h1, so the algorithm can observe that p != q always (and similarly for c, d and m, n), removing the spurious self loops. The final list:

   hP(h1, a, h2)
   hP(h2, a, h1)
   hP(h1, b, h2)
   hP(h2, b, h1)
   hP(h1, b, h3)
   hP(h2, b, h3)

Problem 2. Binary Decision Diagrams

Draw the optimal (minimum number of nodes) BDD for the following expression:

$$\exists x_2, (x_1 \land x_0) \lor (x_1 \land x_3)$$

We first observe that $x_2$ does not appear in the body of the exists() expression, so we can ignore it. We then rewrite the expression as $x_1 \land (x_0 \lor x_3)$ and observe that if $x_1$ is false, the whole expression is false, so we put it first. The order of $x_0$ and $x_3$ then does not matter.

Result:
1. What is the variable order you chose to create this BDD?

\( x_1, x_0, x_3 \)

2. What is the minimum number of nodes?

5 nodes (or 3 if you don’t count the 1 and 0 nodes - which are always there unless the BDD is a tautology/contradiction)

Problem 3. Path Sensitive Analysis With Satisfiability Modulo Theories

In this problem you will use an SMT solver to find test cases exhibiting a bug in the following C function:

```c
int func(int x, int[] data, int N) {
    int v, z;
    if (0 <= x && x < N) {
        if (x <= N/2) {
            x = 2 * x;
        }
        v = data[x]; // line 7
        if (v >= 0 && v < N) {
            z = data[v]; // line 9
        } else {
            z = 0;
        }
        if (z < v) {
            data[z] = 0; // line 14
        }
    }
}
```
return data[data[z]]; // line 16
} else {
    return data[0]; // line 18
}
Note: you DID NOT need to submit the SSA code with your solution. This is just the explanation for the solution.

2. Translation to SMT. The second step is to translate into an SMT formula. An assignment $x_3 = e$ becomes an assertion $(\text{assert } (= x_3 E))$ in SMT, where $E$ is the translation of $e$.

You need to translate int operations at the C level into bit vector operations at the SMT level. Do not use Int in Z3, as these model mathematical integers. Assume 32-bit 2's complement representation for int.

For example, the translation of $x = y + 1$ is:

```
(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(assert (= x (bvadd y #x00000001)))
(check-sat)
(get-model)
```

Z3 responds with:

```
sat
(model
 (define-fun y () (_ BitVec 32) #x00000000)
 (define-fun x () (_ BitVec 32) #x00000001)
)
```

Here the program is satisfiable with model $x = 1$, $y = 0$. Note that the variables $x$ and $y$ are given as functions of no arguments (which must be constants because there are no side effects), and the constants themselves are hexadecimal.

To translate arrays, you will use variables of the sort $(\text{Array } (_ \text{ BitVec 32}) (_ \text{ BitVec 32}))$. Array dereferences like $\text{data}[i]$ become $(\text{select data i})$ when translating a read, and $(\text{store data i x})$ when translating a write. Note that $(\text{store data i x})$ returns a new array, whose $i$'s element is now equal to $x$, and does not modify the original array.

To translate phi-nodes, you must use a logical expression that captures the condition under which the phi node is evaluated. For example, given the following code in SSA form:
if (c) {
  b1:
    x1 = ...;
} else {
  b2:
    x2 = ...;
}

x3 = phi(x1 from b1, x2 from b2);

the translation of x3 is (ite c x1 x2). ite is short for if-then-else, and evaluates to
the second or third argument based on the first.

Solution:

(declare-fun x1 () (_ BitVec 32))
(declare-fun x2 () (_ BitVec 32))
(declare-fun x3 () (_ BitVec 32))
(declare-fun data1 () (Array (_ BitVec 32) (_ BitVec 32)))
(declare-fun data2 () (Array (_ BitVec 32) (_ BitVec 32)))
(declare-fun data3 () (Array (_ BitVec 32) (_ BitVec 32)))
(declare-fun N () (_ BitVec 32))
(declare-fun v () (_ BitVec 32))
(declare-fun z1 () (_ BitVec 32))
(declare-fun z2 () (_ BitVec 32))
(declare-fun z3 () (_ BitVec 32))
(declare-fun ret1 () (_ BitVec 32))
(declare-fun ret2 () (_ BitVec 32))
(declare-fun ret () (_ BitVec 32))
(declare-fun zero () (_ BitVec 32))
(declare-fun two () (_ BitVec 32))
(assert (= zero #x00000000))
(assert (= two #x00000002))

(declare-fun cond1 () Bool)
(declare-fun cond2 () Bool)
(declare-fun cond3 () Bool)
(declare-fun cond4 () Bool)

(assert (bvsge N zero))
(assert (= cond1 (and (bvsle zero x1) (bvsle x1 N))))
(assert (= cond2 (bvsle x1 (bvsdiv N two))))
(assert (= x2 (bvmul two x1))
(assert (= x3 (ite cond2 x2 x1)))
(assert (= v (select data1 x3)))
(assert (= cond3 (and (bvsge v zero) (bvsle v N))))
(assert (= z1 (select data1 v)))
(assert (= z2 zero))
(assert (= z3 (ite cond3 z1 z2)))
(assert (= cond4 (bvslt z3 v)))
(assert (= data2 (store data1 z3 zero)))
(assert (= data3 (ite cond4 data2 data1)))
(assert (= ret1 (select data3 (select data3 z3))))
(assert (= ret2 (select data3 zero)))
(assert (= ret (ite cond1 ret1 ret2)))

Note the assert on N. N is the length of data, and is a positive number.

3. **Bounds checks.** The final step is to add an assertion to check each of the three particular array accesses. You need to check that the signed value of the index is in bounds. Further, not all accesses are accessible on all paths, so you need to guard the assertion for a particular access. The assertion should express “the execution reaches this access, and it is out of bounds.” Note that this will possibly constrain some path variables if the access is nested inside an if statement, for example.

You should use the sequence `(push)(assert C)(check-sat)(pop)` for each access, where C is the check for that access. Push/pop allows us to add C to our set of assertions, check satisfiability, and then remove it to add a different C. If you add all the assertions together you will find a path which crash all points simultaneously rather than just at least one.

**Solution:**

```
(define-fun out-of-bounds ((index (_ BitVec 32))) Bool
  (or (bvslt index zero) (bvsge index N)))

(push)
(assert (and cond1 (out-of-bounds x3))) ; line 7
(check-sat)
(get-model)
(pop)

(push)
(assert (and cond1 cond3 (out-of-bounds v))) ; line 9
(check-sat)
(pop)

(push)
(assert (and cond1 cond4 (out-of-bounds z3))) ; line 14
(check-sat)
(get-model)
(pop)
```
The solution uses a `out-of-bounds` helper, although you could have inlined it in every call site.

4. **Interpretation.** Once you find one or more bug, add `(get-model)` after `(check-sat)` within the push/pop sequence for each satisfiable assertion, to print the model found for that bug.

In writing, interpret the results. Does the result indicate a crash can occur on some concrete path? If not, does this mean there can be no crash for this access? If there is a crash, translate the model into a concrete input represented by a call `data = ...; func(...)` which causes a crash at the corresponding access. Hint: what happens if the program has more than one bug, and how should the model be interpreted in that case?

We observe two things. First, we made no approximation in the translation to SMT: this is a loop-free, recursion-free program with no pointers (other than arrays) and no IO. Therefore, if SMT says there is a solution, we have a bug, and if SMT says there is no solution, there is no bug.

To translate a bug into a test case, we look at the model and set the variables at the input accordingly. That does not mean that any SMT solution is a test case directly: we could encounter a case where the program is buggy twice. Practically speaking, SMT assumes infinitely sized arrays, and will happily read and write out of bounds when finding the model for the second buggy instruction. Depending on the machine and OS, this might actually work (and SMT would then model reading from random unrelated memory), or it might crash, and leave us without a test case. We could further constrain the later accesses to make the previous ones within bounds, but that’s not needed (and it makes the formula bigger and slower): a bug is a bug anyway, and we’re going to fix it. You fix the bug on the first access, rerun SMT, and now if the second access is still buggy you have yourself a test case.
Therefore, for the purpose of the homework, asserting that previous accesses are valid or not are both acceptable solutions.

sat
(model
  (define-fun N () (_ BitVec 32)
    #x0f004160)
  (define-fun cond3 () Bool
    false)
  (define-fun z1 () (_ BitVec 32)
    #x00000001)
  (define-fun zero () (_ BitVec 32)
    #x00000000)
  (define-fun data3 () (Array (_ BitVec 32) (_ BitVec 32))
    (_ as-array k!1))
  (define-fun cond2 () Bool
    true)
  (define-fun cond1 () Bool
    true)
  (define-fun x2 () (_ BitVec 32)
    #x0f004160)
  (define-fun z3 () (_ BitVec 32)
    #x00000000)
  (define-fun x1 () (_ BitVec 32)
    #x078020b0)
  (define-fun ret1 () (_ BitVec 32)
    #x00000000)
  (define-fun z2 () (_ BitVec 32)
    #x00000000)
  (define-fun data1 () (Array (_ BitVec 32) (_ BitVec 32))
    (_ as-array k!1))
  (define-fun cond4 () Bool
    false)
  (define-fun data2 () (Array (_ BitVec 32) (_ BitVec 32))
    (_ as-array k!0))
  (define-fun ret2 () (_ BitVec 32)
    #x00000001)
  (define-fun x3 () (_ BitVec 32)
    #x0f004160)
  (define-fun v () (_ BitVec 32)
    #x80000001)
  (define-fun two () (_ BitVec 32)
    #x00000002)
  (define-fun ret () (_ BitVec 32)
    #x00000000))
Line 7 can crash: we follow cond1 and cond2; x2 is exactly equal to N, so x in input is N/2

unsat
(let (($x179 (bvsle (_ bv0 32) v)))
(let (($x100 (bvsle N v)))
(let (($x240 (not $x179)))
(let (($x241 (or $x240 $x100)))
(let (($x242 (not $x241)))
(let (($x43585 (and cond1 cond3 (or (bvslt v zero) (bvsge v N)))))
(let (($x97 (bvsle zero v)))
(let (($x43591 (not $x97)))
(let (($x43597 (or $x43591 $x100)))
(let (($@x43599 (monotonicity (rewrite (= (bvslt v zero) $x43591)) (rewrite (= (bvsge v N)) (= cond1 cond3 $x43597)))
(let (($@x43604 (mp (asserted $x43585) (monotonicity @x43599 (= $x43585 (and cond1 cond3 $x43597)))
(let (($@x43607 (and-elim @x43604 cond3)))
(let (($x245 (= cond3 $x242)))
(let (($@x247 (monotonicity (rewrite (= (and $x179 (not $x100)) $x242)) (= cond3 ($x101 (not $x100))))
(let (($x166 (and $x179 $x101)))
(let (($x201 (= cond3 $x166)))
(let (($x29 (= zero (_ bv0 32)))
(let (($x33 (= (_ bv0 32) zero)))
(let (($@x198 (symm (mp (asserted $x29) (rewrite (= $x29 $x33) $x33) $x29)))
(let (($@x200 (monotonicity (monotonicity @x198 (= $x97 $x179))) (and $x97 $x101)))
(let (($x104 (and $x97 $x101)))
(let (($x107 (= cond3 $x104)))
(let (($@x106 (monotonicity (rewrite (= (bvsge v zero) $x97)) (rewrite (= (bvslt v N)) (= cond3 (and (bvsge v zero) (bvslt v N)))) $x109)
(let (($@x109 (monotonicity @x106 (= cond3 (and (bvsge v zero) (bvslt v N)))))
(let (($x91 (and (bvsge v zero) (bvslt v N)))
(let (($x92 (= cond3 $x91)))

11
Line 9 cannot crash. Intuitively, there is a bound check immediately after. Formally, you can walk through the proof.

Note that the proof was NOT needed for this submission (because SMT proofs are not as readable as SMT models, and anyway we can just assume the SMT is correct).
Line 14 can crash. We take cond1, not cond2, cond3, and cond4. The array data contains 0 in position 0, a negative number in position 1, and other stuff in other places. v is set to 1 (by reading from a high position of the array), so \( x_3 = \text{data}[v] \) is the negative number. Hence, crash.

\[
\text{sat}
\]

\[
\text{model}
\]

```scheme
(define-fun data2 () (Array (_ BitVec 32) (_ BitVec 32)) (_ as-array k!2))
(define-fun ret2 () (_ BitVec 32) #x00000000)
(define-fun x3 () (_ BitVec 32) #x24021560)
(define-fun v () (_ BitVec 32) #x00000001)
(define-fun two () (_ BitVec 32) #x00000002)
(define-fun ret () (_ BitVec 32) #x00000000)
(define-fun k!2 ((x!0 (_ BitVec 32))) (_ BitVec 32)
  (ite (= x!0 #x80000000) #x00000000
       (ite (= x!0 #x00000000) #x00000000
            (ite (= x!0 #x24021560) #x00000001
                 (ite (= x!0 #x00000001) #x80000000
                     #x20010ab0)))))
)
```
The inner access of line 16 can crash. The justification is the same as the previous one (because it is the same access really). If you asserted no consecutive crashes, then this one cannot crash.
(define-fun cond2 () Bool
  false)
(define-fun cond1 () Bool
  true)
(define-fun x2 () (_ BitVec 32)
  #x00042ac0)
(define-fun z3 () (_ BitVec 32)
  #x00002000)
(define-fun x1 () (_ BitVec 32)
  #x00021560)
(define-fun ret1 () (_ BitVec 32)
  #x00000000)
(define-fun z2 () (_ BitVec 32)
  #x00000000)
(define-fun data1 () (Array (_ BitVec 32) (_ BitVec 32))
  (_ as-array k!5))
(define-fun cond4 () Bool
  false)
(define-fun data2 () (Array (_ BitVec 32) (_ BitVec 32))
  (_ as-array k!4))
(define-fun ret2 () (_ BitVec 32)
  #x10000000)
(define-fun x3 () (_ BitVec 32)
  #x00021560)
(define-fun v () (_ BitVec 32)
  #x00000001)
(define-fun two () (_ BitVec 32)
  #x00000002)
(define-fun ret () (_ BitVec 32)
  #x00000000)
(define-fun k!4 ((x!0 (_ BitVec 32))) (_ BitVec 32)
  (ite (= x!0 #x00002000) #x00000000
       (ite (= x!0 #x80000000) #x00000000
            (ite (= x!0 #x00021560) #x00000001
                 (ite (= x!0 #x00000001) #x00002000
                      (ite (= x!0 #x00000000) #x10000000
                           #x00042ac0))))))
(define-fun k!5 ((x!0 (_ BitVec 32))) (_ BitVec 32)
  (ite (= x!0 #x00002000) #x80000000
       (ite (= x!0 #x80000000) #x00000000
            (ite (= x!0 #x00021560) #x00000001
                 (ite (= x!0 #x00000001) #x00002000
                      #x00042ac0))))
The outer access in line 16 can also crash. We take cond1, not cond2, cond3 and not cond4. This means we don’t overwrite data[z3] with zero, and it stays whatever it comes from the input. The z3 access is in fact in bounds. Data initially contains a high positive number at 0, the value for z3 at one, and a negative number at z3.

```
sat
(model
  (define-fun N () (_ BitVec 32)
    #x00000000)
  (define-fun cond3 () Bool
    false)
  (define-fun z1 () (_ BitVec 32)
    #x00080001)
  (define-fun zero () (_ BitVec 32)
    #x00000000)
  (define-fun data3 () (Array (_ BitVec 32) (_ BitVec 32))
    (_ as-array k!7))
  (define-fun cond2 () Bool
    false)
  (define-fun cond1 () Bool
    false)
  (define-fun x2 () (_ BitVec 32)
    #x00801adc)
  (define-fun z3 () (_ BitVec 32)
    #x00000000)
  (define-fun x1 () (_ BitVec 32)
    #x00400d6e)
  (define-fun ret1 () (_ BitVec 32)
    #x00001000)
  (define-fun z2 () (_ BitVec 32)
    #x00000000)
  (define-fun data1 () (Array (_ BitVec 32) (_ BitVec 32))
    (_ as-array k!7))
  (define-fun cond4 () Bool
    false)
  (define-fun data2 () (Array (_ BitVec 32) (_ BitVec 32))
    (_ as-array k!6))
  (define-fun ret2 () (_ BitVec 32)
    #x80001000)
  (define-fun x3 () (_ BitVec 32)
    #x00400d6e)
  (define-fun v () (_ BitVec 32)
    #x80000002))
)
(define-fun two () (_ BitVec 32) #x00000002)
(define-fun ret () (_ BitVec 32) #x80001000)
(define-fun k!6 ((x!0 (_ BitVec 32))) (_ BitVec 32) 
  (ite (= x!0 #x80001000) #x00001000
  (ite (= x!0 #x00000000) #x00000000
  (ite (= x!0 #x00400d6e) #x80000002
  (ite (= x!0 #x80000002) #x00080001
    #x80000002))))
)

(define-fun k!7 ((x!0 (_ BitVec 32))) (_ BitVec 32) 
  (ite (= x!0 #x80001000) #x00001000
  (ite (= x!0 #x00000000) #x80001000
  (ite (= x!0 #x00400d6e) #x80000002
  (ite (= x!0 #x80000002) #x00080001
    #x80000002)))))
)

Line 18 can crash. We don’t take cond1. N is 0 (the data array is empty).

5. **Submission.** Print the Z3 input and output, attach it to your homework with the written answers to part 4, and hand it in before class. If you do not have access to a printer, you can email a txt file with the Z3 input and output to the staff mailing list.