CS 243 Homework 7

Winter 2024

Due: March 13, 2024 at 11:59pm

Directions:

- Submit written answers via Gradescope.
- Submit the SMT file for Problem 6 via the separate Gradescope assignment.
- You may use up to two of your remaining late days for this assignment, for a late deadline of March 15, 2024 at 11:59pm.
- This is an individual assignment. You are allowed to discuss the homework with others, but you must write the solution individually. If you look up any material in the textbook or online, you should cite it appropriately.

Problem 1. For each of the following scenarios, what kind of memory management would you suggest (manual, reference counting, trace-based garbage collector)? If you recommend garbage collection, which garbage collector algorithm would you recommend? You must justify your answer to each question.

1. A long-running web server.
2. An embedded system for real-time signal processing.
3. Large-scale numerical scientific computation.

Problem 2. Garbage Collection

The runtime for a fictional language uses a generational garbage collection with 2 generations. `x` is a root object. Initially, `a` is in the older generation and `b` is in the newer generation (nursery). Both are reachable from the stack. All objects on the stack are rooted. There are no other objects, and all fields are initially null.

You are given the list of operations performed by the mutator. For each operation, write the new state of the heap, distinguishing reachable and unreachable objects. Do not indicate collected (freed) objects.

The operation `minor-gc` indicates a GC of the newer generation, and `major-gc` indicates a GC of all generations. Objects that survive a minor GC are promoted to the older generation.

1. `c = new Object`
2. `a.f = c`
3. `a = null`
4. `d = new Object`
5. `c = null`
6. `minor-gc`
7. `x.f = d`
8. `b = null`
9. `minor-gc`
10. `major-gc`

Problem 3. Let a be the number of live ranges in a program, b be the number of *merged* live ranges (as defined for register allocation), and c be the number of variables in the same program translated to static single-assignment (SSA) form. Express the relationship between a, b, and c using one or more inequalities. Explain your answer.

Problem 4. Recall that a formula F is satisfiable if there exists an interpretation I of the underlying variables and functions that makes the formula $\texttt{F}^I$ true. To check if a formula F is satisfiable, we can supply the following notation to the cvc5[1] SMT solver (ignoring declarations):

```
(assert F)
(check-sat)
```

(check-sat) returns true if F is satisfiable.

Given two input formulae F1 and F2, explain how would you use SMT solvers to perform the following operations by providing a similar notation as the example above:

1. Verify the validity of F1 (Recall that a formula F is valid if it is true under all interpretations)

2. Verify that for all models where F1 is satisfied, F2 is also satisfied.

3. For the following two cases, provide the SMT notation and explain the relationship between the two cases:

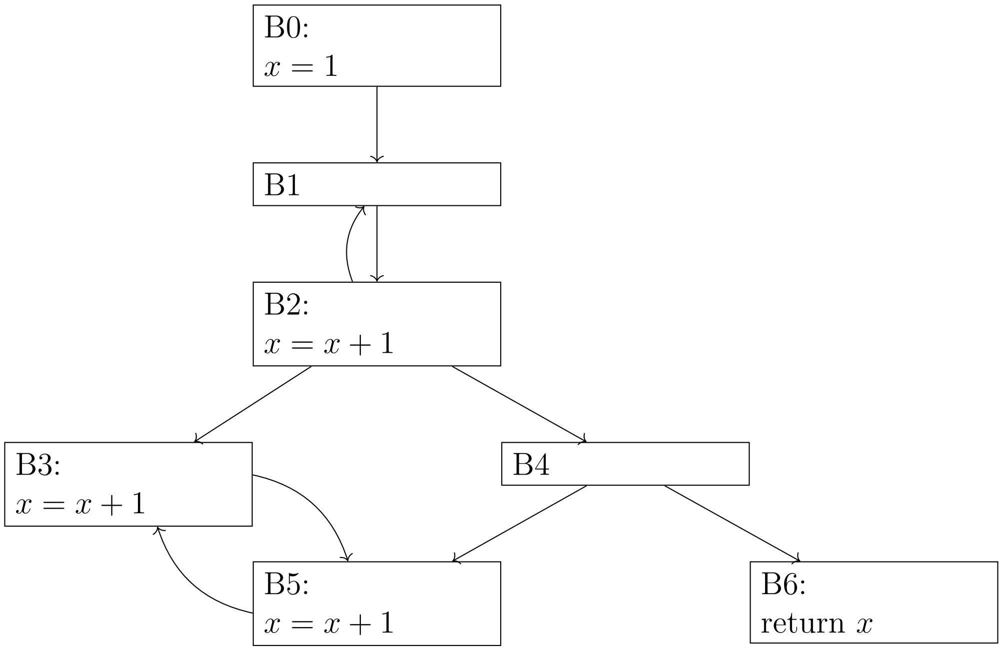
   (a) F1 is satisfiable and F2 is satisfiable

   (b) (F1 $\wedge$ F2) is satisfiable

[1] https://cvc5.github.io/

Problem 5. SSA Form.

In this problem you will find the SSA form for the following program:



Part a. Find the dominator tree for the given program's CFG.

Part b. Insert ϕ-functions (without operands) and show the resulting program.

Part c. Rename variables by assigning unique numbers to each definition and show the resulting program.

Problem 6. Path-Sensitive Analysis with Satifisfiability 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 data[], int N, int x) {
    if (0 <= x && x < N && N > 0 && N < 5000) {
        int m = data[x + 1];     // access
        if (m < 0 || m >= N) {
            m = 1;
        }
        int y = (x * m) % N;
        int i = data[y];         // access
        if (i < 0 || i >= N) {
            return 0;
        }
        m = data[data[y]];       // access
        int z = (x * m * i) % N;
        i = data[z];             // access
        return (i + m) % N;
    } else {
        return 0;
    }
}
```

We are interested in checking whether the program can "crash" due to array out-of-bounds accesses. This means that the index with which we access the `data` array is either less than zero, or greater than `N-1`. To do so, first you will translate this function into an SMT formula. Then you will run an SMT solver on the formula, and interpret its output.

You may make the following assumptions about the program: `N` is at least 0, `data` is an array of length `N`, `int` refers to signed 32-bit integer, and the program terminates (crashes) as soon as the first out-of-bounds access happens.

Using SMT solvers. The research community has produced a large number of SMT solvers. For this assignment, you can use cvc5[2], a state-of-the-art SMT solver used in both research and industry. You can use cvc5's online interface to avoid the hassle of installation (link in footnote), or optionally you can download the solver binaries from its website to run locally.

To get started with writing SMT formulae, you can refer to an introductory guide[3]. The full SMT language specification is available at the SMT-LIB website[4]. For this assignment, you can use any feature defined in SMT-LIB, but the following features should be sufficient:

> Commands: `assert`, `check-sat`, `declare-const`, `declare-func`, `get-model`, `pop`, `push`

[2] https://cvc5.github.io/; online interface at https://cvc5.github.io/app/

[3] https://microsoft.github.io/z3guide/docs/logic/intro

[4] https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf

Sorts: `Array`, `BitVec`, `Bool`

Core functions: `=`, `and`, `false`, `ite`, `not`, `or`, `true`

`Array` functions: `select`, `store`

`BitVec` functions: `bvadd`, `bvsub`, `bvmul`, `bvsdiv`, `bvsrem` (% operator), `bvsle/lt/ge/gt`

Static single-assignment form. It is easier to translate the program into a SMT formula if we can assume that every variable is defined exactly once. One way to satisfy this assumption for imperative code is by transforming the program into *static single-assignment* (SSA) form. To do this, first assign every variable definition a unique suffix. Then, at each join point in the control-flow graph (e.g., after branching), introduce new definitions for the variables that are defined on either path using a ϕ-node.

See the following example which transforms the imperative code on the left into SSA form on the right (here, instead of normal ϕ-functions, we have used the actual branch condition so that the SMT solver can reason about which branch was taken):

```
if (i < next) {
    if (data[i] == cookie)
        i++;
    else
        process(data[i]);

    i++;

    if (i < next) {
        if (data[i] == cookie)
            i++;
        else
            process(data[i]);

        i++;
    }

}

```

```
if (φ_1 = (i_0 < next)) {
    if (φ_2 = (data[i_0] == cookie))
        i_1 = i_0 + 1;
    else
        process(data[i_0]);
    i_2 = φ_2 ? i_1 : i_0;
    i_3 = i_2 + 1;

    if (φ_3 = (i_3 < next)) {
        if (φ_4 = (data[i_3] == cookie))
            i_4 = i_3 + 1;
        else
            process(data[i_3]);
        i_5 = φ_4 ? i_4 : i_3;
        i_6 = i_5 + 1;
    }
    i_7 = φ_3 ? i_6 : i_3;
}
i_8 = φ_1 ? i_7 : i_0;
```

Follow the following steps to complete this problem.

1. **Rewrite the program in SSA form.** Notice that at different points within our function, variables `v` and `z` may refer to different definitions. Transform the function into SSA form as illustrated earlier.

2. **Translation to SMT.** The second step is to translate the program in SSA form into an SMT formula. Start your formula with the following lines:

```
(set-logic ALL)
(set-option :produce-models true)
(set-option :incremental true)
```

An assignment $x_3 = e$ becomes an assertion (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 the SMT-LIB `Int` sort, which models *unbounded* integers.) 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)
```

Upon this query, an SMT solver could respond with:

```
sat
(
(define-fun x () (_ BitVec 32) #b00000000000000000000000000000001)
(define-fun y () (_ BitVec 32) #b00000000000000000000000000000000)
)
```

meaning that the SMT formula 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 given in binary or hexadecimal.

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

To translate ϕ-nodes, you must use a logical expression that captures the condition under which the ϕ-node is evaluated. For example, given the following code in SSA form:

```
if (c)
    x1 = ...;
else
    x2 = ...;
x3 = φ(x1, x2);    // equivalent to x3 = c ? x1 : x2;
```

the translation of x_3 is (ite c x_1 x_2). `ite` is short for if–then–else, and evaluates to the second or third argument based on the value of the first.

3. **Bounds checks.** The final step is to add an assertion to check each of the 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. For each access, you should

use the sequence `(push) (assert` C`) (check-sat) (pop)`, where C is the check for that access. Push/pop allows us to add C to our set of assertions temporarily, check satisfiability, and then remove it to add a different C. (If you added all the assertions together without push/pop, you would find a path which crash all points simultaneously, which is impossible.)

4. **Interpretation.** Once you find one or more bugs, 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 (`x`, `data`) input pair in C syntax that would crash `func()` 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?

5. **Find the smallest value of** N **that can cause out-of-bounds accesses in line 14.** Do this by asserting the value of N in addition to asserting C described in part 3. Hint: use binary search to find the minimal value of N, as it can be quite large.

6. **Submission.** In the homework write-up, include the output of running the SMT solver on your input (`sat` or `unsat`, and, if `sat`, the produced model). Explain your interpretation of the results (part 4) and include answers to the questions in part 5.

   Additionally, submit your SMT-LIB input file to the separate Gradescope assignment.