Programming Languages and Compilers
Qualifying Examination

Monday, January 30, 2017

Answer 4 of 6 questions.

GENERAL INSTRUCTIONS

1. Answer each question in a separate book.
2. Indicate on the cover of *each* book the area of the exam, your code number, and the question answered in that book. On *one* of your books list the numbers of *all* the questions answered. *Do not write your name on any answer book.*
3. Return all answer books in the folder provided. Additional answer books are available if needed.

POLICY ON MISPRINTS AND AMBIGUITIES

The Exam Committee tries to proofread the exam as carefully as possible. Nevertheless, the exam sometimes contains misprints and ambiguities. If you are convinced that a problem has been stated incorrectly, mention this to the proctor. If necessary, the proctor will contact a representative of the area to resolve problems during the *first hour* of the exam. In any case, you should indicate your interpretation of the problem in your written answer. Your interpretation should be such that the problem is nontrivial.

1 Interpolants

This question concerns the logical notion of *Craig interpolants*. You do not require prior knowledge about interpolation to answer this question.

Given two formulas A and B in first-order logic, such that $A \wedge B$ is unsatisfiable, there exists a formula I, called an interpolant, such that

1. $A \rightarrow I$ is valid (recall that a formula ϕ is valid iff all models satisfy it)
2. $I \wedge B$ is unsatisfiable;
3. $vars(I) \subseteq vars(A) \cap vars(B)$, where $vars(\phi)$ is the set of all variables that appear in ϕ.

As an example, consider the following formulas in propositional logic (i.e., all variables are Boolean):

$$A \triangleq a \wedge b$$

$$B \triangleq \neg b \wedge c$$

We know that $A \wedge B$ is unsatisfiable. An interpolant I here is b. Observe that $A \rightarrow I$ is valid, $I \wedge B$ is unsatisfiable, and I only contains variables that appear in A and B.

Part A

An alternative definition of an interpolant is as follows:

Suppose we have two formulas A and C such that $A \rightarrow C$ is valid, then there exists a formula I such that

1. $A \rightarrow I$ is valid;
2. $I \rightarrow C$ is valid;
3. $vars(I) \subseteq vars(A) \cap vars(C)$.

Prove that the two definitions of an interpolant are equivalent.

Part B

Give two formulas A and B such that $A \wedge B$ is unsatisfiable, does there always exist a unique interpolant (up to logical equivalence)? If not, provide an example of two formulas A and B and two interpolants I_1 and I_2, such that $I_1 \neq I_2$.

Part C

Suppose you are working with formulas in quantifier-free linear integer arithmetic: meaning, formulas that are Boolean combinations (conjunctions, disjunctions, negations) of linear inequalities over integers of the form: $a_1 x_1 + \ldots a_n x_n \leq c$, where a_i, c are integer constants, and x_i are integer variables.

Consider the following two formulas in quantifier-free linear integer arithmetic:

$$A \triangleq x = 2y$$
$$B \triangleq x = 2z - 1$$

Is $A \wedge B$ satisfiable? If not, does there exist an interpolant for A and B that is also in quantifier-free linear integer arithmetic? If no such interpolant exists, explain why that is the case.

```
 (1) main(int argc, char* argv[]){
 (2)    char header[2048], buf[1024],
             *cc1, *cc2, *ptr;
 (3)    int counter;
 (4)    FILE *fp;
 (5)    ...
 (6)    ptr = fgets(header, 2048, fp);
 (7)    cc1 = copy_buffer(header);
 (8)    ptr = fgets (buf, 2048, fp);
 (9)    cc2 = copy_buffer(buf);
(10) }
(11)
(12) char *copy_buffer(char *buffer){
(13)    char *copy;
(14)    copy = (char *) malloc(strlen(buffer));
(15)    strcpy(copy, buffer);
(16)    return copy;
(17) }
```

Figure 1: Example Program

2 Buffer Overruns

Part A

Languages like C, do not guarantee array-bounds checking and that means pointer arithmetic can lead to programs that are vulnerable to certain kinds of malicious attacks. Consider the program shown in Figure 1. How could a malicious user cause a buffer overrun?

Part B

Explain how a malicious user can exploit a buffer-overrun vulnerability in a program to execute the system call `system("exec /bin/sh")`? If the program is being executed as a privileged user, what are the consequences of the exploit?

Part C

Suppose you design a new data-type (called safeArray). and all array access happens through your newly designed library that provides an implementation of this new data-type. Briefly describe the design of your library and rewrite the program in Figure 1 to use your new library. Demonstrate why your previous exploit from part (b) does not work for the new rewritten program.

Part D

If one re-writes a large program to use the data-type safeArray, it will probably execute quite slowly. (i) Explain why, and, (ii) describe how static analysis can be used to optimize the re-written program.

3 Logic and Approximation in Garbage Collection

Consider imperative programs with automatic garbage collection for heap-allocated memory. Assume that programs are single-threaded unless stated otherwise. Consider two possible definitions of "garbage":

Reachability-based: Define *roots* to be values stored in either global variables or in local variables of any function activation frame on the stack. A block of heap-allocated memory is garbage if it is not transitively reachable from any root value.

Future-use-based: A block of heap-allocated memory is garbage if it is not accessed (either written to or read from) by any future computation.

Part A

Without considering implementation details, but merely the definitions themselves, how do these two definitions **relate to each other**? Is one more precise than the other? Does one imply the other, or are the two incomparable? How do the sets of garbage blocks identified by one relate to the sets of garbage blocks identified by the other? **How does your response change** depending on whether the language is type-safe (e.g., Java, Python, or C#) or not type-safe (e.g., C, C++, or assembly)?

Part B

Most garbage collectors use the first definition above. **Describe a strategy** for garbage collection based on safely approximating the second definition instead. If your approach uses both static and dynamic components, describe both and how they interact. You are welcome to reuse any off-the-shelf techniques (e.g., well-known static analyses) as part of this, and need not describe those in detail provided you identify them clearly. Instead focus on any non-standard components and/or how the pieces of your solution are put together to give the desired behavior.

Keep your strategy simple but non-trivial: a strategy that never collects anything is not acceptable here. However, you need not guarantee that your collector eventually collects everything: it is OK to assume that a reachability-based collector is also operating.

Part C

Excerpted from Wikipedia:

> A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not. In other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives.

Describe how a Bloom filter could be used in a reachability-based garbage collector. Use just one Bloom filter instance at a time. Assume that your filter has a "`reset`" operation that erases all elements from the filter, an "`insert`" operation that adds one element to the filter, and a "`contains`" test that checks for one element's membership in the filter (subject to the probabilistic limitations described above).

Part D

Again from Wikipedia:

> An empty Bloom filter is a bit array of m bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions with a uniform random distribution.
>
> To add an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.
>
> To query for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions. If any of the bits at these positions is 0, the element is definitely not in the set—if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive. In a simple Bloom filter, there is no way to distinguish between the two cases.

If our program is multi-threaded, we might reduce contention by using one Bloom filter per thread, then combining filters later. It can be useful, then, to know how to perform set-like operations on Bloom filters and how these affect the probabilistic guarantees.

Given two Bloom filters A and B, the bitwise OR of A's and B's bits is a Bloom filter representing the set union of A and B. Furthermore, this union operation is lossless in the sense that it is identical to the filter one would get by iteratively adding each element in the union of exact representations of A and B.

Suppose instead we compute the bitwise AND of A's and B's bits, and treat this as representing the set intersection of A and B. **Is this operation also lossless?** If yes, justify why this is so. If no, then how does this change the likelihood of false positives or false negatives as compared with a Bloom filter built from scratch by iteratively adding each element in the intersection of exact representations of A and B?

4 Predicate Abstraction

A predicate-abstraction domain over Boolean variables B involves the set A of all *Boolean assignments* over B. That is, $A \stackrel{\text{def}}{=} B \to \text{Bool}$. (Note that A is isomorphic to the power set of B, i.e., $\mathcal{P}(\mathcal{B})$.) For instance, suppose that B is $\{b_1, b_2\}$. Letting $(v_1 v_2)$ denote the Boolean assignment in which b_1 has value v_1 and b_2 has value v_2, the set A is $\{(00), (01), (10), (11)\}$.

When using predicate abstraction to analyze a particular concrete program, each Boolean variable represents some predicate on the states of the concrete program (e.g., "$x < 100$" or "$x = y + z$"). The abstract transformer associated with a statement *st* is the binary relation $R_{st} \subseteq A \times A$ that represents the effect of *st* on Boolean assignments over B.

To implement a predicate-abstraction-based analyzer, we need to implement appropriate abstract transformers (also known as "transition relations") that transform sets of Boolean assignments to sets of Boolean assignments. The set of all possible transition relations *TR* is $\mathcal{P}(A \times A)$. Each transition relation $R \subseteq A \times A$ in *TR* consists of a set of tuples; each tuple (a, a') represents the possibility of a pre-state Boolean assignment a being mapped to a post-state Boolean assignment a'.

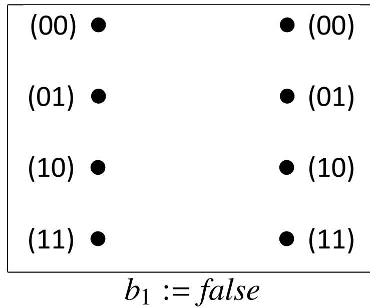
Statements are of three types: *assignment* statements (e.g., "$b_2 := b_1$"), *assume* statements (e.g., "assume $b_1 = \mathit{false}$"), or *havoc* statements (e.g., "havoc b_2").

- The semantics of "assume φ" is that if φ holds for a Boolean assignment a, then execution continues (i.e., the corresponding post-state is a); however, if φ does not hold for a, then a is blocked (i.e., a has no corresponding post-state).

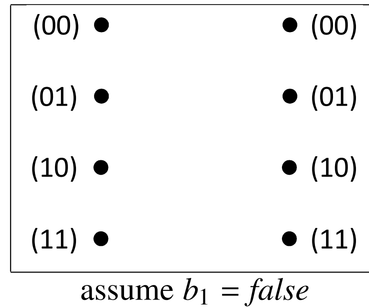
- The semantics of "havoc b_j" is that pre-state a can transition to the post-states in which the value of b_j is either true or false, but the values of all other variables are left unchanged.

Part A

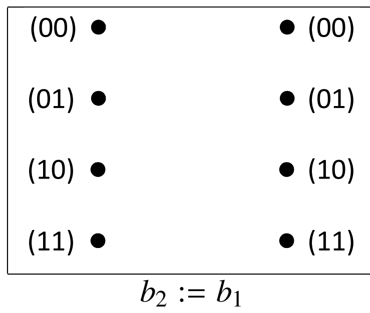
Consider the abstract transformers for "$b_1 := \mathit{false}$," "assume $b_1 = \mathit{false}$," "$b_2 := b_1$," "$b_1 := b_2$,", and "havoc b_2." Draw the graphs that represent the transition relations of these five statements.



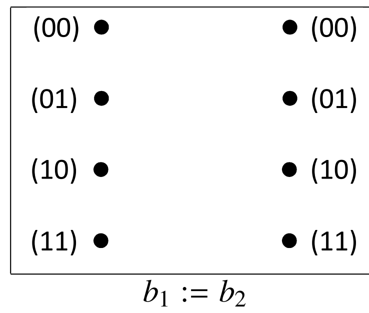
$b_1 := \mathit{false}$



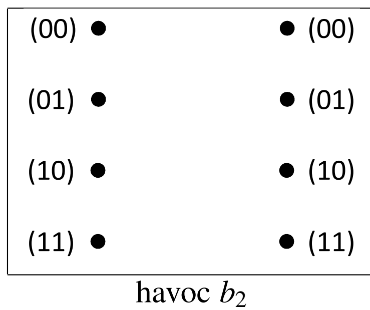
assume $b_1 = \mathit{false}$



$b_2 := b_1$



$b_1 := b_2$



havoc b_2

Part B

Explain how to encode a predicate-abstraction problem using Boolean-valued vectors and matrices, where (i) a vector encodes a *set of Boolean assignments* (e.g., $\{(00),(10)\}$), and (ii) a Boolean matrix encodes a *transition relation*. In particular,

1. Explain what a given entry V_i in a vector V represents. Say what vectors represent the following four sets of Boolean assignments:
   (a) $\{(00)\}$
   (b) $\{(00),(10)\}$
   (c) $\{(01),(10),(11)\}$
   (d) $\{(00),(01),(10),(11)\}$
2. Explain what a given entry $M_{i,j}$ in a matrix M represents.
3. Pick two of the transformers from Part (a)—say which ones you pick—and give the matrices for the transformers. Explain how the matrices relate to the graphs.

Part C

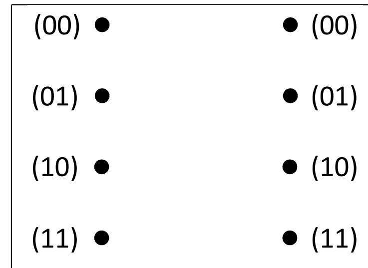
Explain how the following operations are performed on the matrix representations:

1. Transition-relation composition (where the composition operator ";" denotes the operation $(R_1;R_2)(V) = R_2(R_1(V))$).
2. Join of transition relations (where join is denoted by $\sqcup$).

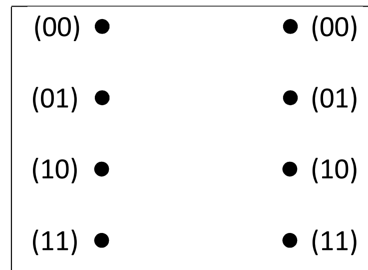
Part D

Draw the graphs that represent the results of the four sequences of operations indicated below:

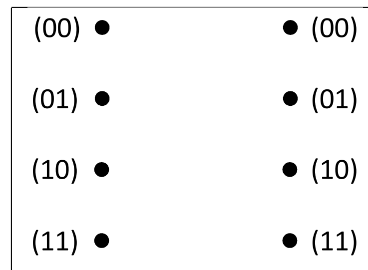
$(b_2 := b_1); (b_1 := \mathit{false})$



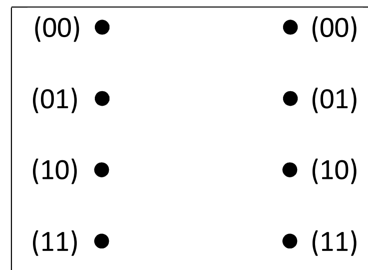
$(b_2 := b_1); (\text{assume } b_1 = \mathit{false}); (b_1 := b_2)$



$(b_1 := b_2); (\text{havoc } b_2)$



$(b_2 := b_1) \sqcup (\text{assume } b_1 = \mathit{false})$



5 Fixed-Point Combinators

This question concerns fixed-point combinators and methods for finding fixed points in λ-calculus. Parts (a) and (b) concern the following theorem for characterizing fixed-point combinators themselves as fixed points:

$$\text{Let } G = \lambda y.\lambda f.f(yf)\text{. Then } M \text{ is a fixed point combinator if and only if } M = GM. \qquad (1)$$

(Note: Recall that the following λ-calculus transformation is called the η-reduction rule:

$$(\lambda x.Mx) \to_\eta M,$$

where x does not occur as one of the free variables of M. You are allowed to use η-reduction in this question.)

Part A

Use (1) to show that $Y =_{\text{df}} \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$ is a fixed-point combinator.

Part B

Prove (1). (Note that (1) involves an "if and only if"; consequently, your proof should have two parts.)

Part C

The fixed-point combinator discussed in Part (a) allows us to find a λ-term g that satisfies a single recursive equation over λ-terms of the form $g = \ldots g \ldots g \ldots$

Suppose that we are presented with a collection of k mutually recursive equations:

$$\begin{array}{rcl} g_1 & = & \ldots g_1 \ldots g_k \ldots \\ & \vdots & \\ g_k & = & \ldots g_1 \ldots g_k \ldots \end{array}$$

Explain how to solve for $g_1, \ldots, g_k$.

6 Partial deadness

In static analysis we are often interested in detecting whether a variable becomes dead (i.e., it is not used from a certain point on). In this problem, we investigate a variant of deadness. We say that a variable is *partially dead* at a program point if there exists *at least* one path on which there is *no use* of that variable before its redefinition and there is *at least* one path on which there is *a use* of that variable before its redefinition. In this problem you are asked to find techniques for detecting such variables and remove partial deadness.

Part A

Show an example of a control flow graph (CFG) that shows at least an instance of partial deadness and at least an instance of total deadness.

Part B

Describe a dataflow analysis to detect partially dead variables. (For purposes of this part, you can assume that you are dealing with a simple while-loop language with integer-valued variables only - in particular, no pointer variables.)

Part C

Describe a code-motion framework that removes partial deadness (as a best effort) without introducing any redundancy. *Hint:* Move complete assignments – e.g., you could move `t1 = x + y` to remove partial deadness with respect to the use of `t1`.

Part D

For a language with pointer variables, the statement of the partial-deadness property becomes slightly different (if one is trying to be precise). Say how the partial-deadness property should be restated when dealing with a language with pointers.