Programming Languages and Compilers
Qualifying Examination

Fall 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.

Question 1: Points-to Analysis

This question is about points-to analysis for a language like C that allows pointers, pointers-to-pointers, etc. The goal of the analysis is to discover, for each variable x, which variables x might point to.

Assume that our language includes assignment statements, if-then-else statements, and while loops (but no function calls, no pointer arithmetic, and no heap-allocated storage). Also assume that the assignment statements have been normalized so that there is at most one pointer dereference per statement (e.g., the following statements are okay: `x = &y; x = y; x = *y; *x = y;`, but `*x = *y;` and `**x = y` are not).

Part (a):

Describe a method to perform *flow-insensitive* points-to analysis for this language. (Your description can be in any of several formalisms: as a "standard" dataflow analysis defined by supplying a lattice of values, plus the transfer function for each edge in the control-flow graph; or as a Horn-clause program. You may use whatever formalism you find most convenient.)

Part (b):

Now define a *flow-sensitive* points-to analysis for the language by providing the following:

1. The dataflow functions for each kind of edge in the control-flow graph (i.e., the four different kinds of assignments, and the outgoing true-edges and false-edges at a branch node), as well as the initial value to use at the program-enter node.

2. The join operator used to combine dataflow facts at convergence points in the control-flow graph.

Part (c):

Why is it interesting to know whether the dataflow functions for a flow-sensitive analysis are distributive?

Are the dataflow functions that you wrote for Part (b) all distributive? (Justify your answer.)

Question 2: Abstract Interpretation

This question concerns abstract interpretation via *operator-by-operator reinterpretation* (henceforth, "reinterpretation" for short).

Part (a):

One way to perform reinterpretation is via an *interpretive strategy*. That is, to reinterpret the expression "$(-a + 3) * 5$," an interpretive abstract interpreter would have two arguments: (i) the root r of the expression's abstract syntax tree (AST), and (ii) an abstract environment $\sigma^\sharp$, (in this case to hold the abstract value for variable a).

Let Plus, Times, Id, and Const be the operators of the tree-grammar for ASTs; $+^\sharp$, and $*^\sharp$ be the abstract operations for addition and multiplication; and $\alpha(c)$ be the abstraction operation that maps a concrete value c to the abstract value that represents c in the abstract domain.

The interpretive abstract interpreter would look much like a standard interpreter:

$$
\begin{array}{l}
\text{interp}^\sharp(r, \sigma^\sharp)\ \{ \\
\quad \text{with } \mathit{op}(r)\ \{ \\
\quad\quad \text{Plus}(e_1, e_2) : \\
\quad\quad\quad \text{let } v_1 = \text{interp}^\sharp(e_1, \sigma^\sharp) \text{ and } v_2 = \text{interp}^\sharp(e_2, \sigma^\sharp) \text{ in} \\
\quad\quad\quad\quad v_1 +^\sharp v_2, \\
\quad\quad \text{Times}(e_1, e_2) : \\
\quad\quad\quad \text{let } v_1 = \text{interp}^\sharp(e_1, \sigma^\sharp) \text{ and } v_2 = \text{interp}^\sharp(e_2, \sigma^\sharp) \text{ in} \\
\quad\quad\quad\quad v_1 *^\sharp v_2, \\
\quad\quad \text{Id}(i) : \\
\quad\quad\quad \sigma^\sharp(i), \\
\quad\quad \text{Const}(c) : \\
\quad\quad\quad \alpha(c) \\
\quad \} \\
\}
\end{array}
$$

1. Extend interp$^\sharp$ to handle conditional expressions of the form "$b\ ?\ e_1 : e_2$" (represented by the abstract-syntax operator $Cond(b, e_1, e_2)$).

   Your interpreter will also need to support a few simple numeric conditions—say $e_1 < e_2$ (LessThan(e_1, e_2)), $e_1 \leq e_2$ (LessEq(e_1, e_2)), and $!b$ (Not(b))—and will abstract Boolean values with *trivalues*: $\{\texttt{tt}, \texttt{ff}, \texttt{maybe}\}$, standing for true, false, and unknown, respectively. Note that, for a given conditional expression and specific value of $\sigma^\sharp$, the abstract interpreter may have to interpret just the true branch, just the false branch, or both branches.

2. Using the domain of signs, which has the values $\{\mathit{neg}, \mathit{zero}, \mathit{pos}, \mathit{nonneg}, \mathit{nonpos}, \mathit{unknown}\}$—with the obvious meanings, give a trace of the execution of your version of interp$^\sharp$ on the expression "$(a < 0)\ ?\ (-1 * a) : ((a + 3) * 5)$," where $\sigma^\sharp = [a \mapsto \mathit{unknown}]$, and show that it produces a sound answer. (Sound answers are *pos*, *nonneg*, and *unknown*.)

Part (b):

An alternative method for implementing an abstract interpreter is to use a *compiled strategy*. In this approach, an expression to be abstractly interpreted is *translated* to a code sequence that performs the appropriate computation. For instance, assuming that the code sequence has access to an abstract environment $\sigma^\sharp$, the expression "$(a + 3) * 5$" might be translated to

$$\begin{array}{rcl} t_1 & = & \sigma^\sharp(a); \\ t_2 & = & t_1 +^\sharp \alpha(3) \\ t_3 & = & t_2 *^\sharp \alpha(5) \end{array}$$

Now suppose that we want to extend this approach to handle conditional expressions. As with the interpreted method, for a given conditional expression and specific value of $\sigma^\sharp$, it may be necessary to evaluate just the true branch, just the false branch, or both branches.

One issue that comes up when designing a compiled abstract-interpretation strategy is that a naive method for compiling (nested) conditional expressions can lead to an exponential explosion in the code size. That is, the generated code can be exponentially larger than the original expression, where the value of the exponent is the greatest depth of nesting of conditional expressions.

1. (a) Describe a naive method for compiling nested conditional expressions that causes exponential explosion.

   (b) Illustrate your method—and its exponential explosion—on the following example:

$$\begin{array}{rcl} c_0 & ? & \left(\; c_1 \; ? \; e_{1,1} : e_{1,2} \; \right) \\ & : & \left(\; c_2 \; ? \; e_{2,1} : e_{2,2} \; \right) \end{array}$$

   (c) Explain why the method causes exponential explosion.

2. (a) Describe a method for compiling nested conditional expressions that leads to code that is linear in the size of the original expression.

   (b) Illustrate your method on the example from 1b.

   (c) Explain how the method avoids exponential explosion and why it produces linear-size code.

Question 3: Compiling Switch Statements

This question concerns compilation strategies for `switch` statements such as are found in languages like C, C++, and Java. Assume that the cases within a switch must correspond to integer or character literals. For example, none of the following are allowed:

```
case x:
case "text":
case 1.5:
```

Part (a): Compilation Strategies

Describe three distinct strategies for compiling `switch` statements to native code on mainstream instruction set architectures such as $x86\_64$. Compare and contrast the advantages and disadvantages of each strategy either in isolation or as compared with the others.

You do not need to give specific assembly instructions. It is sufficient to describe your strategy in broad terms that would have a straightforward interpretation in any modern, mainstream ISA. For example, you may refer to conditional branching without needing to know the exact mnemonic for any specific $x86\_64$ conditional-branch instruction. You may also describe your strategies at a higher level, such as via source-to-source rewrites combined with descriptive prose, without ever getting into assembly-language details. However, the compilation strategies for your rewritten forms must be compilable in their own right without using `switch` statements: rewriting a `switch` to something else that also uses `switch` is not a complete answer.

Part (b): Dynamic Feedback

Suppose that your compiler now has the option of tuning its compilation strategies using dynamic feedback from earlier runs. How might feedback-directed optimization be incorporated into the compilation strategies that you described above?

Question 4: Security

Part A: Suppose that there is a server S that interacts with remote clients. Further assume that the code in S that parses the requests from the client has a buffer overrun. How can a malicious client C_M exploit this vulnerability and install a malicious program on the host that the server is executing on? Please provide details of the exploit.
Part B: *Taint-analysis* is one of the techniques to protect against such remote exploits. In taint-analysis, variables that can be "affected" remotely are labeled as *tainted* (e.g., a buffer that holds a network packet). How can taint-analysis be used to protect against such remote exploits? Please be specific and use the exploit that you describe in part-A to clarify your answer.
Part C: Describe a dynamic-analysis system that performs taint-analysis described above. What are the performance bottlenecks in the system that you describe? Describe techniques to alleviate these performance bottlenecks.

Question 5: Havoc

We assume a simple imperative language in which all variables are (positive) integer valued. We will reason about the programming construct `havoc`, which sets a variable to a non-deterministic value—i.e., when we execute `havoc X` the value of `X` after the execution can be **any 16-bit positive integer**—and `havoc-bin`, which sets a variable to a binary non-deterministic value—i.e., when we execute `havoc-bin X` the value of `X` after the execution can be **either 0 or 1**.

Part (a):

As a warm-up you will have to define the operational semantics of the new operators. Recall that the small-step semantics of the assignment-statement operator is defined as follows:

$$\frac{(st, a_1 \Downarrow_e n) \wedge \forall \mathtt{Y} \neq \mathtt{X}.st(\mathtt{Y}) = st'(\mathtt{Y}) \wedge st'(\mathtt{X}) = n}{(\mathtt{X} ::= a_1)/st \Downarrow_s st'}$$

Here, a state st is a total function from the set of variables to integer values and $\Downarrow_e$ denotes an arithmetic expression evaluation and $\Downarrow_s$ denotes a small step in the semantics. Using the same notation as above, define the semantics of `havoc` and `havoc-bin`.

Part (b):

Now that we have defined the semantics of `havoc` we also want to reason about it in a formal fashion. Recall that the Hoare-logic rule for the assignment-statement operator is axiomatized as follows:

$$\frac{}{\{Q[e_1/\mathtt{X}]\}\mathtt{X} := e_1\{Q\}}$$

Here, $Q[e_1/\mathtt{X}]$ denotes the result of substituting every occurrence of `X` in Q with the expression e_1. Using the same notation as above, define a Hoare logic rules for `havoc` and `havoc-bin`.

Part (c):

Hoare triples are closely related to the notion of weakest precondition. The weakest-precondition $wp(S, Q)$of a program statement S with respect to a postcondition (a predicate) Q is the largest predicate P that ensures that the Hoare triple $\{P\}S\{Q\}$ holds.

Recall that `havoc` non-deterministically sets a variable to a **any 16-bit positive integer** and `havoc-bin` non-deterministically sets a variable to **either 0 or 1**. Provide the missing weakest preconditions for the following triples:

- $\{$ $\}$`havoc X`$\{X \geq 0\}$
- $\{$ $\}$`havoc X`$\{X = 12\}$
- $\{$ $\}$`havoc-bin X`$\{X < 34\}$
- $\{$ $\}$`havoc-bin X`$\{X = Y\}$

Part (d):

Nondeterminism is a powerful construct that can help us write programs with complex behavior.

1. Is it possible to simulate non-determinism using existing constructs of the classic IMP language—i.e., assignments, conditionals, and loops? Give an informal (but precise) argument for your answer.

2. Can `havoc` and `havoc-bin` simulate each other—i.e., can you write an implementation of `havoc` using `havoc-bin` and vice-versa? If so, show such implementations and argue for their equivalence; if not, explain why.

Question 6: Extending Static Analysis to a Probabilistic Setting

Let $\mathcal{L}$ be a very simple programming language where a program $P \in \mathcal{L}$ is comprised of assignment, conditional, and while-loop statements. Assume also that all variables are either Boolean or real-valued, and that all programs of interest are well-typed and exhibit no runtime errors (e.g., divisions by zero).

For a program P, we assume it has a set of input variables V_i and a set of output variables V_r. For the rest of this question, imagine that we have at our disposal an almighty static analyzer $\mathcal{A}$ that, given $P \in \mathcal{L}$, returns a set S of all states reachable by executing P from any possible input. A state $s \in S$ is considered to be a map from every variable in P to a value. Given variable x, we use $s[x]$ to denote the value of x in s.

Example 1 For illustration, consider the following example:

```
def p(x)
  y = x + 1
  return y
```

Given the above program, the static analyzer returns the set

$$\{s \mid s[x] = c, s[y] = c + 1, c \in \mathbb{R}\}$$

In other words, it returns the set of all states s where $y = x + 1$.

Example 2 We also assume that our language $\mathcal{L}$ allows non-deterministic choice, which is denoted by `if (*) ... else ...`, where the program can non-deterministically execute either branch of the conditional statement. Consider, for example, the following simple program:

```
def p()
  if (*)
    y = 1
  else
    y = 2
  return y
```

Given the above program, the static analyzer returns the set

$$\{s \mid s[y] \in \{1, 2\}\}$$

Part (a): Using $\mathcal{A}$ for probabilistic reasoning

Suppose that we decide to extend our language $\mathcal{L}$ into a new language $\mathcal{L}^{\sim}$ with random assignments of the form

```
x ~ bern(c)
```

where Boolean variable `x` is assigned true with probability `c` and false with probability `1-c`, where `c` is a constant in $[0, 1]$. (`bern` stands for a Bernoulli distribution.)

Given a program $P \in \mathcal{L}^{\sim}$, we are typically interested in the probability of the program returning a specific set of values. For example, consider the following probabilistic program:

```
def p()
  x ~ bern(0.5)
  y ~ bern(0.5)
  z =  x && y
  return z
```

The probability that `p` returns true is 0.25, as both `x` and `y` have to be true.

Your task in this question is as follows: Given a program $P \in \mathcal{L}^{\sim}$ and the static analyzer $\mathcal{A}$, you are to use $\mathcal{A}$ to compute the probability that P returns a value in some set X. Note that $\mathcal{A}$ only accepts programs in $\mathcal{L}$, so you have to somehow transform $P \in \mathcal{L}^{\sim}$ into a new program in $\mathcal{L}$ in order to be able to use $\mathcal{A}$. Once you have the output set S of $\mathcal{A}$, you may apply any mathematical operation on S to extract your desired result.

You should assume that P is always terminating, has no non-deterministic conditionals, has no input variables (like the above example), and only manipulates Boolean variables. **Hint:** your transformation of P may introduce non-determinism and real-valued variables.

Part (b): Discovering maximizing inputs

In the first part of the question, we assumed that P has no input variables. In this part, we will assume that P has input variables. Our goal is to find a value for the input variables that maximizes the probability of returning some output value. Consider the following example:

```
def p(x)
  if (x)
    y ~ bern(0.5)
  else
    y ~ bern(0.9)
  return y
```

Suppose we want to maximize the probability that `p` returns the value true. To do so, we have to set the input variable `x` to false in order to force the program down the else branch of `p`, which has a higher probability of setting `y` to true.

Describe how you would extend your technique from Part (a) to this setting.