Programming Languages and Compilers
Qualifying Examination

Monday, January 31, 2011

This exam asked students to answer 4 out of 6 questions. This document contains the 5 questions that were answered.

Question 1 (Automata).

A *non-deterministic, bottom-up, finite-state tree automaton* (which will be abbreviated as FSTA) is a formalism for recognizing (or "accepting") members of a language of trees. An FSTA $A = (Q, Q^F, \Sigma, \delta)$ has a set of *states* Q, a set of final states $Q^F \subseteq Q$, a *ranked alphabet* Σ, and a transition relation δ.

A ranked alphabet means that each symbol has an *arity*, which indicates how many children it has. We will denote, e.g., a binary (arity-2) symbol *foo* by *foo*^{2}. Thus, if T_1 and T_2 are two trees, *foo*$^2(T_1, T_2)$ is another tree; it has the symbol *foo*^{2} at its root.

The transition relation δ consists of rules of the form

$$q(f^n) \leftarrow f^n(q_1, \ldots, q_n),$$

where $q, q_1, \ldots, q_n \in Q$ and f^n is an n-ary symbol. We allow A to be non-deterministic; that is, one can have multiple result states (i.e., left-hand-side states) for a given combination of symbol and child states:

$$\begin{array}{rcl} q(f^n) & \leftarrow & f^n(q_1, \ldots, q_n) \\ q'(f^n) & \leftarrow & f^n(q_1, \ldots, q_n) \end{array}$$

An FSTA A accepts a language of trees $L(A)$. For a given tree T, T is accepted or rejected depending on the outcome of the possible *runs* of A over T. A run labels each leaf of T with a state, and then moves upwards to successively label each node of T with a state, using the rules of δ. That is, if we have the rule

$$q(f^n) \leftarrow f^n(q_1, \ldots, q_n)$$

in δ and there is a subtree S whose root symbol is f^n and whose n children are labeled with $q_1, \ldots, q_n$, respectively, then the root of S can be labeled with q.

An *accepting run* is one that labels the root of the tree with a state in Q^F. Because we allow A to be non-deterministic, only *one* of the possible runs of A over T needs to be an accepting run for T to be *accepted* (i.e., for $T \in L(A)$ to hold).

An FSTA has no initial state, but the rules for 0-ary symbols cause certain states to act as initial states at a tree's various leaves. For instance, suppose that we have the rule

$$q_{17}(a^0) \leftarrow a^0()$$

Then if T has any instance of a^0 as a leaf, that leaf can be labeled with q_{17}, and serves as one of the "initial" states for runs of A over T. Note that we are permitted to have multiple rules for a 0-ary symbol and these can be used at different leaves in a given run:

$$\begin{array}{rcl} q_{15}(a^0) & \leftarrow & a^0() \\ q_{17}(a^0) & \leftarrow & a^0() \end{array}$$

Example. Consider the FSTA A_{exp} defined as follows:

$$A_{exp} = (\{q_{int}, q_{\text{float}}, q_{\text{error}}\}, \{q_{int}, q_{\text{float}}\}, \{plus^2, a^0, m^0, x^0\},$$
$$\left\{ \begin{array}{rcl@{\qquad}rcl} q_{int}(a^0) & \leftarrow & a^0() & q_{\text{error}}(plus^2) & \leftarrow & plus^2(q_{\text{float}}, q_{int}) \\ q_{int}(m^0) & \leftarrow & m^0() & q_{\text{error}}(plus^2) & \leftarrow & plus^2(q_{\text{error}}, q_{int}) \\ q_{\text{float}}(m^0) & \leftarrow & m^0() & q_{\text{error}}(plus^2) & \leftarrow & plus^2(q_{\text{error}}, q_{\text{float}}) \\ q_{\text{float}}(x^0) & \leftarrow & x^0() & q_{\text{error}}(plus^2) & \leftarrow & plus^2(q_{int}, q_{\text{error}}) \\ q_{int}(plus^2) & \leftarrow & plus^2(q_{int}, q_{int}) & q_{\text{error}}(plus^2) & \leftarrow & plus^2(q_{\text{float}}, q_{\text{error}}) \\ q_{\text{float}}(plus^2) & \leftarrow & plus^2(q_{\text{float}}, q_{\text{float}}) & q_{\text{error}}(plus^2) & \leftarrow & plus^2(q_{\text{error}}, q_{\text{error}}) \\ q_{\text{error}}(plus^2) & \leftarrow & plus^2(q_{int}, q_{\text{float}}) & & & \end{array} \right\}).$$

Let T_1 and T_2 be two trees defined as follows:

$$\begin{array}{rcl} T_1 & = & plus^2(plus^2(m^0(), a^0()), a^0()) \\ T_2 & = & plus^2(plus^2(a^0(), a^0()), x^0()) \end{array}$$

Note that there is both an accepting run for T_1, namely,

$$\begin{array}{rl} & plus^2(plus^2(m^0(), a^0()), a^0()) \\ \Rightarrow & plus^2(plus^2(q_{int}(m^0), q_{int}(a^0)), q_{int}(a^0)) \\ \Rightarrow & plus^2(q_{int}(plus^2(m^0(), a^0())), q_{int}(a^0)) \\ \Rightarrow & q_{int}(plus^2(plus^2(m^0(), a^0()), a^0())) \end{array}$$

and a non-accepting run for T_1,

$$\begin{array}{rl} & plus^2(plus^2(m^0(), a^0()), a^0()) \\ \Rightarrow & plus^2(plus^2(q_{\text{float}}(m^0), q_{int}(a^0)), q_{int}(a^0)) \\ \Rightarrow & plus^2(q_{\text{error}}(plus^2(m^0(), a^0())), q_{int}(a^0)) \\ \Rightarrow & q_{\text{error}}(plus^2(plus^2(m^0(), a^0()), a^0())) \end{array}$$

In contrast, there is only a non-accepting run for T_2, namely,

$$\begin{array}{rl} & plus^2(plus^2(a^0(), a^0()), x^0()) \\ \Rightarrow & plus^2(plus^2(q_{int}(a^0), q_{int}(a^0)), q_{\text{float}}(x^0)) \\ \Rightarrow & plus^2(q_{int}(plus^2(a^0(), a^0())), q_{\text{float}}(x^0)) \\ \Rightarrow & q_{\text{error}}(plus^2(plus^2(a^0(), a^0()), x^0)) \end{array}$$

Consequently, $T_1 \in L(A_{exp})$ but $T_2 \notin L(A_{exp})$. $\square$

Abbreviations:

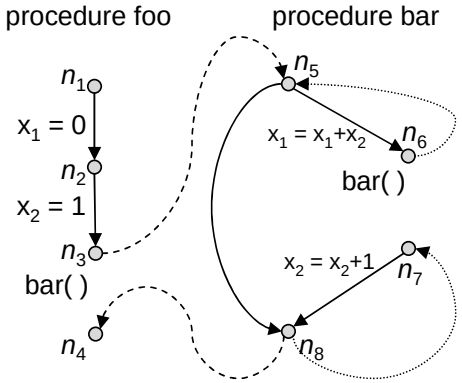
- You may drop superscripts on alphabet symbols.
- Although we wrote out all of the possible transitions involving q_{error}, it would have been convenient to treat q_{error} as a "stuck" state—in which case, in the set of rules for A_{exp} we would have omitted the last two rules in the first column and all the rules in the second column. Such rules would be implicit: an occurrence of q_{error} in any child of an arity-k symbol results in the symbol being labeled with q_{error}.

Part (a) (Relationship to String Automata):
Explain how ordinary non-deterministic finite-state (string) automata are a degenerate case of FSTAs.

FSTAs are useful in dataflow-analysis and model-checking problems because they can be used to describe the matched call-and-return structure of paths in multi-procedure programs. Parts (b) and (c) concern how to define an FSTA to specify a tree-language that captures the matched paths specific to a particular program, namely, the one shown below. (The "if(*)" denotes a non-deterministic branch.)

```
void foo() {
   n1: x1 = 0;
   n2: x2 = 1;
   n3: bar();
   n4: ;
}
```

```
void bar() {
   n5: if (*) {
          x1 = x1 + x2;
      n6: bar();
      n7: x2 = x2+1;
   }
   n8: ;
}
```



The FSTA you will define should accept a language, each tree of which represents a properly matched path from n_1 to n_4. For each properly matched path ρ from n_1 to n_4, the FSTA should accept a tree that represents ρ. The FSTA should reject trees that either (i) do not represent a path, or (ii) represent a path that violates matched call-and-return structure in the graph given above.

The alphabet Σ consists of three kinds of symbols:

1. Nine 0-ary symbols for the nine edges in the graph given above:
$$\{e^0_{n_1\to n_2}, e^0_{n_2\to n_3}, e^0_{n_3\to n_5}, e^0_{n_5\to n_8}, e^0_{n_5\to n_6}, e^0_{n_6\to n_5}, e^0_{n_7\to n_8}, e^0_{n_8\to n_7}, e^0_{n_8\to n_4}\}.$$
2. Sixty-four binary symbols for possible start/end positions in a sub-path:
$$\{p^2_{n_i\to n_j} \mid 1 \le i, j \le 8\}.$$
3. Sixty-four ternary symbols for subtrees that represent possible matched call-and-return sub-paths: $\{c^3_{n_i\to n_j} \mid 1 \le i, j \le 8\}$. (The symbols are ternary so that the three children can represent a call-edge from caller-to-callee, an edge or a matched path from the entry node to the exit node of the callee, and a return-edge from callee-to-caller.)

Part (b) (Representing Paths with Trees):
The idea is that the frontier of each tree (i.e., its sequence of leaves in left-to-right order) represents a candidate path. Draw the three trees that represent the following paths:

1. $[n1 \to n2, n2 \to n3, n3 \to n5, n5 \to n8, n8 \to n4]$
2. $[n1 \to n2, n2 \to n3, n3 \to n5, n5 \to n6, n6 \to n5, n5 \to n6, n6 \to n5, n5 \to n8, n8 \to n7, n7 \to n8, n8 \to n7, n7 \to n8, n8 \to n4]$
3. $[n1 \to n2, n2 \to n3, n3 \to n5, n5 \to n8, n8 \to n7]$

(Note: the first two trees should be accepted by the FSTA that you will define in Part (c); the third tree should be rejected by the FSTA from Part (c).)

Part (c) (Identifying Matched Call-and-Return Structure):
The alphabet Σ of the FSTA has been defined above. The set of states Q of the FSTA consists of a stuck state, q_{error}, together with 64 states that are indexed by a pair of node names: $Q = \{q_{\text{error}}\} \cup \{q_{n_i \to n_j} \mid 1 \leq i, j \leq 8\}$. The set of final states is defined as follows: $Q^F = \{q_{n_1 \to n_4}\}$.

Using Q, Q^F, and Σ as defined above, give the definition of an FSTA that accepts the language of trees that represent all properly matched paths from n_1 to n_4. The intention is that state $q_{n_i \to n_j}$ only arises in a run when there exists a matched path from n_i to n_j.

(There are 65 different states and 137 alphabet symbols. We do not expect you to write out the full transition relation; however, it should be clear from your answer what the essential features are and what the intended pattern is.)

Explain why your FSTA accepts the first two trees from your answer to Part (b), and why it rejects the third tree from Part (b).

Part (d) (Checking Emptiness):
Given an FSTA $A = (Q, \Sigma, \delta, q^0, Q^F)$, give an algorithm for determining whether $L(A) = \emptyset$.

Question 2 (Lambda Calculus Evaluation Strategies).

Recall that lambda expressions can be reduced either using normal order reduction (NOR)—i.e., reduce the leftmost-outermost redex—or using applicative-order reduction (AOR)—i.e., reduce the leftmost-innermost redex. Two reduction strategies, S_1 and S_2 are considered to be equivalent iff for every lambda expression e, either both S_1 and S_2 reduce e to normal form, or neither does (i.e., neither terminates).

Part (a):
What are the advantages of NOR over AOR and vice versa? Give examples to illustrate your answers.

Part (b):
Is the strategy "reduce the rightmost-outermost redex" equivalent to NOR? If yes, briefly justify your answer. If no, give an example of a lambda expression for which one strategy leads to a normal form while the other strategy fails to terminate.

Part (c):
Is the strategy "reduce the rightmost-innermost redex" equivalent to AOR? If yes, briefly justify your answer. If no, give an example of a lambda expression for which one strategy leads to a normal form while the other strategy fails to terminate.

Question 4 (Security).

Part (a) (Provoking a Buffer Overrun):
Languages like C that do not guarantee array-bounds checking and that allow 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) (Exploiting a Buffer Overrun):
Explain how a malicious user can exploit a buffer-overrun vulnerability in a program.

Part (c) (Buffer-Overrun Analysis):
We will sketch a static-analysis technique to detect buffer overruns. Each buffer *buf* (variable of type `char *`) is associated with two range-valued variables $R_{len}(buf)$ and $R_{alloc}(buf)$ (one for the length and other for the allocated space). A program variable i (of type `int`) is associated with a single range-valued variable $R(i)$ (representing the possible values of i). Intuitively, if $R_{len}(buf) = (n, m)$, then the minimum and maximum length of the buffer *buf* are n and m, respectively. Similarly, $R_{alloc}(buf) = (n, m)$ indicates that the minimum and maximum allocated space for the buffer *buf* are n and m, respectively. Zero or more subset constraints on range-valued variables are generated for each statement in the program. For example, consider the following statement:

```
strcpy(a,b)
```

Since b is copied into a, the following constraint is generated:

$$R_{len}(b) \subseteq R_{len}(a)$$

Show the range constraints generated for the program given in Figure 1.

Note: You will have to use sets of constraints that model the library functions `fgets` and `strcpy`. We are assuming you know the semantics of `strcpy`. Descriptions of `strlen` and `fgets` are given below:

```
strlen()
  // Returns the number of characters up to, but not including, the nearest '\0'

char *fgets(char *s, int size, FILE *stream);
  // Reads in at most size-1 characters from stream and stores them in the buffer pointed to by s.
  // Reading stops after an EOF or a newline.  If a newline is read, it is stored in the buffer.
  // A '\0' is stored after the last character in the buffer.
```

Part (d) (Identifying Overruns using Range Constraints):
Solving a system S of range constraints means finding the "tightest" possible ranges that respect all of the constraints in S. For example, consider the following system S_1 of range constraints:

$$\begin{array}{rcl} (4,4) & \subseteq & R(a) \\ (8,8) & \subseteq & R(b) \\ R(b) & \subseteq & R(a) \end{array}$$

```
 (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, 1024, 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.

The following assignment of ranges is the "tightest" that respects constraints in S_1:

$$\begin{aligned} R(a) &= (4, 8) \\ R(b) &= (8, 8) \end{aligned}$$

Notice that the following assignment of ranges also respects the constraints in S_1, but does not assign the "tightest" possible ranges.

$$\begin{aligned} R(a) &= (1, 9) \\ R(b) &= (5, 8) \end{aligned}$$

Suppose that there is procedure P for solving a system of range constraints, i.e., procedure P returns the "tightest" possible ranges that respect the constraints in a system S. How will you use procedure P to discover buffer overruns?

Part (e) (Defining a Graph Algorithm for Solving Range Constraints):
Consider the range constraints from Part (c). Give a graph algorithm to solve the range constrains generated in Part (c). You need only explain your algorithm with respect to the specific set of constraints generated in Part (c).

Question 5 (Registers).

A C programmer can declare local variables to be *register* variables. This kind of declaration tells the compiler to try to keep those variables in registers rather than in the function's activation record (on the stack).

Part (a):
Suppose that there are N register variables in a function but fewer than N registers available for allocation. What can the compiler do to determine whether all N variables can be kept in registers?

Part (b):
Normally, if a variable is in a register, the value in that register must be saved before every function call and restored after the call. If the number of variables in registers is large, this can make function calls quite expensive. What can a compiler do to avoid unnecessary saving and restoring of registers across calls?

Part (c):
Even if a register's value must be saved and restored across a function call, it may be possible to improve the code (by reducing code size and/or execution time) by placing the save/restore code somewhere other than immediately before/after the call. Give some examples of this and explain how the compiler can determine where to place the save/restore code.

Question 6 (Array and Function Subtyping).

Let τ represent some type. For any type τ, let $\tau[\,]$ represent the type of arrays whose elements all have type τ. For any pair of types τ_1, τ_2, let $\tau_1 \to \tau_2$ represent the type of functions from τ_1 to τ_2. Let int be the type of integers. Let $unit$, also known as *void*, be the empty type of statements that compute no value.

Let $\sqsubset$ be a binary relation that represents strict subtyping, with $\sqsubseteq$ as its reflexive closure.

Part (a)
Java and C# extend subtyping across array elements. That is, $\tau' \sqsubseteq \tau \implies \tau'[\,] \sqsubseteq \tau[\,]$. Write a fragment of Java or C# code that type checks according to this rule, but that will fail at runtime due to an incorrectly-typed element appearing in an array.

Part (b) (Elimination of Runtime Checks):
Java and C# add runtime checks to every array-element assignment in order to trap this sort of error and throw an exception instead. Propose a static program analysis and optimization that could safely eliminate some of these runtime checks. Describe your analysis in detail, including placing it in context with respect to well-known families of analysis techniques.

Part (c) (Function Subtyping):
What are the most flexible subtyping relations that can safely be permitted among function types? That is, what are the weakest conditions on τ_1, τ_2, τ_1', and τ_2' for which we can safely treat $\tau_1' \to \tau_2'$ as a subtype of $\tau_1 \to \tau_2$?

Justify the correctness of your answer. It may be useful in your arguments to treat types as mathematical sets with subtyping as subsetting.

Part (d) (Arrays as Functions):
Suppose we model array operations as a pair of functions: one for getting the value of an array element and one for setting the value of an array element. These functions must work for arrays of all types, and therefore are actually a polymorphic family of functions parameterized by array element type:

$$get : \forall\tau\ .\ \tau[\,] \to int \to \tau$$
$$set : \forall\tau\ .\ \tau[\,] \to int \to \tau \to unit$$

Note that *set* is treated as imperative: it modifies the given array in place and returns nothing ($unit$).

Any form of polymorphism means that a single value can simultaneously have multiple types. In the case of parametric polymorphism, the types of a value include its polymorphic type as well as all monomorphic instantiations of that type. In the case of subtyping, the types of a value include some most-specific type as well as all supertypes of that specific type. Show how extending array subtyping across elements allows deriving a type for *set* that violates runtime type safety of arrays. You should expect to use the function subtyping relation developed in part (c) when formulating your answer.