FALL 2002
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN—MADISON
PH. D. QUALIFYING EXAMINATION

Programming Languages and Compilers
Qualifying Examination

Monday, September 23, 2002
3:00 – 7:00 PM
382 Mechanical Engineering

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.

SPECIFIC INSTRUCTIONS:

Answer 5 of 6 questions.

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 a problem has been stated incorrectly, mention this to the proctor. If necessary, the proctor can 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.

Consider the following language of regular expressions:

$$
\begin{array}{lll}
E & ::= & 0 \\
& | & 1 \\
& | & a \qquad\qquad\qquad a \in \Sigma \\
& | & E_1 + E_2 \\
& | & E_1 \cdot E_2 \\
& | & E_1^*
\end{array}
$$

The symbol 0 stands for the empty language (*i.e.*, the empty set of strings), whereas the symbol 1 stands for the language $\{ \varepsilon \}$ (*i.e.*, the language consisting of one string, namely the empty string).

Let the symbol $\otimes$ be defined as follows (where L_1 and L_2 are sets of finite strings):

$$L_1 \otimes L_2 =_{df} \{ \alpha\beta \mid \alpha \in L_1 \text{ and } \beta \in L_2 \}.$$

The meaning of this language of regular expressions is defined by the following meaning function M, which maps a regular expression to the language it denotes:

$$
\begin{array}{l}
M[\![0]\!] = \varnothing \\
M[\![1]\!] = \{ \varepsilon \} \\
M[\![a]\!] = \{ a \} \\
M[\![E_1 + E_2]\!] = M[\![E_1]\!] \cup M[\![E_2]\!] \\
M[\![E_1 \cdot E_2]\!] = M[\![E_1]\!] \otimes M[\![E_2]\!] \\
M[\![E_1^*]\!] = \{ \varepsilon \} \cup M[\![E_1]\!] \cup (M[\![E_1]\!] \otimes M[\![E_1]\!]) \cup (M[\![E_1]\!] \otimes M[\![E_1]\!] \otimes M[\![E_1]\!]) \ldots \\
\qquad\qquad = \bigcup_{i=0}^{\infty} (M[\![E_1]\!])^i
\end{array}
$$

where L^0 means $\{ \varepsilon \}$, and L^i, for $i \geq 1$, means the i-fold $\otimes$-product of L (i.e., $L \otimes \ldots \otimes L$, i times).

Part (a)

In this part, you are to define a function *AE* (for "AcceptsEmpty") on regular expressions whose value is true if and only if the regular expression accepts ε.

$$
\begin{array}{l}
AE(0) = \mathit{false} \\
AE(1) = \mathit{true} \\
AE(a) = \\
AE(E_1 + E_2) = \\
AE(E_1 \cdot E_2) = \\
AE(E_1^*) =
\end{array}
$$

Part (b)

Show, by structural induction, that for all regular expressions E,

$$AE(E) \textit{ iff } \varepsilon \in M[\![E]\!].$$

Part (c)

A *residual* of a regular expression E with respect to symbol $a \in \Sigma$ is another regular expression E' that accepts all strings β such that $a\beta \in M[\![E]\!]$. That is, any E' that satisfies

$$\beta \in M[\![E']\!] \textit{ iff } a\beta \in M[\![E]\!].$$

is considered to be a residual of E with respect to a. For instance, if E is $a^* \cdot b + a \cdot c^* + d^* \cdot a \cdot b \cdot c$, one residual of E with respect to a is $a^* \cdot b + c^* + b \cdot c$,

(Note the resemblance of residuation to partial evaluation: We have "fixed" one symbol of the input; the residual E' defines how to handle the rest of the computation.)

The operation of *differentiation* of a regular expression with respect to a, denoted by $[a](E)$, is a way to construct a residual of E during a traversal of E's abstract syntax tree. Complete the following definition of $[a](E)$:

$[a](0) = 0$
$[a](1) = 0$
$[a](a) = 1$
$[a](b) =$ $\qquad b \in \Sigma$ and $b \neq a$
$[a](E_1 + E_2) =$
$[a](E_1 \cdot E_2) =$
$[a](E_1^*) =$

(Hint: It may be helpful to use AE in one of the cases.)

Note that if your definition were applied to the regular expression E given in the example above, it will *not* produce exactly the regular expression $a^* \cdot b + c^* + b \cdot c$; however, the language accepted by the regular expression you produce should, of course, be $M[\![a^* \cdot b + c^* + b \cdot c]\!]$.

Question 2.

This question concerns intraprocedural, flow-insensitive pointer analysis in a language with the following kinds of assignment statements:

```
ID = & ID
ID = ID
ID = * ID
* ID = ID
```

The goal of the analysis is to discover, for each variable x, which variables x might point to. Because we are interested in a flow-*insensitive* analysis, we ignore the actual control flow of the program being analyzed; in other words, we treat the program as if there were an edge in its control-flow graph from every node to every other node (so the statements could be executed in arbitrary order, and each statement could be executed an arbitrary number of times).

For example, if a program includes (only) these statements:

```
p = &x
q = p
p = &y
x = &a
t = &z
*q = t
```

then the result of performing flow-insensitive pointer analysis on the program should be:

```
p might point to x or y
q might point to x or y
x might point to a or z
y might point to z
t might point to z
```

Remember, the fact that the statement “p = &y” comes after the statement “q = p” is irrelevant -- the statements are treated as if they could execute in any order.

Part (a)

One way to define our desired pointer analysis is using a standard dataflow analysis applied to the “flow-insensitive” control-flow graph. Recall that a standard dataflow analysis is defined by supplying:

1. A lattice of values (where each value represents one “dataflow fact”) including a meet operator.

2. A dataflow function for each CFG node *n* that maps n_{in} to n_{out} (where n_{in} is the dataflow fact that holds before node *n* is executed, and is defined to be the meet over all m_{out} for all predecessors *m* of *n*, and n_{out} is the dataflow fact that holds after node *n* is executed).

3. A special “init” value (the dataflow fact that holds before the enter node of the CFG).

Define a standard dataflow analysis so that for each node *n*, n_{in} is a map from variables to their points-to sets.

Part (b)

Another way to define our pointer analysis is using Horn clauses. The idea is that each assignment statement in the program generates one Horn clause fact, and that there are a set of Horn clause rules that define how to infer more facts.

For example, the assignment

p = &x

would generate the fact

pointsTo(p, x).

The assignment

q = p

would generate the fact

assign(q, p).

and there would be a rule

pointsTo(A, X) :- pointsTo(B, X), assign(A, B).

Note that we can use the two facts and the rule given above to deduce the fact

pointsTo(q, x).

You are to finish the definition of the pointer analysis via Horn clauses by defining a Horn clause fact for each of the remaining two kinds of assignment statements, and by defining as many new Horn clause rules as needed.

Note that the desired result of using this technique on the program shown above is:

pointsTo(p, x)
pointsTo(p, y)
pointsTo(q, x)
pointsTo(q, y)
pointsTo(x, a)
pointsTo(x, z)
pointsTo(y, z)
pointsTo(t, z)

Illustrate your Horn clause facts and rules by giving the initial set of facts that would be generated for the example program, and then showing how to use those facts and the Horn clause rules that you defined to determine the two pointsTo facts:

pointsTo(q, x)
pointsTo(x, z)

Part (c)

Now consider extending the analysis to be interprocedural. We still want it to be flow-insensitive, and also context-insensitive; e.g., the analysis of the following code:

```
foo(&p, &x)
foo(&q, &y);

void foo(int **a, int *b) { *a = b }
```

should produce the information that both p and q might point to both x and y.

Assume that, as in the C language, all parameters are passed by value. Choose either the standard dataflow analysis approach or the Horn clause approach, and explain how function calls should be treated to achieve the desired effect (i.e., explain how to represent a function call in the flow-insensitive control-flow graph, and what changes if any must be made to the analysis defined in Part (a), or explain what new Horn clause facts are generated for a function call, and what new rules, if any, must be defined to deduce the appropriate pointsTo facts).

Question 3.

For this question, consider a procedural language that allows non-local gotos: A statement of the form "goto L" is legal if, when the goto is executed, there is some procedure that has been called (and has not yet returned) that includes a statement labeled L (this procedure could be the one that contains the goto). If this is the case, control is transferred to the L label of the most recently called such procedure; otherwise, a runtime error occurs.

There are two basic approaches to implementing this kind of non-local goto: one that involves "unwinding" the stack when a goto is executed (i.e., executing some code in each procedure up the call chain), and one that involves executing a constant number of instructions to fix up the stack regardless of the depth of the call chain, then jumping directly to the target of the goto.

Describe each of the two approaches. For each case, say what kind of special static analysis (if any) the compiler would need to do, for which procedure calls and procedure entries (if any) non-standard code would need to be generated, and what would happen when a non-local goto is executed.

Question 4.

For this question you will be asked to describe how a compiler might take advantage of a new language feature to speed up certain function calls. Be sure to consider whether it will always be possible to apply your optimization, or whether that will depend on some property of the function. Also consider whether your technique will always be successful, or whether there might be cases where the average execution time for the function in the "optimized" version of the program is greater than in the unoptimized version.

Part (a)

Consider a procedural language in which one of a function's parameters can be declared "frequentlySame"; for example:

```
int foo( float f, frequentlySame int k, int *p )
```

This means that the programmer expects that foo will be called many times, and that parameter k will usually have one of a small set of values. However, the actual values in that set are not known at compile time.

Describe how a compiler could take advantage of "frequentlySame" declarations to lower the average execution time for function foo when it is called with one of k's frequently occurring values. (The execution time for foo when it is called with a value for k other than one of its frequently occurring values is unimportant.)

Part (b)

Now assume that a "frequentlySame" declaration applies to *all* of foo's parameters; i.e., the declaration

```
int foo( float f, int k, int *p ): frequentlySame
```

means that the programmer expects that foo will be called many times, and that there will be a single set of values $\{v_1, v_2, v_3\}$ such that on most calls, f will have value v_1, k will have value v_2, and p will have value v_3. Again, the actual values of v_1, v_2, and v_3 are not known at compile time.

Describe how a compiler could take advantage of this kind of "frequentlySame" declaration to lower the average execution time for function foo when it is called with values v_1, v_2, v_3. (Again, the execution time for foo when it is called with values other than v_1, v_2, v_3 is unimportant.)

Question 5.

This question concerns the design and use of a garbage collector as part of a program's run-time system.

Part (a)
Assume you have a language like Java, in which all pointers (references) are strongly typed (known at compile-time to reference a single fixed type). Outline the design of a garbage collector that may be run while the application program (the mutator) is stopped. Your collector should (of course) have the property that all objects marked as garbage (and deleted) really are garbage. Does the collector you present also have the property that all inaccessible objects are recognized and deleted?

Part (b)
Languages like C and C++ complicate garbage collection in that pointers may be difficult to recognize because of casting and type unions. What changes are needed in the garbage collector you proposed in (a) to support languages like C and C++?

Part (c)
Some languages, like Java, allow simultaneous execution of multiple threads. On processors that can execute more than one thread simultaneously, it may be important to make a garbage collector exhibit a degree of concurrency; i.e., if there is more than one thread, at least one should continue to run during garbage collection. What changes are needed for your garbage collector to make it concurrent?

Question 6.

Computers are used in a wide-variety of critical applications, like banking, communication and defense. Applications in these areas often must run continuously, with no down-time. Yet these applications still must be corrected or upgraded. One approach is to do corrections or upgrades while the application is running. Assume the application in question is coded in Java, and that we wish to replace a class C with a corrected or upgraded class, C_new.

Part (a)
What kinds of changes to class C should be allowed and why? Consider methods and fields, public and private access, and changes involving deletion, modification, and addition.

Part (b)
After C is replaced with C_new, we have to worry about how old and new instances of C will co-exist. One possibility is to convert existing objects of class C into objects of class C_new. How might this be done?

Another alternative is to allow the two kinds of objects to co-exist indefinitely, with all newly created objects being of class C_new. What are the advantages and disadvantages of the two approaches?