FALL 1996
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN—MADISON
PH. D. QUALIFYING EXAMINATION

Programming Languages and Compilers
Qualifying Examination

Monday, September 16, 1996
3:00 – 7:00 PM
115 Psychology

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.

This question concerns recursion, and optimization opportunities when a function is not recursive. (We will use the term "function" to refer generically to both functions and procedures.) Assume throughout that the language in question does not support higher-order functions, indirect calls (*e.g.*, calls through pointers to functions), or inheritance. You may think either in terms of a functional language, such as first-order pure Scheme, or an imperative language, such as C without function pointers.

Part (a)

Explain two optimizations that can be performed on non-recursive functions (that could not be performed on recursive functions), and *describe* what the savings would be in each case.

Part (b)

One way to identify a program's non-recursive functions is to solve a set of equations: For each function f, there is a variable V_f, whose value is a (possibly empty) set of function names. If f calls functions $g_1, \cdots, g_k$, there is an equation

$$V_f = \{ g_1, \cdots, g_k \} \cup V_{g_1} \cup \cdots \cup V_{g_k}.$$

If f does not call any functions, the equation is

$$V_f = \emptyset.$$

For example, if f calls g, g calls h, and h does not call any functions, the system of equations is:

$$\begin{aligned} V_f &= \{ g \} \cup V_g \\ V_g &= \{ h \} \cup V_h \\ V_h &= \emptyset \end{aligned}$$

These equations happen to have a unique solution:

$$\begin{aligned} V_f &= \{ g, h \} \\ V_g &= \{ h \} \\ V_h &= \emptyset \end{aligned}$$

Part (b)(1)

Not all sets of equations that arise from the above definition have a unique solution. *Give* an example program (and its set of equations) such that the program's equations have more than one solution. (Note: For your program to be an acceptable answer, all functions should be callable (transitively) from some distinguished function, *e.g.*, f in the example given above.) *List* all the solutions to the equations you give.

Part (b)(2)

Recall that if you have a complete partial order D with a least element $\perp$, and F is a continuous function from D to D, a recursive equation of the form $X = F(X)$ has a least solution $\bar{X}$ (where "least" is defined with respect to the ordering relation $\sqsubseteq$ of D) that can be obtained as

$$(*) \quad \bar{X} = \bigsqcup_{i=0}^{\infty} F^i(\perp).$$

Given the set of equations that arise from a program P, as defined above, *define* an appropriate partial order D_P and a function F such that the *least* fixed-point of F is the desired solution; that is, the goal is that in the least fixed-point solution the value of each variable V_f corresponds exactly to the functions callable (transitively) from f. (Be sure to *say explicitly* what the ordering is on elements of D_P since the notion of "least" is defined with respect to this ordering.)

(Hint: Recall that if a partial order D is finite, then (i) every monotonic function from D to D is continuous and (ii) D is a *complete* partial order.)

Part (b)(3)

Recall that (if D has no infinite ascending chains) equation (*) has operational significance—it corresponds to an iterative process for finding $\bar{X}$. For your set of equations from Part (b)(1), *trace* to completion the iterative process that would arise from your answer to Part (b)(2).

Question 2

This question concerns some of the basics of the λ-calculus.

Part (a)

(i) What does it mean for a λ-term to be in normal form?

(ii) What does it mean for a λ-term to be equal to another λ-term?

(iii) What does it mean for λ-term T_1 to be a fixed point of λ-term T_2?

Part (b)

Which of the following are true statements about the λ-calculus? In each case, justify your answer.

(i) Every λ-term is equal to a λ-term that is in normal form.

(ii) Every λ-term is equal to a λ-term that is not in normal form.

(iii) Every λ-term has some λ-term as a fixed point.

(iv) Every λ-term is a fixed point of some λ-term.

(v) There exists a λ-term that is its own fixed point.

(vi) The term $\lambda x.xx$ is its own fixed point.

Question 3.

Part (a)

What opportunities for optimization does procedure in-lining open up?

Part (b)

What are the differences between the results of optimizations based on in-lining, and optimizations based on the results of interprocedural analysis (*e.g.*, for constant propagation)?

Part (c)

With a recursive program, a program optimizer that implements procedure in-lining could go into an infinite loop when it tries to do an in-line expansion of recursive procedure *P* into *P* into *P* . . . However, there are other optimizations that can still be done for recursive programs. In particular, if one can determine that some of the arguments at a certain call site are known at compile time, then a specialized version of a procedure can be created that is optimized for that particular invocation. One way to do this involves an analysis phase, called *binding-time analysis*, that propagates symbolic values "KNOWN" and "UNKNOWN" through the program. For instance, consider the following program:

```
procedure P(a, b)          // KNOWN, UNKNOWN
begin
  call Q(b*b, a)           // UNKNOWN, KNOWN
  call Q(a*a, a*b)         // KNOWN, UNKNOWN
end
```

If the first argument to *P* is KNOWN then the second argument at the first call on *Q* is KNOWN, as is the first argument at the second call on *Q*. We say that procedure *Q* has two *binding patterns*, <UNKNOWN, KNOWN> and <KNOWN, UNKNOWN>.

Now consider the following program (fragment):

```
procedure P4(a, b, c, d)
begin
    if . . . then
        call P4(1, b, c, d)
    else if . . . then
        call P4(a, 2, c, d)
    else if . . . then
        call P4(a, b, 3, d)
    else
        call P4(a, b, c, 4)
end
```

Just taking into account this fragment (*i.e.*, without knowing anything about other calls on P_4) how many binding patterns can arise for P_4, and why?

Generalizing from this example, what are the problems with the idea of specializing a procedure based on known arguments?

Question 4

Part (a).

Consider a simple language that includes only variables of type *int* or type *pointer-to-int*. Assume that programs are represented using a control-flow graph in which a node either represents a predicate or an assignment statement of one of the following forms:

- x = &y;
- x = y;
- x = *y;
- *x = y

Recall that the goal of dataflow analysis is to determine, for each node of a program's control-flow graph, the set of facts that are guaranteed to hold just before and just after the code represented by that node is executed. Further recall that a dataflow problem can be defined by specifying:

1. A lattice of values (each lattice element represents one "set of facts") and the lattice's meet operation.
2. A dataflow function for each predicate and each kind of statement. (The function maps the set of facts true before the predicate/statement executes to the set of facts true after it executes.)

Define a dataflow problem to determine, for each pointer variable *p*, the set of variables *p* might point to.

Part (b).

Recall that a function f is distributive (over meet operator $\sqcap$) iff

$$\forall\, x, y\colon f(x \sqcap y) = f(x) \sqcap f(y)$$

and that f is monotonic iff

$$\forall\, x, y\colon x \leq y \Rightarrow f(x) \leq f(y)$$

Why do we care whether the functions defined for a dataflow-analysis problem are monotonic or distributive?

Part (c).

If you got Part (a) right, the function you wrote for the statement "x = *y". is *not* distributive. Give an example to illustrate this.

Question 5

Part (a).

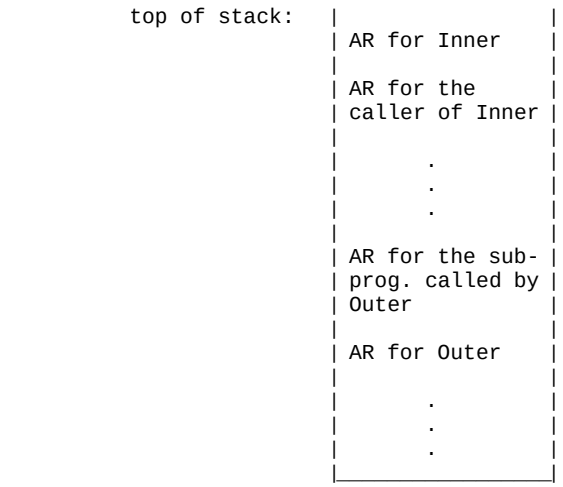
In Pascal, whenever a procedure defined at nesting depth n is called, there are guaranteed to be active instances of all enclosing procedures. One reason this is important is given below in Part b. Give another reason why this is important.

Part (b).

In Pascal, it is possible to have a *goto* in a subprogram whose target is in an enclosing subprogram. For example:

```
program main(output);
    procedure P1;
    label 10, 20;
        procedure P2;
            procedure P3;
            begin
              goto 10;         /* here's the non-local goto */
            end;
        begin
          P3;
        end;
    begin
          P2;
          goto 20;
      10: writeln('executed goto');   /* here's the target of the goto */
      20: writeln('all done');
    end;
begin
  P1;
end.
```

Consider a subprogram Inner that includes a *goto* to a label in an enclosing subprogram Outer. At runtime, just before the non-local goto is executed, the runtime stack holds activation records as pictured below:



Part of executing the *goto* involves popping off the stack all of the activation records down to (but not including) Outer's activation record. Describe how this could be done; that is, how could the activation record for Outer be identified at runtime, and what values would be changed to accomplish the pop? Note that the efficiency of the execution of the non-local *goto* itself is not important; however, your solution should not introduce a lot of overhead into the execution of the rest of the program.

Question 6

Assume we have an expression involving only integer scalar variables and literals. Moreover, each variable is used only once (so that the expression can be represented as a tree rather than a dag). Also assume that all binary operators are non-associative and non-commutative.

Part (a).

Give an algorithm that determines the minimal number of registers needed to evaluate the expression, and then generates a sequence of instructions that evaluates the expression using the minimal number of registers. (You can write a single algorithm that does both parts, or one algorithm that determines the minimal number of registers needed, and a second algorithm that uses the results of the first to actually generate the instructions). Every instruction must have one of the following forms (in particular, all variables and literals must be loaded into registers before they can be used):

```
Load  Rx, id   // load variable id into register x
LoadI Rx, k    // load integer literal k into register x
Op    Rx, Ry   // register x := register x Op register y
```

Part (b).

Many modern architectures have delayed loads in which a value loaded into a register cannot be used in the very next instruction. This leads to a trade-off between the number of registers and the time needed to evaluate an expression. Give an example expression with the following property:

> The sequence of instructions produced by your algorithm of part (a), while optimal in the number of registers used, is not optimal in time (because of a pipeline stall due to a premature use of a loaded register).

Demonstrate this property by giving another sequence of instructions (that uses more registers) that evaluates the expression without any pipeline stall.

Part (c).

Give an algorithm that takes as input the sequence of instructions produced by your algorithm of part (a) and that produces a new sequence of instructions that causes as few pipeline stalls as possible (using only as many additional registers as absolutely necessary).