FALL 1994
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN—MADISON
PH. D. QUALIFYING EXAMINATION

Programming Languages and Compilers
Qualifying Examination

Monday, September 19, 1994
3:30 – 7:30 PM
1240 Computer Sciences

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 is about properties of lambda expressions.

Definition:

Lambda expressions A and B are said to be *similar* (written **sim**(A, B)) iff there exists a lambda expression C such that both A and B can be rewritten to C using a sequence of zero or more α and/or β reductions.

Definition:

Lambda expressions A and B are said to be *convertible* (written **con**(A, B)) iff one can be rewritten to the other using a sequence of zero or more α and/or β reductions and/or β expansions.

Prove that **sim**(A, B) iff **con**(A, B). You may justify steps of your proof by invoking the Chirch-Rosser Theorem: if $X_0 \rightarrow^*_{\alpha,\beta} X_1$ and $X_0 \rightarrow^*_{\alpha,\beta} X_2$, then $\exists\, X_3$ such that $X_1 \rightarrow^*_{\alpha,\beta} X_3$ and $X_2 \rightarrow^*_{\alpha,\beta} X_3$.

Question 2.

Consider the simple imperative language defined below.

program → *cmd*
cmd → Id := *exp* | if *cond* then *cmd* else *cmd* fi | *cmd* ; *cmd*
exp → Int | Id

That is, a program is a command, and a command is an assignment, an if-then-else, or a command followed by another command; expressions are either integer literals or identifiers. (The definition of conditions is not needed for this question.)

A standard denotational semantics for this language is given below (the meaning function **B** for boolean expressions is not needed for this question and so is omitted). A *State* is a mapping from identifiers to values; initial state σ_0 is the state that maps all identifiers to zero. The meaning function **I** used to define the meaning of an integer literal expression simply returns the value of its argument; the function "update" used to define the meaning of the assignment command takes three parameters: a state σ, an identifier x, and a value v, and returns a state that is the same as σ except that it maps x to v.

Meaning Functions

P: Command → State
C: Command → State → State
E: Expression → State → Integer

P[[*C*]] = **C** [[*C*]] σ_0
C[[*Id* := *exp*]] = $\lambda\sigma$. update(σ, *Id*, **E**[[*exp*]] (σ))
C[[if *cond* then C_1 else C_2 fi]] = $\lambda\sigma$. if **B**[[*cond*]] (σ) then **C**[[C_1]] (σ) else **C**[[C_2]] (σ)
C[[C_1 ; C_2]] = $\lambda\sigma$. **C**[[C_2]] (**C**[[C_1]] (σ))
E[[Int]] = $\lambda\sigma$.**I**[[Int]]
E[[Id]] = $\lambda\sigma.\sigma$(Id)

Now consider extending the language by adding values of type *real*, variable declarations, and dynamic type checking. The new syntax is given below.

program → *decl cmd*
decl → Id : int | Id : real | *decl* ; *decl*
cmd → Id := *exp* | if *cond* then *cmd* else *cmd* fi | *cmd* ; *cmd*
exp → Int | Real | Id

The idea is that the meaning of a program should be $\perp$ if an assignment statement is executed that assigns a value of one type into a variable of another type.

You are to write a denotational semantics for this extended language. Define the meaning functions to have the types given below, where *TypeEnv* is a map from identifiers to types, and $State_\perp$ is a map from identifiers to values or $\perp$.

P: (Command × Declaration) → $State_\perp$
D: Declaration → TypeEnv → TypeEnv
C: Command → ($State_\perp$ × TypeEnv) → ($State_\perp$ × TypeEnv)
E: Expression → State → Integer ∪ Real

(QUESTION CONTINUED ON THE NEXT PAGE)

Complete the definitions given below. You may use the meaning function **F**, which returns the value of its real literal argument. You may assume that a program includes neither multiply declared variables nor uses of undeclared variables. Be sure to provide any other auxiliary functions that you use in your definitions.

P[[C, D]] = _____

D[[Id:*int*]] = $\lambda\tau$._____

D[[Id: *real*]] = $\lambda\tau$._____

D[[D_1 ; D_2]] = $\lambda\tau$._____

C[[Id := *exp*]] = λ < σ, τ >._____

C[[if *cond* then C_1 else C_2 fi]] = λ < σ, τ >._____

C[[C_1 ; C_2]] = λ < σ, τ >._____

E[[Int]] = $\lambda\sigma$._____

E[[Real]] = $\lambda\sigma$._____

E[[Id]] = $\lambda\sigma$._____

Question 3.

This question is about dataflow-analysis problems in which all the functions are assumed to be monotonic, but not necessarily distributive.

Recall that $f\colon D \to D$ is monotonic iff $\forall\, d_1, d_2$ such that $d_1 \sqsubseteq d_2$, $f(d_1) \sqsubseteq f(d_2)$.

Part (a)

Prove that if f is monotonic then

$$f(d_1 \sqcap d_2) \sqsubseteq f(d_1) \sqcap f(d_2).$$

Part (b)

In this part, you will show that the greatest fixed-point solution (*gfp*) to a dataflow problem approximates the meet-over-all-paths solution (*mop*). But first we need some notation.

Let N be the set of nodes of the flow chart, let D be a meet semi-lattice with greatest value $\top$, and let $E(m, n)\colon D \to D$ denote the function on flow-chart edge $m \to n$. Let $pred(m)$ denote the set of nodes that are predecessors of node m in the flow chart. Then, $gfp\colon N \to D$ is the greatest solution to

$$gfp(n) = \bigsqcap_{m \,\in\, pred(n)} E(m, n)(gfp(m)).$$

Let $pathsTo(m)$ be the (possibly infinite) set of paths from the flow-chart's start node to node m. (Note that the paths in $pathsTo(m)$ are all of finite, but unbounded, length.) Suppose that p is a path in $pathsTo(m)$, for example $p = [start, n_1, \cdots, n_k, m]$. We define pf_p, the *path function* for p, to be:

$$pf_p = E(n_k, m) \circ \cdots \circ E(n_1, n_2) \circ E(start, n_1).$$

(The path function for an empty path is defined to be the identity function, $\lambda\, d\,.\, d$.) Then $mop(n)$, the meet-over-all-paths solution is defined as $mop(n) = \bigsqcap_{p \,\in\, pathsTo(n)} pf_p(\top)$.

Show that for all $n \in N$ and all paths $p \in pathsTo(n)$:

$$gfp(n) \sqsubseteq pf_p(\top). \qquad (\dagger)$$

[Hint: Use induction on the length of paths.]

Now use (†) to show that $gfp(n) \sqsubseteq mop(n)$.

Question 4.

Assume that we are translating a simple language that contains only scalar integer variables and constants (*i.e.*, no arrays, pointers, structs, *etc*.) and that a program consists of a single procedure (with no calls). Assume that every expression E is of the form $v_1\ op_1\ v_2\ op_2 \cdots v_n$. If we can show that E has already been computed and not killed along all paths to point p, then E is *redundant* and its computation at p would be unnecessary.

Part (a)

Formulate a data flow problem that determines, for every point p, which expressions are redundant at that point. Give the lattice of values you will use along with the appropriate meet operation. Each basic block b will have values In_b and Out_b that will indicate which expressions are redundant on entrance to and exit from b, respectively. Give the appropriate equations that define *In* and *Out* in terms of the program's flow graph and the contents of its basic blocks. How do we determine the expressions generated and killed in each basic block?

Part (b)

If an expression E is computed somewhere in the program and not killed along some, but not all, of the execution paths to p, it is said to be *partially redundant* at p. Explain how to modify your solution to Part (a) to determine, for every point p, what expressions are partially redundant.

Part (c)

Suppose E is used at point p and is partially redundant at p. If we insert computations of E on those paths where E is not available, we may be able to make E redundant at point p (and hence unnecessary). Give an example that shows why this transformation can be a worthwhile optimization. Does it always increase speed? Does it always decrease size? Assuming the transformation is worthwhile, outline how you would determine where to add computations of E to make E redundant at p.

Part (d)

Assume now that we add pointers and the & operator (but no arrays, structs, or heap-allocation operations) to our language. How must your solution to Part (a) be changed?

Question 5.

Part (a)

Register allocation within a procedure is often modeled as a graph coloring problem in which selection of colors corresponds to register assignment. In this formulation, to what in the procedure do the nodes and arcs of the graph correspond?

Part (b)

If we have R colors (*i.e.*, registers) it may be impossible to color a particular graph. The graph then has to be simplified by removing nodes and arcs. Explain how we might choose which nodes and arcs to remove when coloring is unsuccessful (*i.e.*, what are the criteria that make a node or arc suitable for removal).

Part (c)

What changes (if any) must be made to the current intermediate code (i) when a node is successfully colored and (ii) when a node or arc is removed because the coloring process is blocked? Illustrate your answers with suitable examples.

Part (d)

Some values, like double precision values, require a pair of registers. Architectures usually require that the register pair be adjacent (*e.g.*, r_i and r_{i+1}). Explain how to generalize the usual graph coloring model of register allocation to handle adjacent register pairs. How should we handle graphs that cannot be colored with R colors (being sure to respect adjacency requirements)?

Question 6.

Most versions of Lisp contain two related constructs: `CATCH` and `THROW`. The semantics of `CATCH` and `THROW` are as follows: The execution of

```
(CATCH tag-name expression)
```

causes "`expression`" to be evaluated; if this evaluation returns a value *v*, then *v* is the value of the entire `CATCH` expression. (The evaluation may not return a value due to some `THROW` being encountered; see below.) In the term above, "`expression`" may be any legal Lisp expression.

In the execution of

```
(THROW tag-name return-expression)
```

"`return-expression`" is first evaluated; if this evaluation returns a value, that value is immediately returned to the most recently encountered (at runtime) `CATCH` with the same "`tag`-name". (Evaluation of "`return-expression`" may not return a value if another `THROW` is encountered during the evaluation of "`return-expression`".) In the term above, "`return-expression`" may be any legal Lisp expression.

For example:

```
(CATCH 'foo
  (if a
      (begin (THROW 'foo 1) expr1)
      (begin expr2 expr3)))
```

returns `1` if is `a` is true and the result from `expr3` otherwise.

Part (a)

Describe an implementation of `CATCH` and `THROW`.

Part (b)

Some versions of Lisp further support the so-called "return again" feature, which allows a `THROW` to occur even after the (dynamic) scope of the enclosing `CATCH` has been exited. In such cases, execution of the program continues as if the relevant `CATCH` (the one with the same tag-name as the `THROW`) had just completed execution.

Describe an implementation for `CATCH` and `THROW` that supports the "return again" feature. Be sure to explain how this scheme efficiently locates the proper `CATCH` to jump to as the result of a `THROW`.