SPRING 1997
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN—MADISON
PH. D. QUALIFYING EXAMINATION

Programming Languages and Compilers
Qualifying Examination

Monday, February 3, 1997
3:00 – 7:00 PM
2365 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 that 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.

Assume we have allocated registers for a procedure's live ranges using graph coloring (any variant—your choice). At point p within the procedure a subroutine call occurs. Naturally, at p some (or all) of the registers must be saved and restored.

Part (a)

One very simple approach is to have a fixed set of registers that are saved before *every* call (and restored after the call), and another (disjoint) set of registers that are saved at the beginning of every subroutine (and restored just before the subroutine returns). What are the disadvantages of this approach? Describe a better approach.

Part (b)

It is standard practice to save caller-save registers just before transferring control to a subroutine, and to restore them immediately after returning from a call. However, this may force us to do more work than necessary. Just because a register value is live at a call, it is not necessarily live on *all* execution paths containing the call. Hence it may be better to save and restore a particular register only along those paths where the save and restore really are necessary. Explain how we can determine where to move save and restore code so that it is executed only on those paths where it is really needed.

Question 2.

This question is about debuggers, such as *dbx* or *gdb*.

Part (a)

What type of information does a compiler communicate to a debugger?

Part (b)

Give three examples that illustrate how program optimization can cause a debugger to produce misleading or erroneous results. In each case, identify which debugger operations the optimization affects, and explain why it causes the debugger to produce misleading results.

Part (c)

Choose one of the examples from part (b) and show how the compiler could pass additional information to a debugger to avoid this problem.

Question 3.

Two approaches to (automatic) memory management are *garbage collection* and *reference counting*.

Part (a)

Briefly describe the key aspects of each of the two techniques. What are their relative advantages and disadvantages?

Part (b)

It would be nice to change the C language so that dynamic memory allocation (via calls to *malloc*) is still supported, but explicit calls to *free* are replaced by automatic deallocation via garbage collection or reference counting. What aspects of the C language make this difficult?

Part (c)

Choose one aspect that you discussed above in part (b) and explain how it could be handled. (That is, describe additional information that must be generated by the compiler and maintained at run time, as well as any modifications to the standard approach to garbage collection or reference counting that would be necessary.)

Question 4.

Part (a)

An instance of variable x is considered to be "non-local" if there is no declaration of x in the current scope. Recall that in a statically scoped language (*e.g.*, Pascal, C, etc.), a use of a non-local variable x corresponds to the declaration of x made in the most closely enclosing scope.

Now consider a dynamically scoped language. To which declaration does a use of non-local variable x correspond?

Part (b)

What are the relative advantages and disadvantages of static and dynamic scoping?

Part (c)

In a statically scoped language, it is possible to determine at compile time whether each use of a variable is "undeclared"; *i.e.*, whether it has a corresponding declaration. In general, this is not true in a dynamically scoped language, and so programs written in a dynamically scoped language may have run-time errors due to the use of an undeclared variable.

It *is* possible to design a compile-time analysis for a dynamically scoped language to detect such uses that is correct under the assumption that all paths in the program's control-flow graph are executable. That is, the analysis will identify all uses of non-local variables such that there is some path to that use for which there is no corresponding declaration.

Assume that the language in question does not support functions as objects; *i.e.*, function calls are made only using the name of the function, rather than via a function-valued expression. Define an analysis that identifies uses of undeclared variables, and illustrate the analysis using the following program:

```
void main() {
  call f();
  call g();
}

void f() {
  int x;
  call g();
}

void g() {
  write(x);
}
```

Question 5.

Consider the following definitions of recursive functions concat, append, and combine:

```
let rec concat(L1, L2) =
cases (L1) of
    nil: L2
    x::L: x::concat(L, L2)

let rec append(L, x) =
cases (L) of
    nil: x::nil
    a::L1: a::append(L1, x)

let rec combine(L1, L2) =
cases (L2) of
    nil: L1
    x::L: combine (append(L1, x), L)
```

Part (a)

Given lists L1 and L2 of lengths n and m, respectively, what are the worst-case running times of concat and combine?

Part (b)

Use structural induction to prove that, for all lists L1 and L2,

concat(L1, L2) = combine(L1, L2).

Question 6.

East is east and west is west, and never the twain shall meet . . .

Some programming-languages researchers think that the same chasm exists between dataflow analysis and abstract interpretation. This is both wrong and counter-productive. The two areas have simply tended to emphasize different aspects: Dataflow-analysis papers tend to concern efficient *algorithms* for solving dataflow-analysis problems, whereas abstract-interpretation papers emphasize the issue of demonstrating that an analysis is *correct* with respect to a language's semantics. Both are important issues. (But the absence of proofs of correctness in dataflow-analysis papers sometimes raises the ire of abstract-interpretation enthusiasts.)

This question concerns how to show the correctness of dataflow-analysis problems. In particular, Parts (a) and (b) concern how abstract-interpretation can be used to show that a dataflow-analysis problem is correct. Then, Parts (c), (d), and (e) will relate this to standard algorithms for constant propagation.

First, some words about the setting and notation:

- We will deal only with the single-procedure case: A program consists of a single procedure that manipulates only integer variables. We assume that there are no edges into the start node of the program's control-flow graph.
- We will use unsuperscripted symbols when referring to the concrete (collecting) semantics (*e.g.*, t, c, $\sqcup$, etc.), and use symbols superscripted with "#", such as ($t^{\#}$, $c^{\#}$, $\sqcup^{\#}$, etc.), when referring to the abstract semantics (or "domain of dataflow facts").
- We assume that the concrete and abstract semantic functions are associated with the *edges* of the program's control-flow graph. Let t_i and $t_i^{\#}$ denote the semantic functions associated with edge i.

Let *Concrete* and *Abstract* be the domains from which are drawn the values assigned to the nodes of the control-flow graph by the concrete and abstract semantics, respectively. Let α: *Concrete* $\rightarrow$ *Abstract* be the abstraction function.

One thing to note: Following standard practice in abstract interpretation, we will assume that the abstract confluence operator is the join operator ($\sqcup^{\#}$) of *Abstract*. The standard convention in dataflow analysis is the opposite: The confluence operator is typically meet ($\sqcap^{\#}$), and one speaks, for example, of finding the "meet-over-all-paths solution". For this reason, here we relate the concrete semantics to the "join-over-all-paths solution".

In Parts (a) and (b), assume that the following properties hold:

(i) The $t_i^{\#}$ are monotonic.

(ii) Function α is (infinitely) distributive; *i.e.*, $\alpha(\bigsqcup_{c \in C} c) = \bigsqcup^{\#}_{c \in C} \alpha(c)$, where C is a possibly infinite set of concrete values.

(iii) For all edges i and all $c \in$ *Concrete*, $\alpha(t_i(c)) \sqsubseteq t_i^{\#}(\alpha(c))$.

Let Λ and $\Lambda^{\#} =_{df} \alpha(\Lambda)$ be the appropriate concrete and abstract values for the start node of the control-flow graph.

Let p_n be a path from the start node of the control-flow graph to node n, and let $pf[p_n]$ and $pf^{\#}[p_n]$ be the corresponding concrete and abstract path functions, respectively (*i.e.*, the compositions of the concrete and abstract semantic functions on the edges of the path, respectively).

Finally, define

$$\textit{collecting-semantics}(n) =_{df} \bigsqcup_{p \in \textit{pathsTo}(n)} pf[p](\Lambda)$$

and

$$\textit{join-over-all-paths}(n) =_{df} \bigsqcup^{\#}_{p \in \textit{pathsTo}(n)} pf^{\#}[p](\Lambda^{\#})$$

Part (a)

Show that for all nodes n,

$$\alpha(\textit{collecting-semantics}(n)) \sqsubseteq \textit{join-over-all-paths}(n). \qquad (*)$$

Part (b)

Dataflow-analysis problems are normally solved by setting up equations from the abstract semantic functions $t_i^{\#}$ and finding the fixed point (least solution) of these equations. Why does (*) imply that the equation-solving method yields a solution that is correct with respect to the language's semantics? (Note: We are not looking for a proof here, just a short prose justification.)

Part (c)

In constant propagation, the concrete (collecting) semantics assigns each node of the control-flow graph a set of variable-to-integer maps:

$$\textit{Concrete} = 2^{\textit{var} \to \textit{int}},$$

and $\sqcup$ is set union. Suppose edge i represents an assignment statement $x := e$. The semantic function t_i is then

$$t_i =_{df} \lambda\, S\, .\, \{\, s[x \leftarrow [\![e]\!]s] \mid s \in S \,\},$$

where $[\![e]\!]$ denotes the expression-evaluation function and $s[x \leftarrow c]$ denotes the update of map s at x with value c.

The abstract semantics, on the other hand, associates each control-flow-graph node with a single map:

$$\textit{Abstract} = \textit{var} \to \textit{int}_{\perp}^{\top}.$$

Here $\textit{int}_{\perp}^{\top}$ is the discrete domain of integers extended with $\top$ and $\perp$, so that, for example, $3 \sqcup^{\#} 17 = \top$.

Give an appropriate definition for the semantic function $t_i^{\#}$ (where edge i represents an assignment statement $x := e$). Then give an example and show that it demonstrates Property (i).

Part (d)

Give the definition of abstraction function α. (Remember that a concrete value is a *set* of variable-to-integer maps.) Then give an example and show that it demonstrates Property (ii). (It is okay to give an example that just demonstrates finite distributivity.)

Part (e)

Using your definitions from Parts (c) and (d), give an example and show that it demonstrates Property (iii) (*i.e.*, that $\alpha(t_i(c)) \sqsubseteq t_i^{\#}(\alpha(c))$).