Compiler

Qualifier Exam

August 21, 2006

This is a closed book exam. Answer all questions.

1. (25 points)

Consider an optimization called "common subexpression elimination", in which redundant computations (common subexpressions) are to be avoided. For example in the code:

```
(a+b)*i + (a+b)/j;
```

the (a+b) should not have to be computed twice.

A) Discuss what a compiler would have to do in order to perform this optimization correctly during the code generation for a particular program.

B) Explain and give an example that shows why this optimization is of value even if the programmer is smart enough to avoid writing explicit common subexpressions such as the example given above.

2. (25 points)

Some object-oriented languages support a feature called operator overloading, in which classes may define implementations of familiar operators. But many object-oriented languages explicitly disallow operator overloading.

A) Give an example that shows why operator overloading can pose a semantic dilemma for language designers.

B) Sketch out a design for how an object-oriented language might implement operator overloading. Show what information (if any) must be represented at compile time, or at run-time in each instance and in the class, so that operators can be overloaded by various classes. Indicate the runtime time and space complexity of your solution.

3. (25 points)

Many higher-level (for example, scripting) languages are implemented with a simple bytecode compiler (that does not do much program analysis but just compactly stores a straightforward representation of the program), and a relatively complex run-time interpreter. In Unicon, variables are untyped but values are strongly typed; the type of the variable is determined by the value it is holding. If a variable whose value is currently a string is used in an arithmetic expression, and if it can be successfully converted to a number, it is converted automatically; similarly, numeric values are automatically converted to strings on demand by operations such as string concatenation, or the write() function.

A) Given the dynamic typing scheme above, the following piece of code

```
i := 1
j := 1
```

```
while i < 100 do {
   while j < 100 do {
      j +:= 1             # uses j as int (increments by 1)
      write("i,j = ", i,",",j) # uses i and j as strings
      # ... more arithmetic use of i and j here
      }
   i +:= 1                # uses i as int (increments by 1)
   }
```

causes the value of i and j to repeatedly be converted to the string representations of the numbers. This of course drastically affects performance, and is undesireable.

Invent and describe one or more techniques that would significantly reduce the number of redundant string conversions in this example. For your solution(s), detail any language changes you make (if any), changes to the compilation phase that you would make, and changes to the run-time execution that you would have to make. Analyze the runtime savings produced by your technique(s).

B) In the string to integer (or real) conversion, if the string cannot be converted to a number (for example if it is the string "hello world"), a run-time error occurs. Describe enhancements you could make to the compilation phase to warn developers of possible run-time type violations like this. Consider that strings can come from at least two sources: constant strings in source code, and user input (from files, keyboard input, GUI input). As part of your answer, explain why it is impossible to identify ALL possible run-time type violations, and describe whether your solution will identify false positives – that is, it might warn that code is using non-numeric strings when in fact it always does use numeric strings.

4. (25 points)

Modern CPU's are increasingly featuring 2 (and soon 4) processor cores per CPU, making hardware concurrency a ubiquitous desktop feature. Suppose you are extending a sequential very high-level language (you could use Unicon or Tcl or Python or similar as an example in your solution) to take advantage of ubiquitous concurrency in order to increase performance.

A) Discuss how concurrency might best be introduced into the language without complicating its syntax or semantics for programmers.

B) Design a strategy for how to handle garbage collection safely in the presence of concurrent threads.