

Compiler

Qualifier Exam

August 16, 2005

This is a closed book exam. Answer all questions.

1. (25 points)

Consider the merits of an optimization called “procedure vectorization” in which an enclosing loop is moved inside a (custom version of a) method/function/procedure, so the code:

```
for(i=0; i<1000; i++)
    x += f(i);
...
int f(int i) {
    ...body of f
}
```

becomes

```
f_procvec(&x);
...
void f_procvec(int *x) {
    int i;
    for (i = 0; i<1000; i++)
        ...body of f, accumulating sum into *x
}
```

Discuss what information a compiler would have to know in order to allow this optimization, and what calculation or circumstances can determine whether this optimization would be beneficial for any given candidate location where it could be applied.

2. (25 points)

Some object-oriented languages support a feature called multiple inheritance in which subclasses may inherit from more than one superclass, a situation that occurs somewhat commonly in modeling and simulation. But many object-oriented languages explicitly disallow multiple inheritance. (a) Give an example

that shows why multiple inheritance can pose a semantic dilemma for language designers. (b) Sketch out a design for how an object-oriented language such as Java might implement multiple inheritance. Show what information must be represented in each instance, and what information is needed in the class, so that variables and methods can be inherited from multiple superclasses.

3. (25 points)

Tcl is a scripting language implemented with a simple bytecode compiler (that does not really do much program analysis but just compactly stores a straightforward representation of the program), and a relatively complex run-time interpreter. In Tcl, variables are not declared ahead of time, but simply used. A first use of a variable creates it. Variables are dynamically typed, but are always one of three types: string, integer, or real (double). The type of the variable is determined by both the value it is holding and how that value is being used. In other words, if a variable whose value is currently a string is used in an arithmetic expression, and if it can be successfully converted to an integer, it is converted and afterwards stored as an integer rather than the string, so that if it is used in an expression again, it does not need re-converted. The opposite conversion (integer to string) proceeds similarly.

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

```
set i 1
while {$i < 100} {
    incr i /* uses i as int (increments it by 1) */
    puts "i = $i" /* uses i as string (prints it out) */
    --more arithmetic use of i here-- /* use i as int */
}
```

might cause a phenomenon known as “type shimmering”, in which the value of *i* constantly shifts between the integer representation of the number and the string representation of the number. This of course drastically reduces performance, and is undesirable.

Describe at least two solutions to this problem. For each of the solutions, detail any language changes you make (if any), changes to the compilation phase that you would make, and changes to the run-time interpretation that you would have to make.

B) In the string to integer (or real) conversion, if the string cannot be converted to a number (e.g., its value might be “hello world”), and the variable is in an expression that needs a number, a run-time error occurs, generally aborting the script. 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 runtime 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)

Generational (or generation-scavenging) collectors improved the speed of classical garbage collection by up to a factor of five, observing that most data is short-lived and tenuring data once it has survived a collection or two. In this problem, you will consider a similar observation to reduce the amount of data allocated by a program P.

Suppose you notice that statistically, a significant percentage of many programs is spent on allocating and freeing memory, and further, a large percentage of allocated data is redundant – the same strings, numbers, and even structures such as lists are often allocated over and over during program execution. Design (include diagrams or pseudocode for algorithms as appropriate) a heap memory management subsystem (allocator and/or garbage collector) with the goal of avoiding redundant memory. Show pseudocode. Make your design as efficient as you can. Explain any constraints or limitations which might require that your heap contain redundant memory. Evaluate the time complexity of your subsystem's allocation and collection algorithms.