Name: ***Solutions***

100 points total. 4 hours, self-timed. Open book, notes, text, papers, etc.

Be concise and precise in your answers.

1) [5 pts] In the C shell, a command can be prefixed with `exec` to cause the shell to run that command directly, overwriting the process of the shell. This is so you don't have a silly shell process hanging around while you run the subsidiary command, if the shell will just exit when the command completes. What optimization, originally developed for languages like Scheme, is this like?

   tail call elimination

2) [6 pts] How does polyvariant specialization in partial evaluators relate to context-sensitive interprocedural analysis?

   Both analyze a callee procedure for different calling contexts. In PS, multiple specialized versions of a procedure may be produced, while in CSIA, a single callee procedure is usually compiled, but the callers can be optimized as if multiple routines had been compiled.

3) [6 pts] How does context-sensitive interprocedural analysis differ from *k*-CFA-style interprocedural analysis? What is the big advantage of context-sensitive analysis?

   k-CFA-style analyses repeat analysis of a callee blindly for each caller chain of length *k*; this can lead to analysis times that are exponential in *k*. Context-sensitive analysis, on the other hand, repeats analysis of a callee for each different domain element passed by a caller. Context-sensitive analyses can be a lot better in terms of avoiding unnecessary repeated analysis, in maximizing the available context w/ o fixing on *k* a priori, and in coping with recursion better.

4) [8 pts] To cope with compiling Scheme programs through C, Bartlett developed a partially-conservative garbage collector where pointers in the heap were known (through tagging), but pointers on the stack and in registers were ambiguous (since the actions of the C compiler were unknown). Bartlett's collector treated possible pointers in registers and on the stack conservatively, but used non-conservative techniques to deal with pointers once it started scanning the heap.

   Bartlett's system is a "mostly-copying" collector. Why is the "copying" part surprising? Why is it only "mostly"?

   It's surprising because conservative collectors aren't normally copying, since they can't change any pointers unless they're sure it's a pointer. Bartlett's system can copy objects pointed to only by heap objects, but not objects pointed to (possibly) from ambiguous roots on the stack or in registers. These objects must be pinned in place.

5) In lazy functional languages, evaluation of arguments to functions is deferred if and until the value is actually needed in the function or one of its callees. To implement deferred evaluation, compilers typically create a zero-argument closure whose body is the actual parameter expression and pass the closure in as the argument; the callee invokes the closure if the formal parameter's value is needed.

a) [9 pts] What are **three** different sources of costs that make this closure-passing implementation strategy slower than that used in eager functional languages like Scheme or ML?

Because of the allocation & deallocation costs of the closure, the invocation overhead of the closure (plus checking if the closure has already been invoked, in the case of memoized at-most-once lazy evaluation), and the cost in the caller of making the expressions referenced in the closure be accessible when the closure runs.

b) [6 pts] Why is the closure used to pass lazily-evaluated parameters cheaper to implement than regular user-level first-class closure?

Because the closure is LIFO: it will be accessed only during the callee, and hence it can be stack-allocated if desired. The enclosing environment also can be stack-allocated.

c) [12 pts] If the closure is guaranteed to be invoked (i.e. if the formal parameter is guaranteed to be referenced during all possible invocations of the callee), then it would be cheaper to evaluate the expression in the caller and pass just the resulting value rather than the closure. *Strictness analysis* is an interprocedural analysis that identifies which formal parameters of functions are *strict*, meaning that they are guaranteed to be evaluated during execution.

Assume you are given an intraprocedural analysis that computes for each program point in a procedure the set of variables that are definitely evaluated (later) during execution of the procedure, but which is conservative at call sites. Now define an interprocedural strictness analysis that extends the intraprocedural analysis. Explain the kinds of summaries you compute for each procedure, indicate the initial settings for the summaries, tell how you use the summary information when doing intraprocedural analysis of calls, give an efficient strategy for traversing the call graph (you may assume that you do not need to compute the call graph yourself), and describe how you cope with recursion without sacrificing precision unnecessarily. Argue for the correctness (i.e. safety) of your solution (assuming correctness of the intraprocedural strictness analysis), particularly in the presence of recursion.

I would summarize, for each formal, whether it was strict or non-strict. Initially, all formals are initialized to non-strict. When analyzing a call, an actual argument is considered needed iff the corresponding formal parameter is summarized as strict. Bottom-up processing through the call graph is best. Recursion just requires iteration during bottom-up processing. This solution is safe, because a formal is only marked strict when we've found a definite use of the formal, either directly in the function or indirectly through a call. By starting formals as non-strict, we won't hit the situation where a formal is only used in a recursive call, but we consider it strict because we have no reason to make it non-strict. (It is interesting that we don't make the optimistic assumption to initialize analysis, because that would lead to a fixpoint solution that was unsafe.)

d) [8 pts EXTRA CREDIT]: Define an abstract interpretation to perform intraprocedural strictness analysis. Use the following abstract syntax:

```
Expr ::= const | var |
         (primop Expr_1 ... Expr_n) |
         (if Expr_test Expr_then Expr_else) |
         (fn Expr_1 ... Expr_n)
```

Compute for each expression the set of variables definitely evaluated by the expression. This can then be used to ask the body of a lambda which of the formals of the lambda is definitely needed. An expression's value is needed if it is returned from the lambda, if it is the true argument of `if` (or needed on both then and else branches) and the `if`'s result is needed, if it is the argument of a primitive operator like `+` whose result is needed. An argument to a regular function call (*fn* above) is not needed, assuming only intraprocedural analysis.

F: Expr $\rightarrow 2^{Var}$

F [[*const*]] = {}

F [[*var*]] = {var}

F [[**(***primop* $Expr_1$ ... $Expr_n$**)**]] = F [[$Expr_1$]] $\cup$ ... $\cup$ F [[$Expr_n$]]

F [[**(if** $Expr_{test}$ $Expr_{then}$ $Expr_{else}$**)**]] =
F [[$Expr_{test}$]] $\cup$ (F [[$Expr_{then}$]] $\cap$ F [[$Expr_{else}$]])

F [[**(***fn* $Expr_1$ ... $Expr_n$**)**]] = {}

6) a) [10 pts] To schedule instructions well on a current superscalar machine, the scheduler must take into account the resource conflicts of the machine's functional units when striving to issue multiple instructions simultaneously. Consider a hypothetical machine that can issue two instructions in the same cycle, as long as the first instruction is an integer ALU instruction or a memory load or store and the second instruction is a floating point ALU instruction or a branch. (Hardware interlocks will delay the second instruction to the second cycle if the pair of instructions doesn't match this pattern.) How would you modify the heuristics in a list scheduler to try to schedule code for this machine?

This superscalar constraint is a kind of interlock. The first heuristic in choosing which candidate instruction to emit next is that it not interlock with the previous instruction. So we simply keep track of which instruction (column A or column B) we're trying to issue, and choose an instruction in that category, if possible. If multiple choices remain, then we go on to all the other standard heuristics of list schedulers.

b) [10 pts] More advanced code scheduling strategies can move instructions from one basic block to another, e.g. to separate loads from their downstream uses to better fit pipelined machines, to better pack instructions into groups for superscalar issue, etc. Describe the sort of dependence information you would want to gather to be able to determine when it was legal to move an instruction from one place to another. Make sure it's possible to move an instruction from below an if's merge to above its corresponding branch.

I'd want both data and control dependence information. It would be legal to move an instruction as long as it didn't violate either data or control dependence info. We'd need to ensure that instructions downstream of the if's merge are not control-dependent on the if's branch, only the then and else branch's instructions are.

Control dependence trees are a special representation for capturing the relative control dependence of basic blocks. In this representation, it becomes easy to tell when two blocks have the same control dependences, aiding in doing this sort of cross-block code scheduling.

7) Copying & compacting garbage collectors support fast allocation, conceptually just an instruction to bump the end-of-allocated-space pointer and a second compare-and-branch sequence to test whether we've run out of space. But for systems with very high allocation rates, e.g. in ML where even activation records are heap-allocated, we might like to optimize the compare-and-branch tests by coalescing multiple compares into a single comparison for a bunch of allocations.

a) [13 pts] Design an intraprocedural analysis to coalesce checks. What do you compute at each program point? [Hint: you want to know how much space you need to ensure is free for all the downstream allocations whose out-of-space checks have been eliminated.] What direction is analysis? What are the interesting flow functions? What is the merge function? What is the height of your lattice? How do you achieve termination of your analysis? Under what circumstances can checks be coalesced? What prevents checks from being coalesced?

I'd do a backwards analysis, computing at each program point an upper bound on how much space is needed for later (unchecked) allocations. I start out needing 0 space at the end of the procedure. Whenever I hit an allocation, I add the size of allocation, if it's known, and remove the check. If it's an array being allocated of unknown size, I insert a check for the size of the array + what I've computed so far, and then reset my counter back to zero before the array allocation. At function calls, I generate a check for the space I need after the call, and then reset my counter to zero. The merge function is max. Since the height of the Nat lattice is pretty big, I can't reasonably do this analysis around loops, so at loop heads I generate a check for whatever space is required and then reset the space-needed value to zero. (Loop heads thereby reach fixpoint at zero space needed.) Checks will be coalesced along non-looping paths that do not contain function calls or unknown-size array allocations.

b) [9 pts] Extend your coalescing analysis to work interprocedurally. Now what circumstances can be handled? Why might this interprocedural analysis be a bad idea?

This will let me be less conservative around function calls. I compute the space needed by the called function as a summary. I initialize all summaries to zero (to support recursive calls). I do a bottom-up analysis, w/o being dumb at function calls but rather using whatever the function consumes. When I encounter a procedure that's invoked recursively, since I've already assumed it needed zero unchecked space, I insert the appropriate check at the beginning of the procedure, and thus match my assumed summary value of zero. Function calls no longer pose an obstacle to this analysis; only loops, recursion, and unknown-size array accesses block check coalescing.

This analysis could be problematic, since small changes to the implementation of some procedures might force complete recomputation of how much space is needed.

c) [6 pts] What is the main reason why ML heap-allocates activation records?

Because it has first-class functions whose environments require heap allocation in general, so it's easy just to make the whole a.r. heap-allocated. ML also has first-class continuations, which essentially require pieces of the call stack to be preserved even after their creating procedure returns.