1) [5 pts] What is the difference between an optimistic iterative analysis and a pessimistic iterative analysis? What is the main advantage of an optimistic analysis? What advantage might a pessimistic analysis have?

An optimistic analysis starts by assuming the top domain element (the best possible information) and then iterates to weaken the information until a fixpoint is reached. A pessimistic analysis starts by assuming the bottom domain element (the most conservative possible information) and then iterates to strengthen the information until a fixpoint is reached. An optimistic analysis will reach at least as good and sometimes better fixpoint than a pessimistic analysis. A pessimistic analysis is a safe solution at all intermediate stages, even before fixpoint is reached, so it can be stopped in the middle of iteration if analysis is taking too long without giving up all information. Another benefit of pessimistic analysis, identified by one of the students, is that transformations can be performed eagerly during analysis w/o undo support, since their preconditions will never be violated by later iterations.

2) Both the Vortex compiler and the Sharlit system provide a toolkit for constructing data flow analyzers, based partially on the model of defining the important lattice operations and the flow functions for the problem being solved.

a) [5 pts] What does Sharlit provide in addition to this basic framework to improve the performance of the generated analyzers?

Sharlit includes path compression facilities, to gain the advantages of basic block summaries and even interval analysis, but in a nicely modular way.

b) [5 pts] Why is it beneficial to be able to interleave transformations with analysis, as supported by the Vortex framework? Why is this hard to do, in general?

Transformations can improve the information computed by analysis (e.g. constant folding improves constant propagation, and branch folding improves many analyses), without having to simulate the effect of the transformation as part of the analysis's flow functions.

It's hard to do transformation while trying to reach fixpoint of an analysis, as the transformations have to be either simulated or undoable. Non-local transformations or transformation that have non-local effects on the control flow graph (e.g. branch folding, code motion) are hard to handle.

3) [5 pts] Consider the following flow function for a simple assignment statement, implemented as part of an analysis computing the set of live variables at each statement:

$$\mathtt{LV}_{\mathtt{x := y\ op\ z}}\mathtt{:\ Info}_{\mathtt{pred}} = \mathtt{Info}_{\mathtt{succ}} \cup \{\mathtt{y,z}\} - \{\mathtt{x}\}$$

What is wrong with this function? Give an example where this flow function computes the wrong result.

It adds in the gen set before removing the kill set, which is backwards. It will compute that x is not live before `x := x + 1`, which is erroneous.

4) a) [5 pts] What are some optimizations that can be done solely with may-alias information?

Bounding the set of variables potentially affected by a store through a pointer (better side-effect analysis, for lots of analyses). Similarly, doing dead-store elimination, if all possible targets of a stored-through pointer are dead.

Performing more precise analysis of the result of a load instruction (e.g. for reaching definitions, constant propagation, and copy propagation), by limiting the results to the values stored in pointed-to variables. Similarly, replacing a load with the result of the load, if all the possible targets of the load contain the same variable/value.

Constant-folding pointer comparison operators, for the case where two pointers are known not to alias.

b) [5 pts] What are some optimizations that require must-alias information?

Doing a strong update of the target of a pointer store.

A load or store through a pointer can be replaced by a use or definition of the corresponding pointed-to variable, effecting a kind of copy propagation. If replacing a store, then this will enable strong updates.

Additionally constant-folding pointer comparison operators for the case where two pointers are known to alias.

Of course, must-alias info can do everything that may-alias info can do, but more precisely. Also, may-alias information where a pointer may-alias only one other thing is essentially must-alias information, and enables all the must-alias-specific optimizations.

5) Consider a block-structured language that supports exceptions and exception handlers. Each begin-end block can have an associated exception handler, for example:

```
begin
  ... raise larry; ... raise the_dead; ...
except
  when foo, bar: ...
  when larry, moe, curley: ...
  when others: ...
end
```

(Here the `raise` statements are assumed to be embedded in conditionals inside the begin-end block, so they're only conditionally raised in the body of the block.)

Exceptions are only raised explicitly via the `raise` statement, as shown above. Exceptions are handled by the `when` handler attached to the nearest lexically-enclosing block; the optional `when others` handler handles all otherwise unhandled exceptions. After the exception is handled, execution continues with the statement following the begin-end block to which the handling handler is attached.

Note: for this question, you need not worry about how the exception handling facility would be implemented, only about how its effects on control flow would be represented.

a) [5 pts] Assuming that exceptions are always handled within the procedures where they are raised, how would you extend the representation of a procedure to model the control flow effects of `raise` statements and exception handlers, so that iterative dataflow analysis can still be performed?

The raise statement should be implemented as an unconditional branch to the corresponding handler block. (The body of the begin-end block as well as each handler attached to the block ends with an unconditional branch to the end of the block after all the handlers.)

b) [5 pts] Now consider the case when an exception might not be handled by the procedure in which it originates. In this case, the exception is implicitly re-raised at the dynamically-enclosing call site. How would you extend your solution to handle this case as well, both on the callee and caller side? (You should assume that a procedure call can raise any exception.)

In addition to the techniques of part a:

Raises that aren't handled locally are represented as unconditional branches to the procedure exit node.

A procedure call now has multiple successors, one for the normal non-exceptional result as before, plus an edge to each enclosing handler, plus an edge to the procedure exit.

[These techniques support the right control flow properties, but are not directly executable. In addition, some mechanism needs to be implemented to actually represent which exception is being raised, and how a raised exception is passed to the appropriate handler.]

c) [5 pts] Imagine that you can define an interprocedural analysis to compute the set of expressions that might be raised by a procedure call. What are some direct and indirect benefits of this information?

[Uh, that should be “exceptions” above, not “expressions”.]

Direct benefits would be reductions in the exception routing support mechanisms, e.g. fewer or no exceptions would need to be tested for after a procedure call returns. Also, unreachable handlers can be dead-code eiliminated.

Indirect benefits include the simplified control flow due to removing edges corresponding to exceptions that can't be raised, which can improve the speed & precision of analyses.

6) [10 pts] The standard analysis for identifying loop-invariant calculations might determine that some conditional branch tests are loop-invariant. Describe a program transformation that can hoist loop-invariant conditional tests out of loops. Illustrate its effect on an example. What is its benefit? What is its cost?

After hoisting/copying all loop invariant calculations to the loop preheader, the loop body can be duplicated for each invariant branch in the loop body (up to 2^N loops for N invariant branches). The loop-invariant branches are hoisted out of the loops, turning into a decision tree to select a copy of the loop. Each loop copy would be specialized to the particular outcomes of the invariant branches leading to that copy, folding away the branches in the appropriate manner.

For example:

Original program:

```
while(...) {
   A;
   if (invar1) {
      B;
      if (invar2) {
         C;
      } else {
         D;
      }
      E;
   } else {
      F;
   }
}
```

Transformed program:

```
if (invar1) {
   if (invar2) {
      while(...) {
         A;
         B;
         C;
         E;
      }
   } else {
      while(...) {
         A;
         B;
```

```
            D;
            E;
          }
        }
      } else {
        while(...) {
          A;
          F;
        }
      }
```

The benefit of this transformation is both removing a conditional branch from the loop and also optimizing each loop copy with more precise information about that particular path through the loop-invariant conditionals.

The cost is code space blowup.

7) [15 pts] The Tiger language supports lexically nested procedures. As we discussed in class, this poses some complications for tracking definitions and uses of local and non-local variables. Define an interprocedural summary-based analysis that would enable reasonably precise tracking of possible uses and definitions of local and non-local variables. Is your analysis a summary of callees or callers or both? Is your analysis flow-sensitive or -insensitive? Context-sensitive or -insensitive? Computed bottom-up or top-down or neither? How do you handle recursion? How are the results of your analysis used to avoid making some worst-case assumptions?

I'd compute USE and MOD for each procedure, listing all non-local variables USEd or MODified by the procedure (and its callees). The information about a procedure ignoring callees is easy to compute in a single linear scan through the procedure. To handle a call site, the callee's two summaries are computed, variables in the summaries that are local to the caller are dropped, and the remaining variables (which are non-local to the caller) are added to the caller's two summaries.

The analysis is a summary of callees.

The analysis is flow-insensitive.

The analysis is context-insensitive.

The analysis is computed bottom-up. Recursion requires iteration, starting with the best possible info (empty USE and MOD sets).

Given a callee's USE and MOD sets, optimizations in the caller use those variables in the sets that are visible in the caller to avoid worst-case assumptions about e.g. live vars and about modified vars at the call site, just as for application of the USE and MOD sets discussed in lecture.

8) a) [5 pts] In what sense does a context-insensitive analysis still provide interprocedural information about calling context?

A context-insensitive analysis still computes a summary of all caller context information. It simply computes a single summary for all callers, as opposed to separate summaries for different (groups of) callers.

b) [15 pts] Describe how you would extend Callahan et al.'s interprocedural constant propagation algorithm discussed in class to be context-sensitive, using the model of partial transfer functions. What are your input and output domains? How do you construct an input domain element in callee terms from the information available before the call site in the caller? How do you compute the output domain element from the input domain element? How do you map back from the callee's output domain element to the caller's information after the call?

Basically, I'd define calling context to be which of the formals are constants, and if constant what constant values, and reanalyze the callee procedure for each different calling context. This is in contrast to Callahan et al.'s context-insensitive algorithm which computes only a single summary over all calling contexts.

Input domain: Tuple(ConstPropLattice), i.e. an element of the classic 3-level constant propagation lattice for each formal parameter.

Output domain: ConstPropLattice.

The input domain element used for a call site is just the tuple of constant-propagation domain elements corresponding to the actual parameters at the call site.

Computing the output domain element from the input domain element is just doing constant propagation & folding of the callee, using any of the jump function strategies in the paper, including some choice of jump function to compute the output domain element for the return value.

The output domain element is mapped back in the caller as the result of the message without change.

c) [5 pts] Why does it not make sense for an interprocedural USE analysis to be context-sensitive?

Because USE analysis is a bottom-up analysis, and doesn't depend on any caller information. Only for analyses that depend on calling context does it make sense to investigate context-sensitive algorithms.

d) [5 pts] What is the difference between context-sensitive analysis and procedure specialization? Would it be useful to perform context-sensitive analysis without performing procedure specialization?

Context-sensitive analysis is a kind of procedure specialization at analysis time, but it doesn't (on its own) generate multiple compiled versions of procedures. It can easily be useful to do context-sensitive analysis w/o procedure specialization, to provide result information for callers that depends on that caller's argument context, unsmeared by other callers of the same callee.