Project 4: Tiger Language Control Flow Analysis

Dong Son Trinh
Maxim Mints

High-Level Control Flow Analysis Architecture:

The control flow analyzer mainly consists of the class CFG (p4/src/com/tiger/controlflow/CFG.java). It represents a control-flow graph which can be created from a single function or the main statement sequence, and it can detect common subexpression elimination candidates and dead code (non-branch) statements.

The other class used is CFGAnalyzer (p4/src/com/tiger/controlflow/CFGAnalyzer.java). It makes use of the AST created for Project 2. An AST generated by the parser can be passed to this class. It then uses the class CFG to construct a CFG for every procedure and the main statement sequence in the AST. It can then use these CFGs to find the subexpression elimination candidates and dead code in the entire program by calling the respective methods in each CFG.

Each CFG instance receives (from the CFGAnalyzer that instantiates it) an ASTNode representing a sequence of statements, along with a list of names of initial criticals, which are the set of global variable names without its intersection with the currently processed function's parameter names. These are used in dead code detection, which is described later. A CFG converts the sequence of statements to a control-flow graph represented by its inner class, CFGNode. Each CFGNode represents a single program statement. We do not combine them to basic blocks since it reduces some of the complexity of analysis, and efficiency is not a big concern for this project.

Low-Level Design Decisions:

The conversion of a statement sequence to a CFG happens recursively: the statements closer to the end of the sequence are added first as the terminating condition of the recursion, and then they are linked to the preceding statements as the recursion unfolds. It also keeps track of the statement succeeding the current loop so that we know where to link break statements (return statements are linked to a "sentinel" faux-CFGNode representing the end of the CFG). CFG also marks its backedges. Each CFGNode has a field with the ASTNode of the statement it represents. Branches are represented by CFGNodes with an ASTNode representing an expression instead of a statement, and have two outgoing edges instead of one – one for the condition being true, the other for the condition being false. Branches are used to represent "if" statements. "While" loops are represented as a branch which, if true, goes to the loop body, and, if false, goes to the next statement after it. The last statement or statements (this is important, for example, there is no "sentinel" node at the end of if-else statements, so there may be multiple paths leading to a loop body end) of the loop body have the loop branch condition as their next node, which is a back-edge. "For" loops are similar, using a loop with a condition of the counter variable being less than or equal to its maximum value, however, they have a loop counter increment added at the start of the loop body (since our implementation does not analyze the possibility of branches being taken or not, this is not really important, however, this conforms to the design of having the for loop be strictly inclusive, which is stated in the Tiger language specification). Additionally, two statements are added before the loop body: one to set the initial value of the loop counter, and another to decrement it once (note that the latter may get marked as dead code if the loop counter is overwritten by some other variable, which is because control flow dependence on values is not taken into account in our dead code analysis; then it will show up in the output even though it does not exist in the original program). This is needed to offset the decrement happening in the beginning of the loop. Therefore, the original code in and after the for loop will only see values in the expected range, inclusive.

Dead code elimination happens after the CFG is constructed. We do not convert the CFG to an SSA form, but we perform what is essentially the same computations when we look up the "closest definition" of a variable. So, we start at the set of critical variables, which, as the CFG is constructed, is further increased from its initial state by adding the variables used by a return statement, if any. For each of these values, we recursively go up the CFG until we find its "def", i.e. closest definition, i.e. the definition which is the last one above the current point in the CFG to not be overwritten by another statement below it but above the current point in the CFG. Here, we do not account for assignments which do not occur in all branches leading up to the mentioned point. The "def" can actually be multiple assignments, if there are multiple paths in the CFG converging to the current one, all with the value being assigned. Then we can take a list of all of their arguments, which would correspond to the arguments of a phi-function in the SSA. We can perform the same operation with the lists of arguments of these "def"s as what we did with the original critical values. Along the way, for each "def" we find, the corresponding statement is marked as "useful". If a node has already been marked as "useful", and we find it in a "def", we do not process its arguments, since this has already been done. Also, after processing the critical statements, we repeat the procedure for all arguments of all function calls encountered in the sequence of statements, starting the search upwards from the statement where the function call is defined (since a function call cannot be dead, and so all values as they are used in it cannot be dead even if they are subsequently overwritten). After we've finished the marking, we "sweep" by printing all non-marked statements as dead. We make exceptions for condition heads and break statements (both the no-op ones and the normal ones), since we choose not to mark entire branches as dead code if their contents only modify variables which are reassigned later anyway, since this would require more complex analysis.

As the CFG is constructed, all processed statements are added to one list. Also, for each node, the downward-exposed expressions (DEExpr-s) are computed and represented as a list of tokens (terminals) used in the right-hand side of the assignment, if the statement is an assignment. Common subexpression elimination then proceeds by initializing the "Avail" set of each state to an empty set and then going through the list of all statements, repeatedly recomputing the Avail sets using the Avail sets of their parents and using a method of each parent CFGNode to eliminate expressions killed by that parent node from the Avail set of the current node. This computation is performed, repeatedly looping over the array, until the Avail sets stop changing. This can be shown to converge to the real Avail sets of a program's statements. Note that the statements which are recorded as dead are not handled, that is why dead code elimination is normally scheduled to happen before CSE. Then, if an expression which is in an assignment statement's Avail set is in its right-hand side, this statement is a candidate for CSE and output to the screen. In this, constants are ignored. This comparison is simplified by the fact that our input is constrained to have no more than two operands per expression and that we assume our expressions are not commutative, which reduces this operation to a direct comparison between sets of tokens. In fact, due to this, for the purposes of CSE elimination expressions are mostly represented as sequences of tokens, which uniquely identify them.

Challenges and Issues:

This has been by far the most complicated project in this class, with more intricacies and edge cases than anything else we had to program for the Tiger language. The edge cases were certainly the biggest problem, as we had to constantly add extra conditions for statements to be marked as "useful" or CSE and then making sure that this didn't break any existing code. Also implementing the search for "def"s was quite taxing, since it requires complex handling of loops, where you have to only go through the loop body once as if it is a conditional block which might or might not be taken; this is why we needed to mark backedges.

Testing:

The type checker has been tested against the test cases provided on GitHub. We also chose several of the provided Project 1 and 2 test cases. Additionally, Dong Son created several custom comprehensive test cases to test the many edge cases, which can be found on our GitHub.

Unresolved Issues:

We do not perform any analysis of branch conditions to determine if a branch is taken or not. We also chose not to mark entire branches as dead code if their contents only modify variables which are reassigned later anyway, since this would require more complex analysis.
Generally, we do not analyze the dependence of control flow upon variables; thus, if we have a for loop and we do not use its counter variable in non-dead code in the body and if we then overwrite the counter right after the for loop ends, then the increment, initial value and the initial decrement (discussed above) are always marked as dead code. However, I believe none of this was strictly required. Otherwise there were no issues that we could find.