

Compiler Design Document

Dataflow Analysis and Optimization Checkpoint

Team: Human suffering (@ajyliew, @emshen, @skoppula, @srobin)

This design document covers the implementation details of our Decaf compilers control flow graph (CFG) generation, data flow analysis on the CFG, and implementation of our first optimization, common subexpression elimination. Our three main high-level sections are organized accordingly. Throughout the document, we also highlight the structure of our codebase, and when relevant, specific parts of our compiler's source, Graphviz visualizations of our CFGs, and output x86_64 assembly.

1. Control Flow Graph Generation

The first step in creating a dataflow analysis framework was to create a control flow graph connecting basic blocks. As described in our codegen design document, previously we had translated our high-level IR into a low-level IR that consisted of a linear list of Three Address Codes (TACs) that directly converted, TAC by TAC, into lines of assembly (see Appendix Figure A). This circumvented CFG generation completely, but was not particularly farsighted.

We replaced our module that translated our high-level IR tree to a TAC list (`compile/tac/TacGen.scala`), with a module that instead translated high-level IR to a CFG (`compile/cfg/CFGGen.scala`). A new set of procedures (`cfgToTacs(...)` and `tacsToAsm(...)` in `compile/cfg/CFGUtil.scala`) traverse the CFG and generate a linear list of TACs (paired with the corresponding scopes) for conversion to assembly.

There are details that complicated our CFG generation, and we will not describe them in detail here for sake of brevity¹. But some important aspects of our CFG that will be useful to know if we reference them when describing our optimizations in the following sections:

- We have four types of blocks in our CFG; there are blocks to represent branches (`compile/cfg/BasicBlock.scala/BranchBB`), blocks with exactly two parents to represent control flow merges (`compile/cfg/BasicBlock.scala/MergeBB`), blocks to represent non-control flow instructions (`compile/cfg/BasicBlock.scala/NormalBB`), and jump destination blocks that have multiple parents because of break/continue's (`compile/cfg/BasicBlock.scala/JumpDestBB`).

¹ We implemented a few auxiliary procedures to obtain a correct CFG and prepare for dataflow optimization: methods to avoid traversing basic blocks repeatedly on loop backs or jump forwards and templize CFG traversal, compress the CFG so each basic block is maximal, fix codegen errors in break/continue statements from our last turn-in, reconstruct parent pointers based on child pointers, maintain a basic block ID to basic block object map (useful for determining dataflow bitvector size, among other things), and integrate Graphviz for CFG visualization (and more!).

- Inside each basic block is a list of instructions, represented as an `ArrayBuffer[TAC]`. The buffer is populated during CFG creation time, while traversing the high-level IR tree. This stored list of TACs inside each basic block is how we convert an entire CFG tree to a linear list of TACs.
- Furthermore, each basic block stores a pointer to the current scope in the form of a `SymbolTable`. This is important because a TAC does not contain scope information (a TAC is composed of just two or three variables names and an operation).
- Each basic block is assigned a unique ID by using *BasicBlockGenie* (`compile/cfg/BasicBlockGenie.scala`). This ID is for useful, among other things, to read basic block pointer locations in debug dumps and quickly check basic block equality.



Integration with Graphviz allows us to visualize our CFG (and helps immensely with debugging!). Part of an example CFG is shown below (Figure 1).

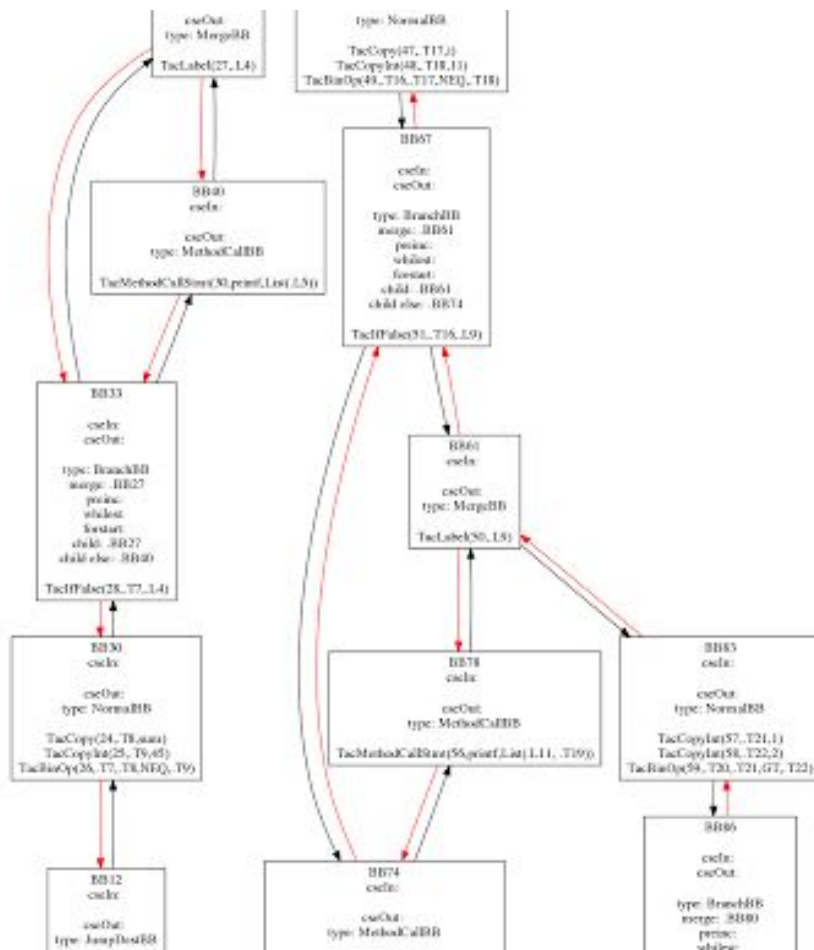


Figure 1. Part of the CFG for `tests/codegen/input/06-control-flow.dcf`

2. Symbolic Variable and Expression Representation

We store the results of expressions into temporary variables on the stack. However, because temporary variables are used once and discarded, most optimizations (such as common subexpression elimination) work best across symbolic variables and not their corresponding temporary variables. Thus, we found it necessary to keep track of the relationship between a temporary variable and its symbolic variable representation (variables in the original Decaf source).

Thus after CFG generation, we iterate over the TACs in each block and extract the TacCopy instructions that copy a symbolic variable into a temporary variable. Figure 2A details this process in an example.

```
Decaf
C = X + V + Z

Generated TACs
TacCopy(".T1", "x")
TacCopy(".T2", "y")
TacBinOp(".T3", "t2", "t1")
TacCopy(".T4", "z")
TacBinOp(".T5", "t3", "t4")

Generated Temp-To-Symbolic-Variable Map
Map(
  ".T1"->("x", [Symbol Table containing x]),
  ".T2"->("x", [Symbol Table containing y]),
  ".T4"->("x", [Symbol Table containing z]),
)
```

Figure 2A. Example of generation of the Temp-to-Symbol Variable map

Since it is possible for variable names to be reused in different scopes, we query the basic block's symbol table to find the innermost scope containing the variable, and save that along with symbol variable name in the map. We similarly represent expressions (binary or unary operations between one or more symbolic variables) with the following signature:

```
case class Expression (
  op: OpEnumVal,
  setVars: Set[(String, SymbolTable)],
  listVars: ArrayBuffer[(String, SymbolTable)]
)
```

Figure 2B. Definition of an *Expression*

`setVars` and `listVars` contain the same (symbol variable) elements, but depending on whether the operation is commutative, we apply the correct `.equals` when in `Expression`'s `.equal` method. This will be used as the object of our fixed point algorithm, as described in the next section.

3. Dataflow Availability Analysis

We implemented the worklist algorithm presented in class in the following manner.

Our analogue to the *in* and *out* bitvector is a map of temporary variables to symbolic expressions that are available for re-use². Each block has an *in*-map and an *out*-map, which are empty at the start of the fixed point algorithm.

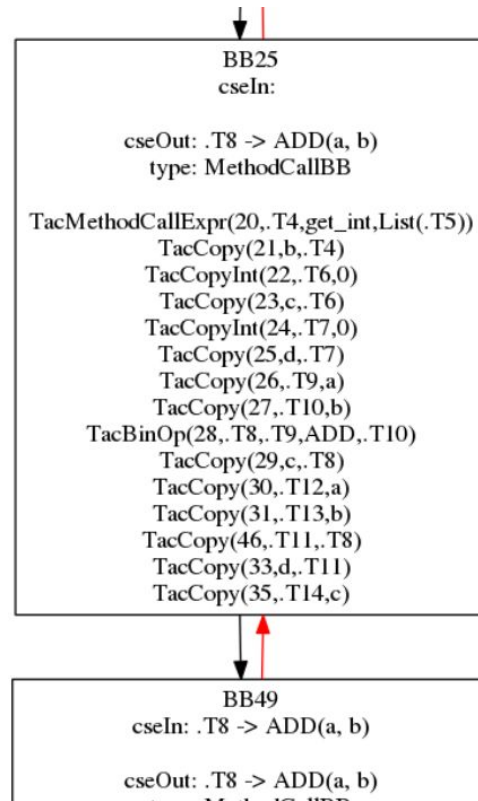


Figure 3. Running availability fixed point algorithm on `tests/dataflow/input/cse-01.dcf`. The *in*-map is empty, but because of the evaluation of `TacBinOp(28, .T8, .T9, ADD, .T10)`, a new expression is available for re-use. You can see this re-usable expression added to the *out*-map. The expression is passed onto the next basic block in temporary variable `.T8`.

Our transformation function is as follows. For every `TacBinOp` and `TacUnOp` in the block, we add the corresponding temporary to expression mapping (our analogue to *gen*). For every TAC

² By symbolic expression, specifically we mean an `Expression` object, as described in section 2.

which overwrites a temporary variable³, we use the *Temp-to-Symbol Variable Map* previously described to find the temporary variable's symbol variable, if it exists. If so, that means the symbol variable is being overwritten, and any expression using the variable is no longer available (our analogue to *kill*). We remove any expression in our *in*-map that has the overwritten symbol variable. The *out*-map is the state of the map after processing the last TAC in the block.

When a block has multiple parents, we compute the *in*-map by taking the intersection of the two maps (our analogue to *join*). Note that this means if the same expression is mapped to multiple temporary variables, then we do not import the expression to the new *in*-map.

An interesting detail is that method calls (every `TacMethodCallExpr` and `TacMethodCallStmt`) may modify global variables, and thus may kill mappings in *in*-map. To handle this, we pre-process every method's CFG and store a list of global (symbolic) variables that are modified. For every method call, we kill the appropriate expressions.

We iterate until the analysis reaches a fixed point for every *in*-map and *out*-map (`runCSEFixedPointAlgorithm(...)` in `compile/analysis/CSE.scala`).

4. Common Subexpression Elimination

With the *in*-map and *out*-map set for every basic block, we iterate through every block, looking to see if common subexpression elimination is possible. If a TAC in a basic block is either a `TacBinOp` or `TacUnOp`, there is a possibility that we could replace the operation with a `TacCopy`, using an available expression in *in*-map.

We convert the `TacBinOp` or `TacUnOp` to the corresponding expression in symbolic variables, and query the expression in *in*-map. If the expression exists, we replace the TAC. Figure 4 illustrates an example of our inter-block common expression elimination (`substituteCSEInBlock(...)` in `compile/analysis/CSEUtils.scala`).

The subexpression elimination just previously described was using available expression from a previous basic block; in addition, we also carry out common subexpression elimination between instructions within a single basic block (`substituteCSEIntraBlock(...)` in `compile/analysis/CSEUtils.scala`). This handles the case in which an expression is created and destroyed within the same block, but re-use is still available between instructions within the block.

³ `TacCopy`, `TacMethodCallExpr`, and `TacCopyInt` are three examples of TAC types that overwrite a temporary variable.

```

int get_int ( int x ) {
    return x;
}
void main ( ) {
    int a, b, c;
    a = get_int ( 2 );
    b = get_int ( 3 );
    c = 0;
    c = ( a + b ) * ( a + b );
    printf ( "%d\n", c );
}

```

Figure 4A. For reference, tests/dataflow/input/cse-02.dcf is printed above. We illustrate how our CSE removes one extra ALU operation in this example.

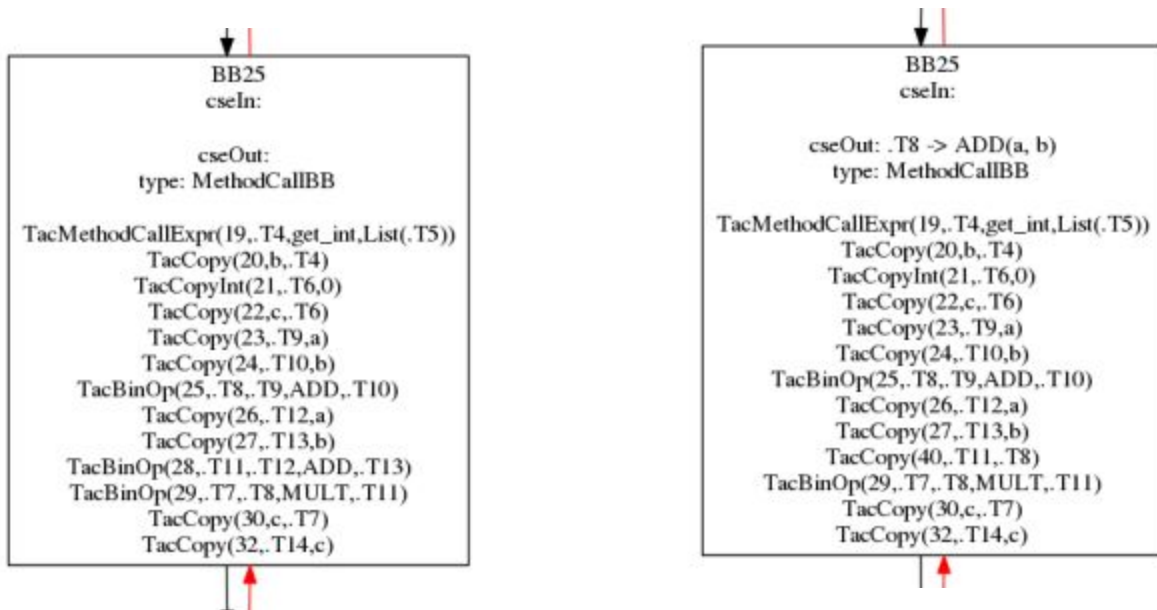


Figure 4B. Part of the CFG created when compiling cse-02.dcf. To the left is the result with common subexpression elimination turned off. On the right, common subexpression elimination was turned on. Notice the elimination of one TacBinOp, replaced with a TacCopy, re-using the available expression $a+b$.

```

skoppula:compilers> diff cse2.s cse2_opt.s
92,95c92,93
<      movq    -112(%rbp), %r10
<      movq    -120(%rbp), %r11
<      addq    %r10, %r11
<      movq    %r11, -104(%rbp)
---
>      movq    -80(%rbp), %r10
>      movq    %r10, -104(%rbp)

```

Figure 4C. diff between the optimized and unoptimized assembly. We saved an ALU operation!