Advanced Compiler Design

Spring Semester 2016

Homework 1

Due date: March 24, 2016, 23:59

1 Introduction

In the previous homework you extended the Javali compiler by adding Control Flow Graph (CFG) construction. The skeleton in your subversion repository already includes CFG construction, so you can base your solution to this homework on the skeleton if you did not do the previous homework. (Or you could merge your solution to Homework 0 into this skeleton.)

The objective of this homework is to develop a compiler that uses Static Single Assignment (SSA) form as intermediate representation (IR). Future homework will ask you to enhance this compiler and to include SSA-based optimizations.

There are four basic steps to be implemented in this homework:

1. Computation of the dominator tree (DT).
2. Introduction of Φ-operations into the control flow graph, using the approach for arbitrary programs.
3. Renaming of the variables to ensure that the correct version of a variable is used.
4. Replacing Φ-operations with equivalent assignments.

2 Step 1: Control Flow Graph and Dominator Tree

The construction of CFG and DT is placed into a separate phase of the compiler, which executes after the semantic analysis. Here is the recommended sequence of steps that the compiler should do in this phase:

- In the class `cd.transform.Dominator`, compute the immediate dominator of each basic block N that appears in the CFG.

  The class `cd.ir.ControlFlowGraph` is used to represent CFGs. One `ControlFlowGraph` instance is associated with each `cd.ir.Ast.MethodDecl` instance (a new field, `cfg`, has been added to `MethodDecl` for this purpose). The class `cd.ir.BasicBlock` is used to represent basic blocks. The immediate dominator of a basic block should be stored in the `dominatorTreeParent` field of the class `BasicBlock`.

- In the class `Dominator`, construct the dominator tree (one tree for each method). The children of a `BasicBlock` in the dominator tree should be stored in the field `dominatorTreeChildren`.
- In the class `Dominator`, compute the *dominance frontier* for each block in each method. The dominance frontier of a `BasicBlock` should be stored in the field `dominanceFrontier`.

  See the dragon book [Dragon] or [Cooper et. al.] for algorithms to compute dominators.

3 Step 2 and 3: SSA Form

In `cd.transform.SSA`, the following steps are to be done for each method:

- For each scalar local variable, determine the set of blocks where a Φ-operation must be inserted. Start with the set of blocks B_i that set the variable and then insert a Φ-operation at the blocks that are a member of DF(B_i). Repeat until no more Φ-operations must be inserted. See [Appel], [Cytron et. al.], [Muchnick] for more details.
- Introduce variable version numbers. You should create a new `VariableSymbol` for each distinct version, and add that symbol to the `locals` field of the corresponding `MethodSymbol`. To keep track of the various versions, you have to use a stack, as explained in class and in the literature.

These two steps convert the IR into SSA form (and we can show that the SSA form is minimal, i.e., there are no unnecessary Φ-operations).

4 Step 4: De-SSA

We have provided a code generator (`cd.backend.codegen.CfgCodeGenerator`) that uses the CFG to generate code from. However, it cannot handle SSA form. Therefore, the compiler first has to "destruct" SSA form again. Implement this step in the `cd.transform.DeSSA` class. This can be done by converting each Φ-statement into copy statements, as discussed in class:

- Each Φ-operation is translated into multiple of assignments, which are inserted at the *end* of the direct predecessor blocks. See [Appel] for more information.

  It is easy to insert these assignments if no node with multiple direct successors has an edge to a node with multiple direct predecessors. We recommend that you ensure that this property holds while constructing the CFG, if necessary by inserting a block with an empty assignment (a NOP).

Test your compiler to demonstrate that it translates programs correctly. You do not have to worry about copy operations that you may consider unnecessary. The objective is to get a working, correct compiler—not to produce the fastest code possible. Future optimizations will take care of many of these copy operations.

References

[Appel] Andrew W. Appel. Modern Compiler Implementation in Java. Cambridge University Press, 1998.

[Dragon] Compilers: Principles, Techniques, and Tools (2nd edition). Addison Wesley.

[Muchnick] Stephen S.Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.

[Cooper et. al.] K. Cooper, T. Harvey, and K. Kennedy. "A Simple, Fast Dominance Algorithm". `http://www.hipersoft.rice.edu/grads/publications/dom14.pdf`

[Cytron et. al.] R. Cytron, J. Ferrante, B. Rosen, M. Wegmann and F. K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". ACM Trans. on Prog. Lang. Syst., Oct. 1991, 13(4), pp 451–490