Finding Dominators

CSE 501 Spring 2013 Compilers Assignment 1

Andre Baixo, Jacob Nelson

April 17, 2013

This writeup describes our optimizer implmentation for Assignment 1.

1 Implementation

Our optimizer implements the dominance algorithm described in **??**.

Programs are processed in four phases. In the first phase, we parse the textual representation of the intermediate language output from the Start front-end and create an object for each element found. Non-executable elements like the *method* construct are stored separately from the executable instructions.

In the second phase, we form the instructions into basic blocks. To do this, we use the *method* elements in the input to partition the set of instructions; then we iterate over the instructions in each method's set and divide them into basic blocks based on branch (and other) instructions as well as branch targets.

In the third phase, we form the control flow graph by adding edges between the basic blocks based on program order as well as branch targets. We store both successor and predecessor edges to make traversal cheaper.

Finally, we compute dominators for each method. The first step here is to compute a topological order over the basic blocks in the method. We do this by running a depth-first search over the CFG. Once we have that order, we execute the dominance algorithm, iteratively refining each block's immmediate dominator until nothing changes. When that is complete, we store the immediate dominators in the blocks and add the dominator tree edges to the CFG.

2 Examples

We implemented three additional examples to better understand our implementation. The first is `examples/hammock.il`, a simple dual hammock shown in Figure 1. The second is `examples/weave.il`, a more complicated set of woven branches shown in Figure 2. Both of these examples provide a simple verification of our dominator construction.

The third example is a script that generates nested if constructs of different sizes. The Ruby file `examples/nested-if.rb` takes two parameters: a total if-construct count, and a nesting depth. The nesting depth controls how many levels of nested if constructs are generated. Figure 3 shows a CFG from a 10-node, singly-nested example; figure 4 shows a 10-node, 5-nested example, and figure 5 shows a 10-node, 10-nested example. This lets us explore how the structure of the CFG affects runtime.

3 Results

We present results for our nested

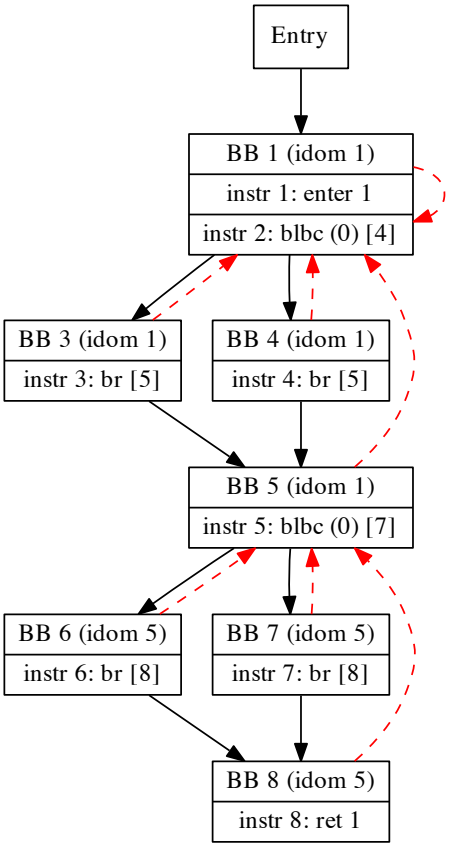

Figure 1: CFG from hammock example. Red edges show dominator relationships.

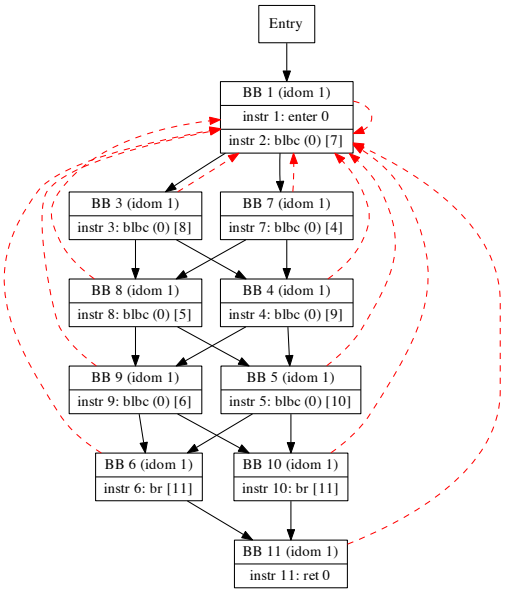

Figure 2: CFG from weave example. Red edges show dominator relationships.

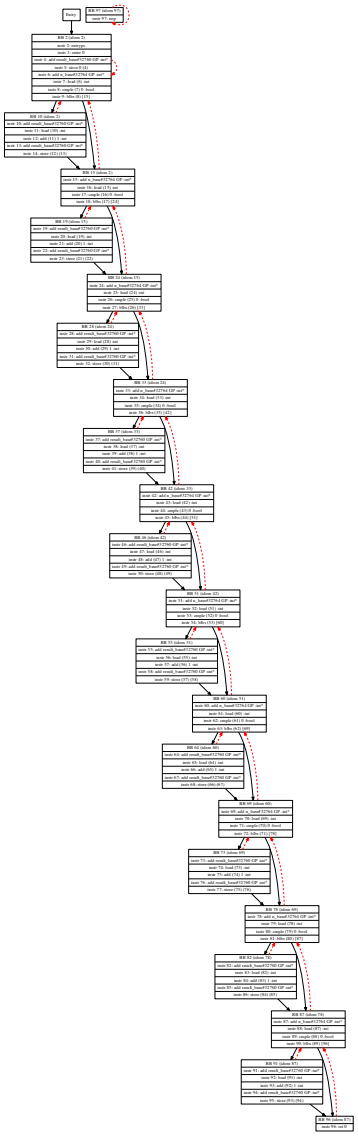

Figure 3: CFG from 10-if singly-nested example. Red edges show dominator relationships.

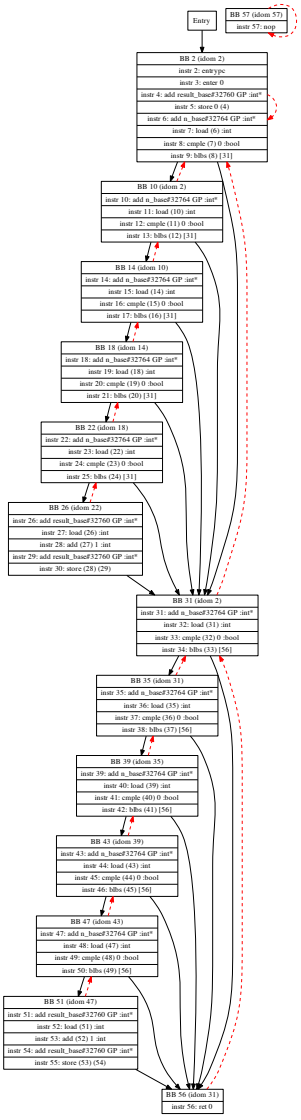

Figure 4: CFG from 10-if 5-nested example. Red edges show dominator relationships.

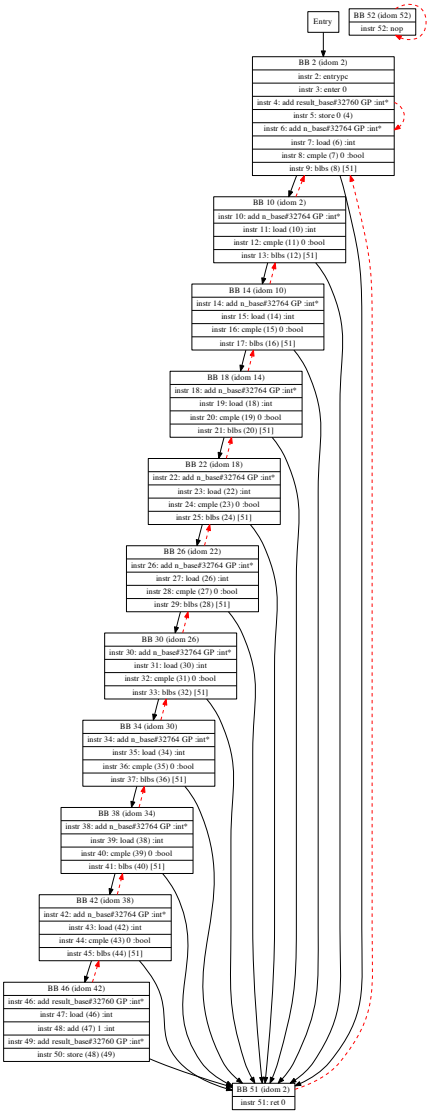

Figure 5: CFG from 10-if 10-nested example. Red edges show dominator relationships.