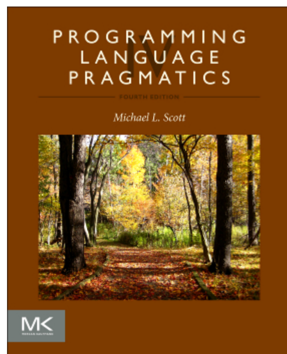
Code Optimization, Day 2: Global Optimization

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 17

Ben Titzer Prof. Jonathan Aldrich

Copyright © 2016 Elsevier and Jonathan Aldrich

Redundancy Elimination in Basic Blocks

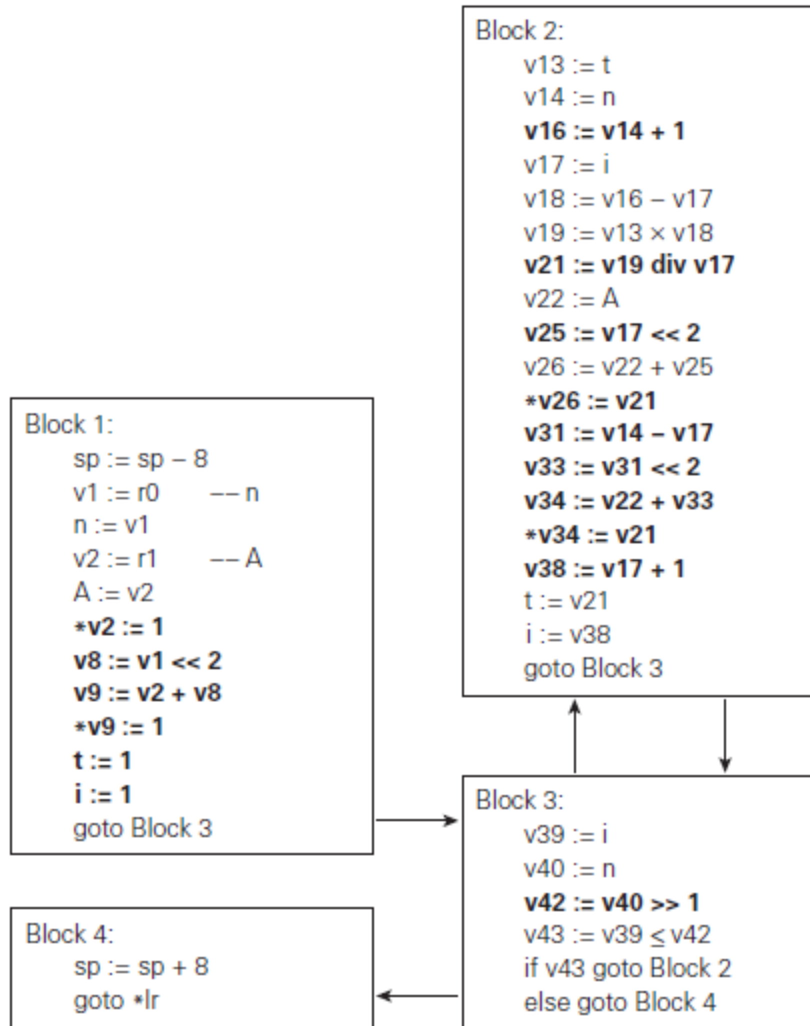


Figure 17.4 Control flow graph for the `combinations` subroutine after local redundancy elimination and strength reduction. Changes from Figure C-17.3 are shown in boldface type.

SSA and Data Flow Analysis

- We now concentrate on optimizations across the boundaries between basic blocks
- We translate the code of our basic blocks into *static single assignment* (SSA) form
- SSA form generalizes local optimizations that previously worked locally to be global:
  - Constant propagation
  - Constant folding
  - Strength reduction
  - Common subexpression elimination
  - Loop invariant code motion
  - Dead code elimination

SSA and Data Flow Analysis

- In today's compilers the translation to SSA form is driven by data flow analysis and essentially all global optimizations are based on it
  - We detail the problems of identifying
    - common subexpressions
    - redundant loads and stores
  - SSA solves the problem of *reaching definitions* once and all subsequent passes benefit
  - Further passes may use additional dataflow analysis to model the heap, available expressions, and/or side-effects
- Global redundancy elimination can be structured in such a way that it catches local redundancies as well, eliminating the need for a separate local pass

SSA Form

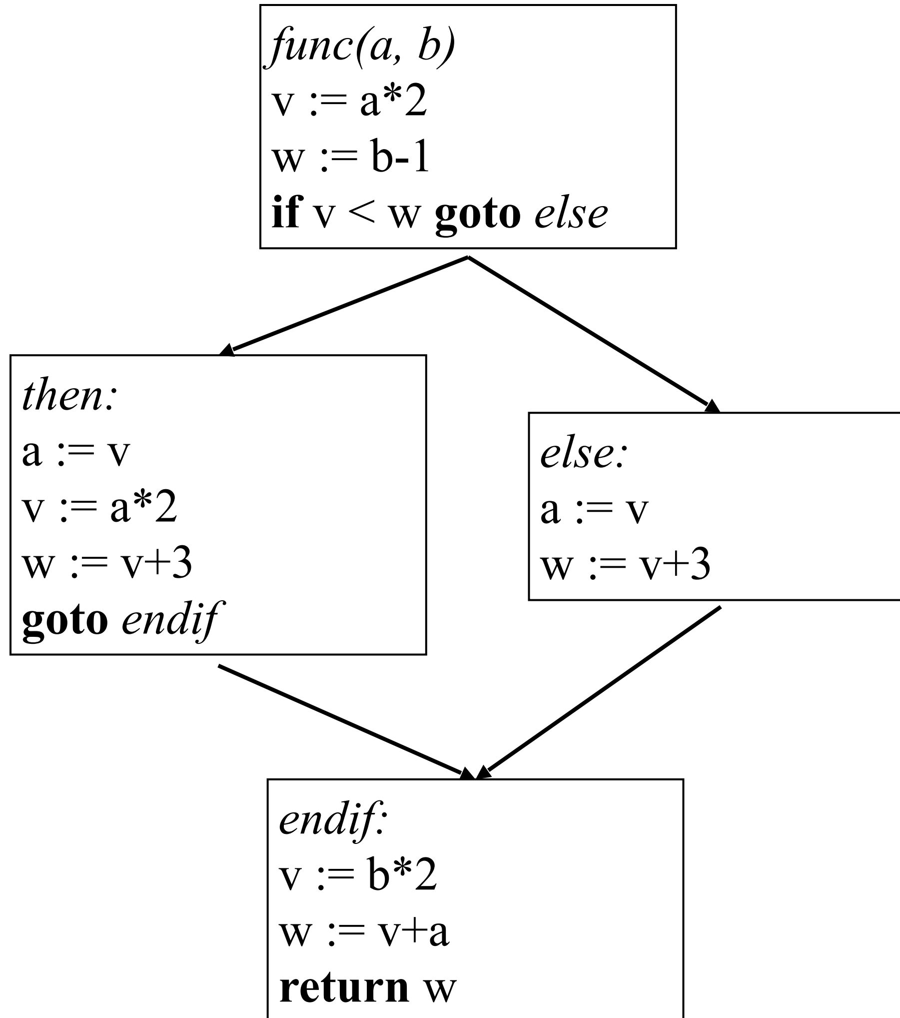
- Static single assignment: every variable assigned only once — uses at merges appropriately renamed
- For example, if the instruction `v2 := x` is guaranteed to read the value of `x` written by the instruction `x3 := v1`, then we replace `v2 := x` with `v2 := x3`
- An IR construct called ϕ *(phi or phi node)* chooses among the possible alternatives at control merges
  - ϕ-nodes exist only at compile time
    - Optimization aide that will be removed late in compilation, e.g. when generating target code
- With SSA, *reaching definitions* of a variable is trivial

ELSEVIER

Conversion to SSA form

- SSA form is primarily a renaming of local variables and introduction of ϕ nodes
- Can be formulated as a dataflow problem (i.e. fixpoint calculation), but most compilers employ a custom renaming pass designed to be efficient
  - Maintain a worklist of basic blocks, each with a mapping from variable to most-recent version (e.g. v = v3)
  - Select a block from the worklist whose predecessors are all finished (if none, there is a loop) and rename in a forward pass, push successors
  - Introduce ϕ nodes for any variable whose version differs between predecessors
  - Every block is visited once, but result is *not optimal*
    - Every compiler book you read tells you this is too inefficient, but nearly all industrial compilers work this way

Example: Convert this to SSA



- You can just update the variable subscripts and introduce ϕs, no need to rewrite the code

Solution: Convert this to SSA

```
func(a, b)
v1 := a1*2
w1 := b1-1
if v1 < w1 goto else
```

```
then:
a2 := v1
v2 := a2*2
w2 := v2+3
goto endif
```

```
else:
a3 := v1
w3 := v1+3
```

```
endif:
a4 := φ(a2,a3)
v3 := φ(v2,v1)
w4 := φ(w2,w3)
v4 := b1*2
w5 := v3+a4
return w5
```

- You can just update the variable subscripts and introduce ϕs, no need to rewrite the code

Value Numbering and SSA Form

- Value numbering, as introduced earlier, assigns a distinct virtual register name to every symbolically distinct value in the IR
  - It allows us to recognize when certain loads or computations are redundant.
- The first step in *global* value numbering is to distinguish among the values that may be written to a variable in different basic blocks
  - We accomplish this step using static single assignment (SSA) form

Value of SSA Form

- All definitions *dominate* their uses
  - A block D dominates a block X if all paths from start to X pass through D
- Almost all local optimizations performed by a compiler get more powerful with SSA
  - Constant propagation
  - Constant folding
  - Strength reduction
  - Common subexpression elimination