A generic worklist analysis algorithm

Maintain a mapping from each program point to info at that point
- optimistically initialize all pp's to T

Set other pp's (e.g. entry/exit point) to other values, if desired

Maintain a worklist of nodes whose flow functions needs to be evaluated
- initialize with all nodes in graph

While worklist nonempty do
  Pop node off worklist
  Evaluate node's flow function,
    given current info on predecessor/successor pp's,
    allowing it to change info on predecessor/successor pp's
  If any pp's changed, then put adjacent nodes on worklist
    (if not already there)

For faster analysis, want to follow topological order
- number nodes in topological order
- pop nodes off worklist in increasing topological order

It Just Works!

Advanced program representations

Goal:
- more effective analysis
- faster analysis
- easier transformations

Approach:
  more directly capture important program properties
- e.g. data flow, independence

Examples

CFG:
- \+ simple to build
- \+ complete
- \+ no derived info to keep up to date during transformations

- – computing info is slow and/or ineffective
  - lots of propagation of big sets/maps

Def/use chains

Def/use chains directly linking defs to uses & vice versa
- \+ directly captures data flow for analysis
  - e.g. constant propagation, live variables easy

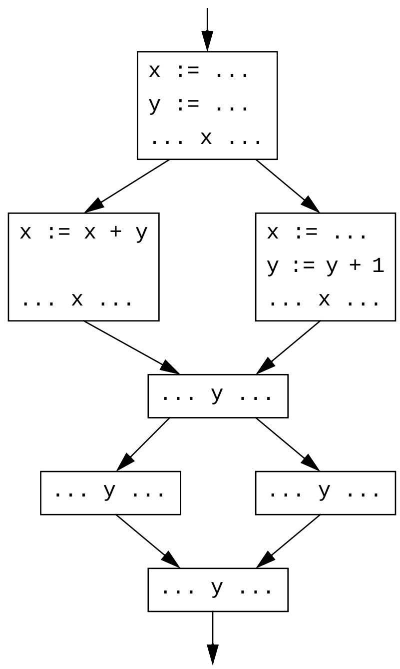
- – ignores control flow
  - misses some optimization opportunities, since it assumes all paths taken
  - not executable by itself, since it doesn't include control dependence links
  - not appropriate for some optimizations, such as CSE and code motion

- – must update after transformations
  - but just thin out chains

- – space-consuming, in worst case: $O(E^2V)$

- – can have multiple defs of same variable in program, multiple defs can reach a use
  - complicates analysis

Example



Static Single Assignment (SSA) form

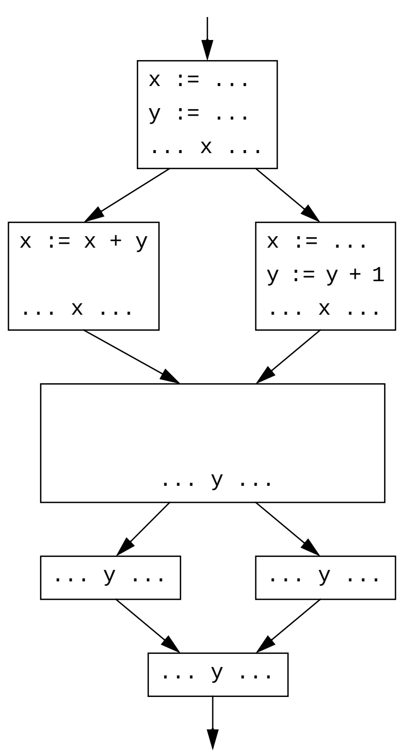
[Alpern, Rosen, Wegman, & Zadeck, two POPL 88 papers]

Invariant: at most one definition reaches each use

Constructing equivalent SSA form of program:

1. Create new target names for all definitions
2. Insert **pseudo-assignments** at merge points
   reached by multiple definitions of same source variable:
   $x_n := \phi(x_1, ..., x_n)$
3. Adjust uses to refer to appropriate new names

Example



Comparison

+ lower worst-case space cost than def/use chains: O(*EV*)

+ algorithms simplified by exploiting
  single assignment property:
  - variable has a unique meaning independent of program point
  - can treat variable & value synonymously

+ transformations not limited by reuse of variable names
  - can reorder assignments to same source variable, without
    affecting dependences of SSA version

– still not executable by itself

– still must update/reconstruct after transformations

– inverse property (static single use) not provided
  - **dependence flow graphs** [Pingali *et al.*] and
    value dependence graphs [Weise *et al.*] fix this,
    with single-entry, single-exit (SESE) region analysis

Very popular in research compilers, analysis descriptions

Common subexpression elimination

At each program point, compute set of **available expressions**:
map from expression to variable holding that expression

- e.g. `{a+b → x, -c → y, *p → z}`

CSE transformation using AE analysis results:
if `a+b`→`x` available before `y := a+b`, transform to `y := x`

Specification

All possible available expressions:
AvailableExprs = {*expr*→*var* | ∀*expr* ∈ Expr, ∀*var* ∈ Var}

- Var = set of all variables in procedure
- Expr = set of all right-hand-side expressions in procedure

[is this a function from Exprs to Vars, or just a relation?]

Domain AV = < Pow(AvailableExprs), $\le_{AV}$ >

$ae_1 \le_{AV} ae_2 \quad \Leftrightarrow$

- top:
- bottom:
- meet:
- lattice height:

Constraints

$AE_{x\ :=\ y\ op\ z}$:

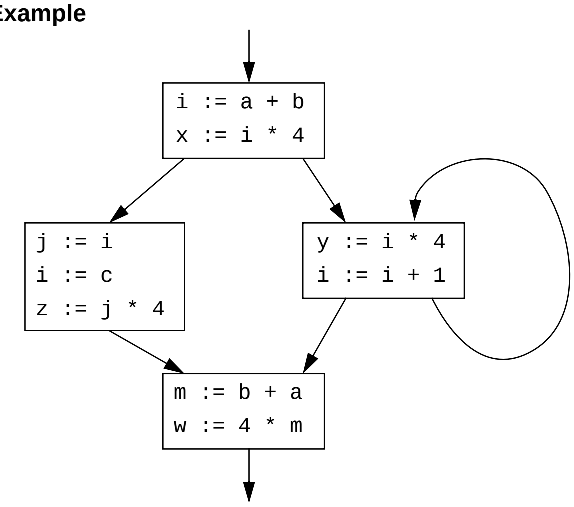
$AE_{x\ :=\ y}$:

Initial conditions at program points?

What direction to do analysis?

Can use bit vectors?

Example



Exploiting SSA form

Problem: previous available expressions overly sensitive to name choices, operand orderings, renamings, assignments, ...

A solution:

Step 1: convert to SSA form

- distinct values have distinct names
  ⇒ can simplify flow functions to ignore assignments

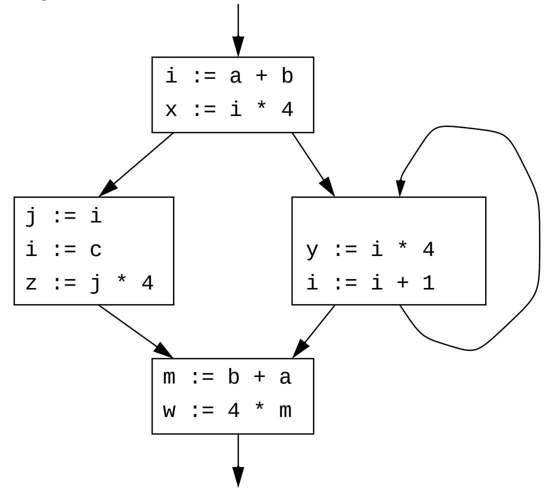
$AE^{SSA}_{\texttt{x := y } op \texttt{ z}}$:

Step 2: do **copy propagation**

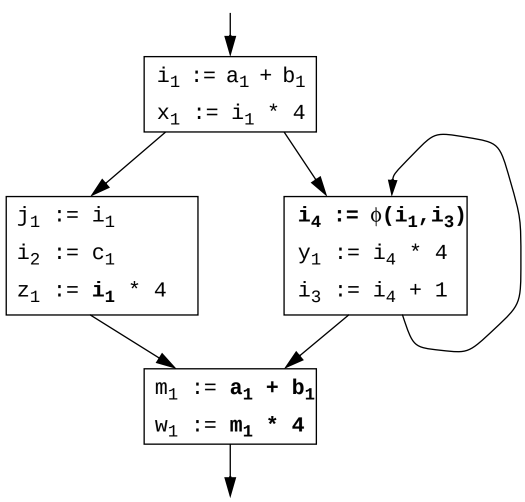
- same values (usually) have same names
  ⇒ avoid missed opportunities

Step 3: adopt canonical ordering for commutative operators
⇒ avoid missed opportunities

Example



After SSA conversion, copy propagation, & operand order canonicalization:



Loop-invariant code motion

Two steps: analysis & transformation

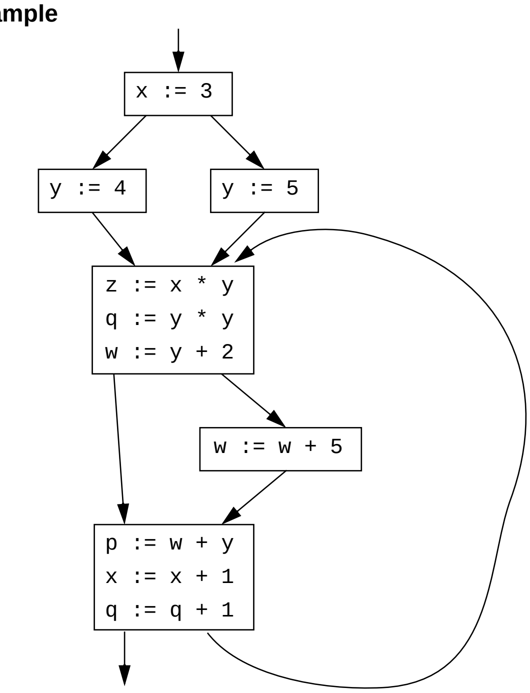
Step 1: find invariant computations in loop

- invariant: computes same result each time evaluated

Step 2: move them outside loop

- to top: **code hoisting**
  - if used within loop
- to bottom: **code sinking**
  - if only used after loop

Example



Detecting loop-invariant expressions

An expression is invariant w.r.t. a loop *L* iff:

base cases:

- it's a constant
- it's a variable use, **all of whose defs are outside *L***

inductive cases:

- it's an idempotent computation
  all of whose args are loop-invariant
- it's a variable use **with only one reaching def**,
  and the rhs of that def is loop-invariant

Computing loop-invariant expressions

Option 1:

- repeat iterative dfa
  until no more invariant expressions found
  - to start, optimistically assume all expressions loop-invariant

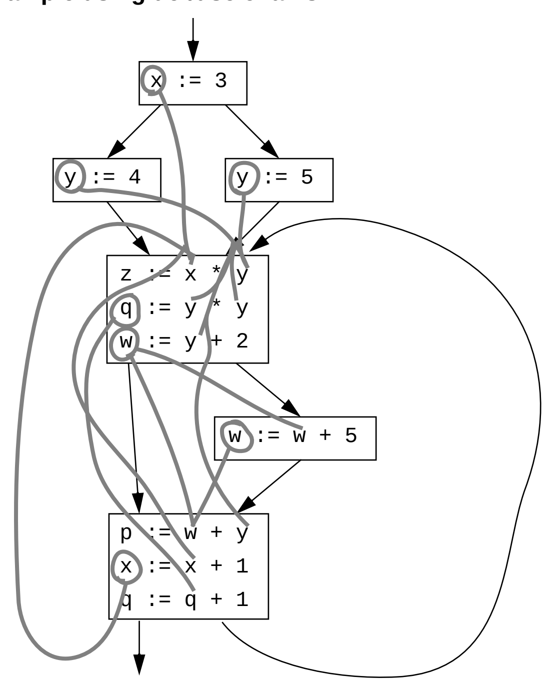
Option 2:

- build def/use chains,
  follow chains to identify & propagate
  invariant expressions

Option 3:

- convert to SSA form,
  then similar to def/use form

Example using def/use chains



Loop-invariant expression detection for SSA form

SSA form simplifies detection of loop invariants,
since each use has only one reaching definition

An expression is invariant w.r.t. a loop *L* iff:

base cases:

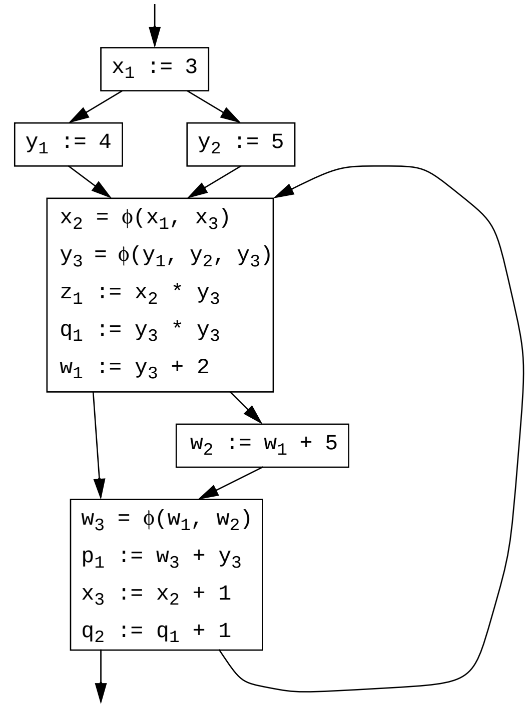
- it's a constant
- it's a variable use **whose <u>single def</u> is outside *L***

inductive cases:

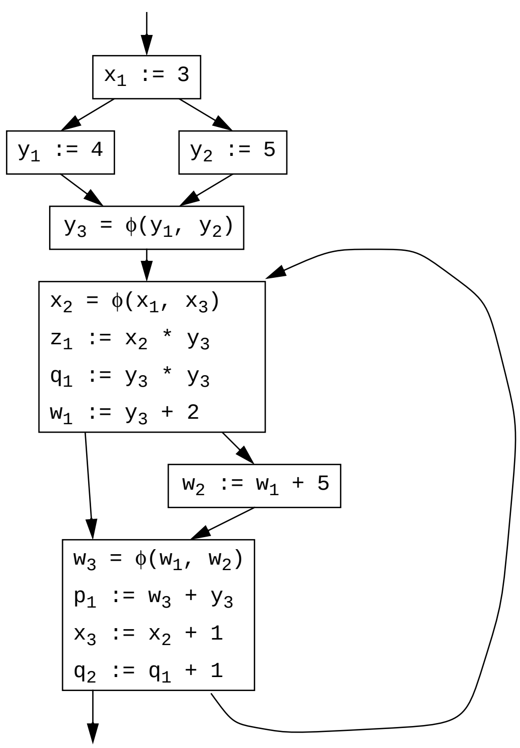
- it's an idempotent computation
  all of whose args are loop-invariant
- it's a variable use
  whose <u>single def</u>'s rhs is loop-invariant

ϕ functions are *not* idempotent

Example using SSA form



Example using SSA form & preheader



Code motion

When find invariant computation *S:* `z := x op y`,
want to move it out of loop (to loop preheader)

When is this legal?

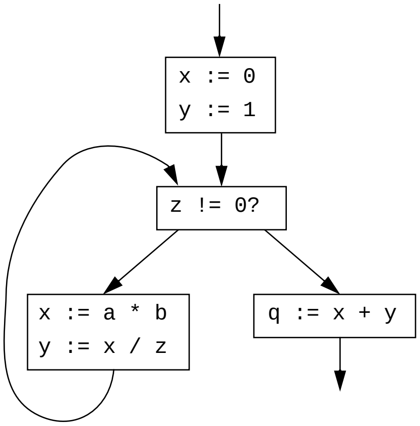
Sufficient conditions:

- *S* **dominates** all loop exits
  [*A* dominates *B* when
  all paths to *B* must first pass through *A*]
  - otherwise may execute *S* when never executed otherwise
  - can relax this condition, if *S* has no side-effects or traps,
    at cost of possibly slowing down program
- *S* is only assignment to `z` in loop, &
  no use of `z` in loop is reached by any def other than *S*
  - otherwise may reorder defs/uses and change outcome
  - unnecessary in SSA form!

If met, then can move *S* to loop preheader

- but preserve relative order of invariant computations,
  to preserve data flow among moved statements

Example of need for domination requirement



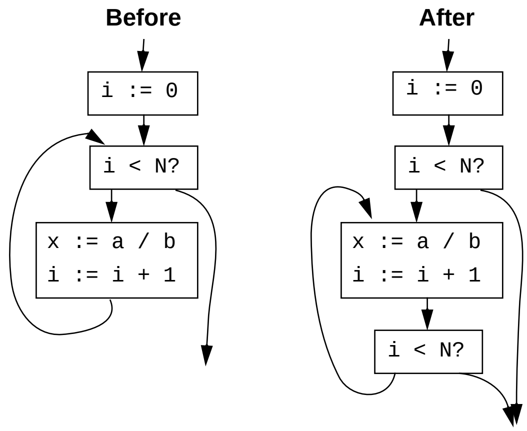
Avoiding domination restriction

Requirement that invariant computation dominates exit is strict

- nothing in conditional branch can be moved
- nothing after loop exit test can be moved

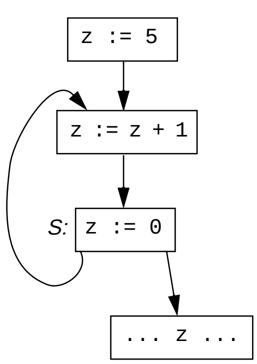
Can be circumvented through other transformations such as **loop normalization**

- move loop exit test to bottom of loop



Example of data dependence restrictions

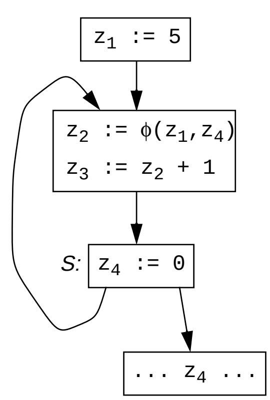
"*S* is only assignment to z in loop, & no use of z in loop is reached by any def other than *S*"



Example in SSA form

Restrictions unnecessary if in SSA form

- if reorder defs/uses, generate code along merging arcs to implement ϕ functions



Loop-invariant code copying

Alternative to code motion:
  copy instruction to loop header, assigning to new temp, then do CSE & copy propagation to simplify in-loop version

- more modular design, leverage off of existing optimizations

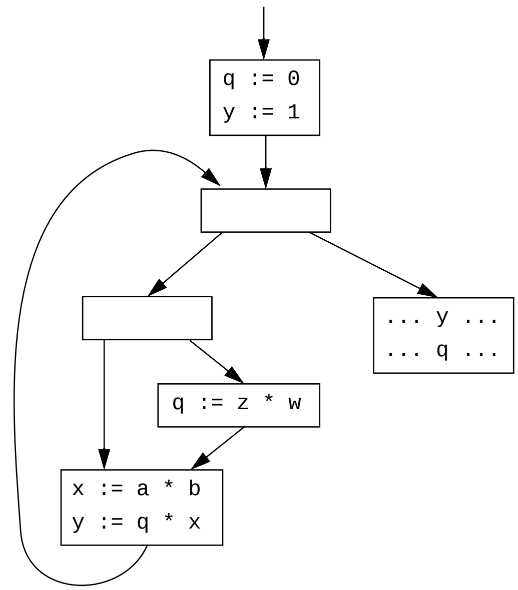
Can always copy, unless instruction has side-effects

CSE & copy propagation will eliminate in-loop instruction exactly when (non-SSA) loop-invariant code motion would have, PLUS can replace invariant but unmovable instructions with copies

SSA-based code motion gets same effect

- copies correspond to reified ϕ functions

Example



Control dependence

Must ensure side-effects occur in proper order

Must ensure side-effects occur only under right conditions

CFG represents control dependence explicitly

– but overspecifies control dependence requirements

Control dependence graph

Program dependence graph (PDG):
  data dependence graph + control dependence graph (CDG)
  [Ferrante, Ottenstein, & Warren, TOPLAS 87]

Idea: represent controlling conditions directly

- complements data dependence representation

A node (basic block) N_1 is **control-dependent** on another N_2 iff N_2 determines whether N_1 executes, i.e.

- there exists a path from N_1 to N_2 s.t. every node in the path other than N_1 is **post-dominated** by N_2
- N_2 does not post-dominate N_1

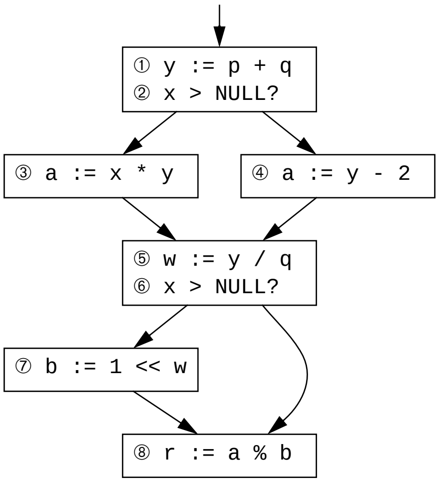
Control dependence graph:
  N_1 proper descendant of N_2 iff N_1 control-dependent on N_2

- label each child edge with required branch condition
- group all children with same condition under **region** node

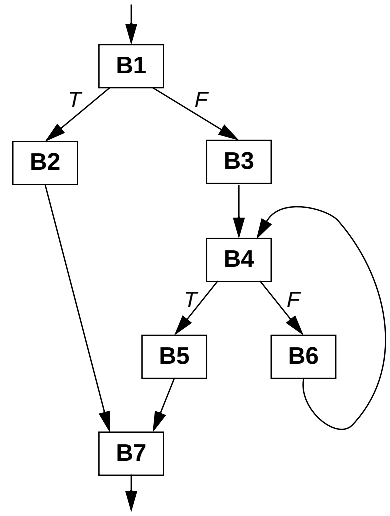
Two sibling nodes execute under same control conditions ⇒ can be reordered or parallelized, as data dependences allow

Challenging to “sequentialize” back into CFG form

Example



An example with a loop



Value dependence graphs

[Weise, Crew, Ernst, & Steensgaard, POPL 94]

Idea: represent all dependences,
including control dependences, as data dependences

+ simple, direct dataflow-based representation of all “interesting” relationships
  - analyses become easier to describe & reason about
– harder to sequentialize into CFG

Control dependences as data dependences:

- control dependence on order of side-effects
  ⇒ data dependence on reading & writing to global Store
  - optimizations to break up accesses to single Store into separate independent chunks
    (e.g. a single variable, a single data structure)
- control dependence on outcome of branch
  ⇒ a select node, taking test, then, and else inputs

Loops implemented as tail-recursive calls to local procedures

Apply CSE, folding, etc. as nodes are built/updated

Like DAG representation of BB, but for whole procedure

VDG for example, after store splitting

```
y := p + q
if x > NULL then a := x * y else a := y - 2
w := y / q
if x > NULL then b := 1 << w
r := a % b
```