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

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

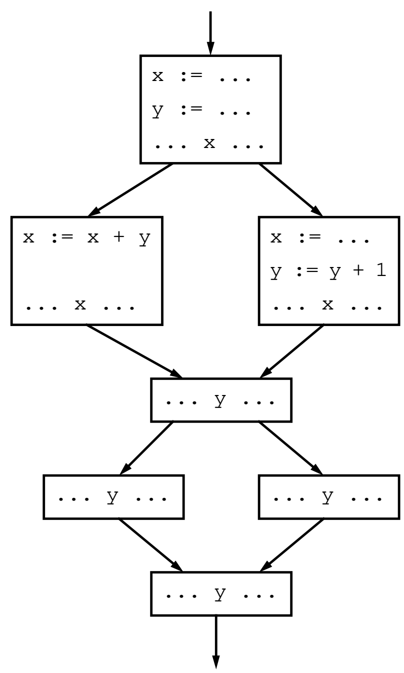
– ignores control flow
  - misses some optimization opportunities, since it assumes all paths taken
  - not executable by itself, since it doesn't include control dependence links

– must update after transformations

– 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

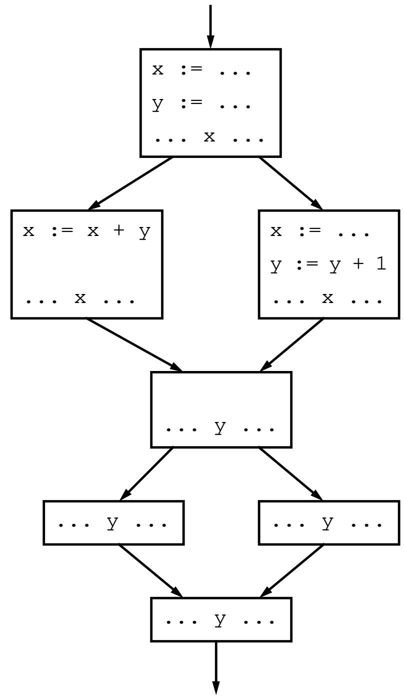
[Alpern, Rosen, Wegman, & Zadeck, POPL 88, two times]

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:
   x_n `:=` $\phi(x_1, \ldots, x_n)$
3. Adjust uses to refer to appropriate new names

Example



Analysis

+ lower worst-case space cost: O(*EV*)
+ algorithms simplified by exploiting single assignment property:
  - variable has a unique meaning independent of program point
  - can treat variable & value synonymously

– still not executable by itself

– still must update/reconstruct after transformations

– inverse property 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}`

Transformation using 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
- function? full relation?

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

$ae_1 \leq_{AV} ae_2 \Leftrightarrow$?

- top?
- bottom?
- meet?
- height?

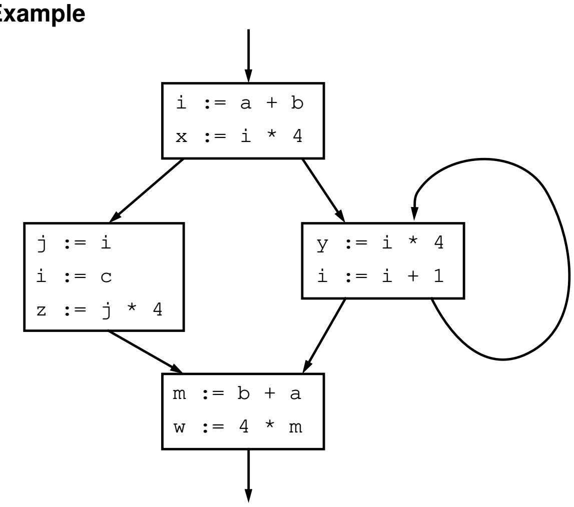
Constraints

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

Initial conditions at program points?

What direction to do analysis?

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

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

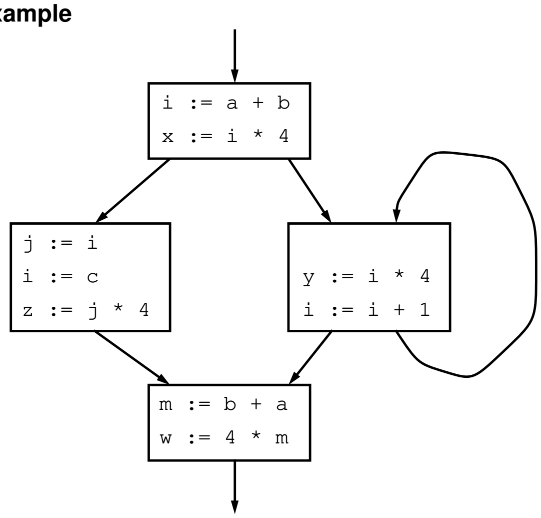
Step 2: do **copy propagation**

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

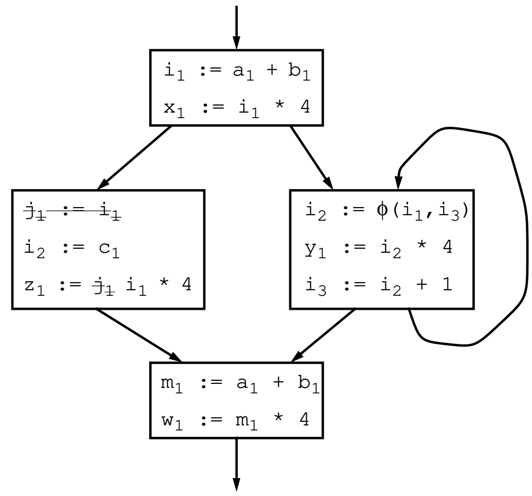
Step 3: adopt canonical ordering for commutative operators

- avoid missed opportunities

Example



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



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 tree

[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 tree:
N_1 child 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

Applications:
- two sibling nodes can be reordered or run in parallel (as data dependences allow)

Example