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

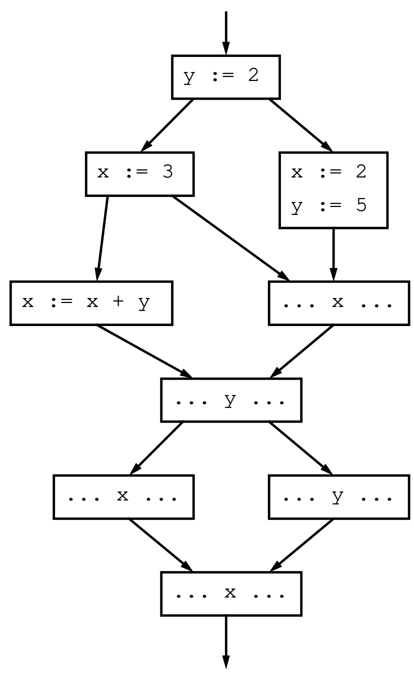
+ directly captures data flow for analysis
  - e.g. constant propagation easy

– 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

– 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

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

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

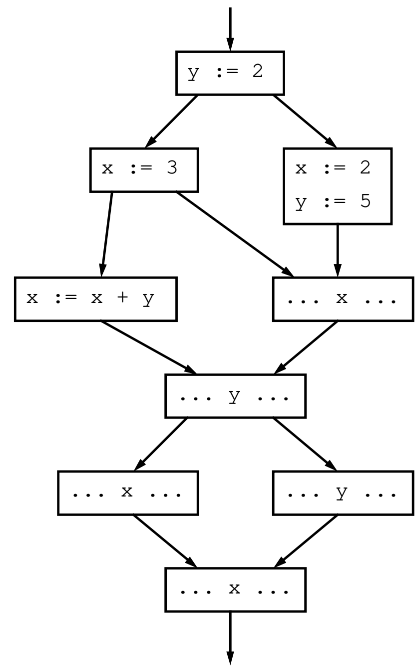
– still not executable by itself

– 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

– must maintain invariant after program transformations

Very popular in research compilers, analysis descriptions

Example



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}`

If `a+b→x` available for `y := a+b`, transform to `y := x`

Set of Domain =
Powerset({*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?

Lattice ≤ ordering operator?

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

Common subexpression elimination, continued

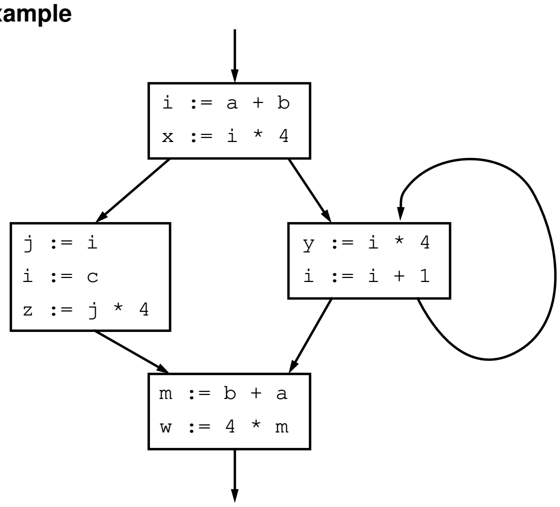
What direction to do analysis?

Initial conditions?

Flow functions:

$AE_{x\ :=\ y\ op\ z}(in) =$

Example



Value graph representation

Revised available expressions:
convert to SSA form; associate a **value** with each name

$$
\begin{array}{lll}
\texttt{value} & ::= & \textit{constant} \\
 & | & \texttt{unknown}_{var} \\
 & | & op(\texttt{value}_{\texttt{op1}}, \ldots, \texttt{value}_{\texttt{opN}}) \\
 & | & \phi(\texttt{value}_{\texttt{pred1}}, \ldots, \texttt{value}_{\texttt{predN}})
\end{array}
$$

$\texttt{unknown}_{\texttt{v1}}$ might be same or different than $\texttt{unknown}_{\texttt{v2}}$, $\texttt{v1} \neq \texttt{v2}$

CSE with SSA form and values

If using SSA form and value graphs,
compute mappings between variables and values

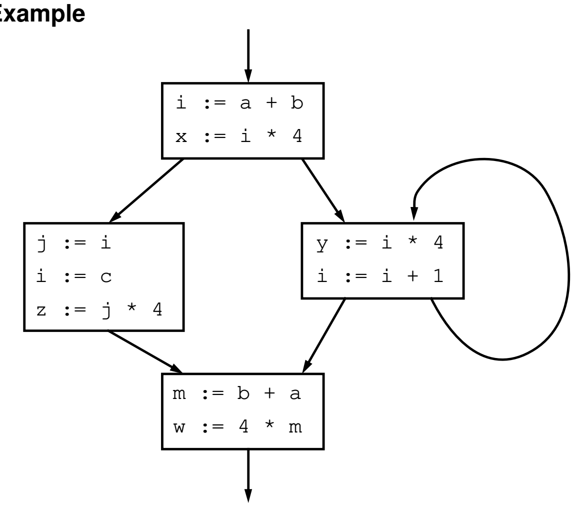
- e.g. $\{x \leftrightarrow v_a + v_b,\ y \leftrightarrow (v_a + v_b) * 2,\ z \leftrightarrow v_z\}$

If at `x := y op z`,
in $\supseteq \{y \leftrightarrow v_y,\ z \leftrightarrow v_z,\ q \leftrightarrow v_y \text{ op } v_z\}$,
replace with `x := q`

Flow function:

`x := y op z`

Example



“Detecting Equality of Variables in Programs”

[Alpern, Wegman, & Zadeck 88]

Introduces SSA form

Describes value graphs

Presents an iterative, **optimistic** algorithm for identifying when two value graphs are the same

Presents a fancier form of value graphs that includes some control dependence info:

```
value  ::=  ...
        |    φ_if(value_then, value_else, value_test)
        |    φ_enter(value_head, value_tail)
        |    φ_exit(value_exit-test, value)
```

- handles simple structured `if` & `while` programs, but not arbitrary CFG’s

Value dependence graphs

[Weise, Crew, Ernst, & Steensgaard 94]

Idea: represent all dependences, including control dependences, as data dependences, in a dataflow graph

+ representation becomes complete (executable)
+ direct representation of all “interesting” relationships
  - analyses become easier to describe & reason about

– harder to linearize into target code

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