Soundness of Data Flow Analysis

We'd like to convince ourselves, even prove formally, that our dataflow analysis is correct, i.e. sound, with respect to some intended uses

We need two things:

- a reference *concrete* semantics that defines the "truth", against which we compare our *abstract* semantics
  - including a *concrete domain* of information at program points against which we compare our *abstract domain* of analysis results at program points
- an *abstraction relation* that specifies when an abstract domain element conservatively approximates a concrete domain element, for our intended uses

Concrete semantics

Many ways to define the semantics of a programming language

A good way for our purposes is *small-step operational semantics*, i.e., a set of transition rules like the following:

$<pp_{in}, mem_{in}> \rightarrow_{x := y+z} <pp_{out}, mem_{out}>$

where pp_{in} = pred-pt($x := y+z$)

pp_{out} = succ-pt($x := y+z$)

$mem_{out} = mem_{in}[x \rightarrow mem_{in}(y)+mem_{in}(z)]$

"if execution reaches program point pp_{in} with memory state mem_{in}, and the instruction after that program point is $x := y+z$, then program execution will "step" to program point pp_{out} with memory state mem_{out}."

These small-step rules are just (concrete) flow functions!

- but the info being "propagated" is the whole state of the computation (and the outside world, perhaps)
- but control flow is more explicit, to account for which way execution proceeds after branches

Traces

Concrete execution of a whole program is a *trace*

- sequence of <*pp*,*mem*> pairs, starting from the initial program entry point and memory state, following the concrete flow functions, until reaching final <*pp*,*mem*> which has no transition
- could be infinitely long

If convenient, we can collapse traces onto the control flow graph, storing not a sequence of pairs but rather a map from each program point to the set of all memories that occur in the trace at that program point, called the *collecting (concrete) semantics*

$\{ (pp \rightarrow mems) \mid$
$mems = \{ mem' \mid <pp,mem'> \in \text{Trace} \} \}$

Abstraction relation

Now we have concrete information (memories) and abstract information (domain elements computed by our analysis). When does the abstract information safely, possibly conservatively, characterize the concrete information?

Depends on the use/intention of the abstract info

Define this using an *abstraction relation* α:

For concrete info c and abstract info a, $(c,a) \in \alpha$ iff a is a safe approximation of c

E.g., for constant propagation:

$(mem, d_{CP}) \in \alpha_{CP} \Leftrightarrow$ $[d_{CP} \subseteq \text{Var}\times\text{Const}]$

$\forall(var \rightarrow const) \in d_{CP}.\ mem(var) = const$

(Could define α as a relation between whole traces and abstract info, to allow the abstract info to approximate history- or future-sensitive info, e.g. for reaching defs or live variables)

Local and global soundess

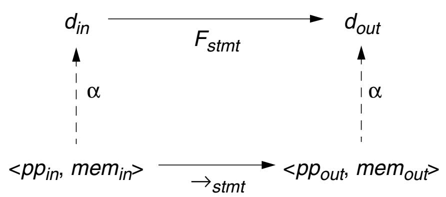
Lemma (Local soundness of analysis $A = \langle\alpha, F\rangle$).

if $\langle pp_{in}, mem_{in}\rangle \rightarrow_{stmt} \langle pp_{out}, mem_{out}\rangle$

and $(mem_{in}, d_{in}) \in \alpha$

and $F_{stmt}(d_{in}) = d_{out}$

then $(mem_{out}, d_{out}) \in \alpha$



- prove this by examining each *F* flow function case

Theorem (Global soundess of analysis $A = \langle\alpha, F\rangle$).

If we start the abstract analysis with safe abstract info at the first program point, the abstract analysis will compute safe abstract info at each program point in the trace.

- by induction over the trace, using local soundness lemma
- proof is independent of the actual analysis!

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

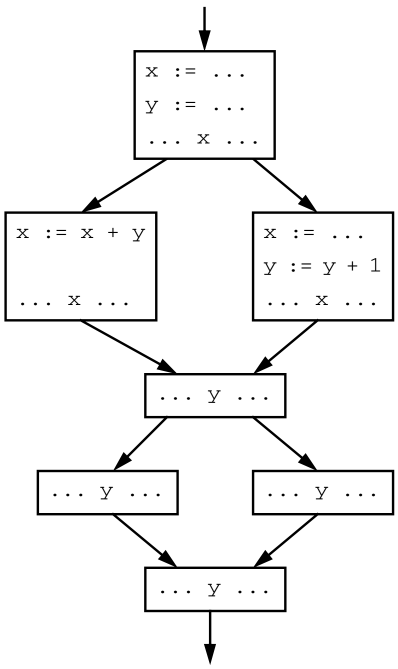
– can have multiple defs of same variable in program, multiple defs can reach a use
  - complicates analysis, representation

– 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
  - not too hard (just remove edges)

– space-consuming, in worst case: $O(N^2)$ edges per variable

Example



Static Single Assignment (SSA) form

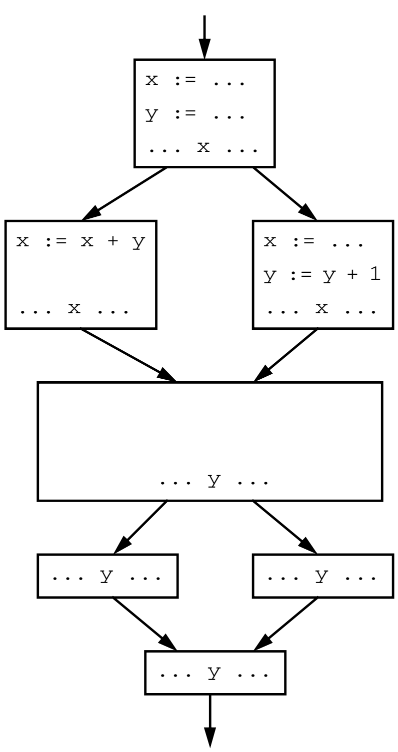
[Alpern, Rosen, Wegman, & Zadeck, two POPL 88 papers]

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

Example



Implementing ϕ-functions

Semantics of $x_m := \phi(x_1, \ldots, x_n)$:
set x_m to x_i, if control last came from predecessor i

How to implement (generate code for) this?

- along each predecessor edge i, insert $x_m := x_i$
- delete ϕ statement

If the register allocator assigns all of x_m, x_1, ... ,x_n to the same register, then these move instructions will be deleted

- x_m, x_1, ... ,x_n usually have non-overlapping lifetimes, so this kind of assignment is legal

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, its defining statement, & its value synonymously
  - can have single global table mapping var to info, not one per program pt. that must be propagated, copied, etc.

+ transformations not limited by reuse of variable names
  - can reorder assignments to same source variable, without changing meaning in 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}`

More generally, can have map from
expensive expression to equivalent but cheaper expression

- subsumes CSE, constant prop, copy prop., ...

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 = $\{expr \rightarrow var \mid \forall expr \in$ Exprs, $\forall var \in$ Vars$\}$
= Exprs × Vars

- Exprs = set of all right-hand-side expressions in procedure
- Vars = set of all variables in procedure

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

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

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

- T:
- ⊥:
- meet:
- lattice height:

Flow functions

What direction to do analysis?

Initial conditions?

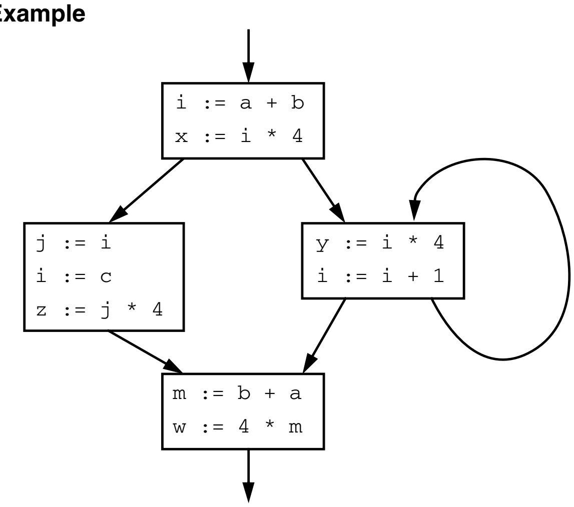
$AE_{x\ :=\ y\ op\ z}$:

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

Can use bit vectors?

Can summarize sequences of flow functions?

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

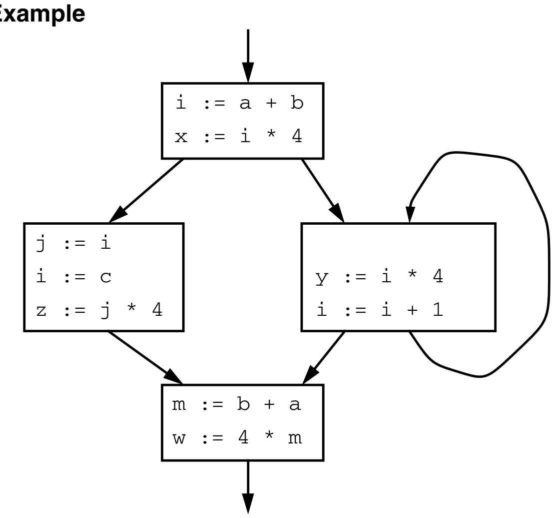
$AE^{SSA}_{x\ :=\ y\ op\ z}$:

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: