Lattice-Theoretic Data Flow Analysis Framework

Goals:

- provide a single, formal model that describes all DFAs
- formalize notions of “safe”, “conservative”, “optimistic”
- place precise bounds on time complexity of DF analysis
- enable connecting analysis to underlying semantics for correctness proofs

Plan:

- define **domain** of program properties computed by DFA, organize domain elements as a **lattice**
- define flow functions, merge function over this domain using lattice operations
- benefit from lattice theory in attacking above issues

History: Kildall [POPL 73], Kam & Ullman [JACM 76]

Lattices

Define lattice $D = (S, \leq)$:

- S is a set of elements of the lattice
- $\leq$ is a binary relation over elements of S

Required properties of $\leq$:

- $\leq$ induces a **partial order** over S
  - reflexive, transitive, & anti-symmetric
- every pair of elements of S has
  a unique **greatest lower bound** (a.k.a. meet) and
  a unique **least upper bound** (a.k.a. join)

Height of D =
longest path through partial order from greatest to least

- infinite lattice can have finite height (but infinite width)

Top (T) = unique element of S that's greatest, if exists

Bottom ($\perp$) = unique element of S that's least, if exists

Lattice models in data flow analysis

Model information by elements of a lattice

- top = best case info
- bottom = worst case info
- merge function = g.l.b. (meet) on lattice elements
- if $a \le b$, then a is a conservative approximation to b
- initial info at program points (at least back edges): top

(Opposite conventions used in semantics!)

Reaching definitions:

- elements:
- $\le$:
  - meet:
  - top:
  - bottom:

Reaching constants:

- elements:
- $\le$:
  - meet:
  - top:
  - bottom:

Tuples of lattices

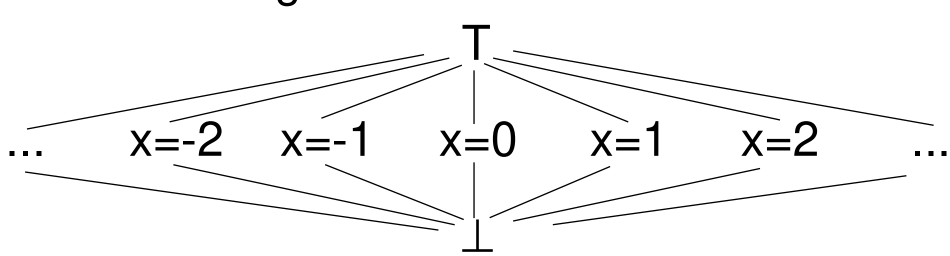
Often helpful to break down a complex lattice into a tuple of lattices, one per variable being analyzed

Formally: $D_T = \langle S_T, \leq_T \rangle \ = \ (D = \langle S, \leq \rangle)^N$

- $S_T = S_1 \times S_2 \times \ldots \times S_N$
  - element of tuple domain is a tuple of elements from each variable's domain
  - i^{th} component of tuple is info about i^{th} variable
- $\langle \ldots, d_{1i}, \ldots \rangle \ \leq_T \ \langle \ldots, d_{2i}, \ldots \rangle \ \equiv \ d_{1i} \leq d_{2i}, \forall i$
  - i.e. pointwise ordering
- meet: pointwise meet
- top: tuple of tops
- bottom: tuple of bottoms
- height(D_T) = N * height(D)

E.g. reaching constants

- lattice for single variable is 3-level lattice:



- whole problem is tuple of individual lattices

Some typical lattice domains

Single-point lattice: just bottom

- trivial do-nothing analysis

Two-point lattice: top and bottom

- computes a boolean property
- a tuple of two-point lattices ⇔ a bit-vector

A lifted set: a set of incomparable values, plus top & bottom

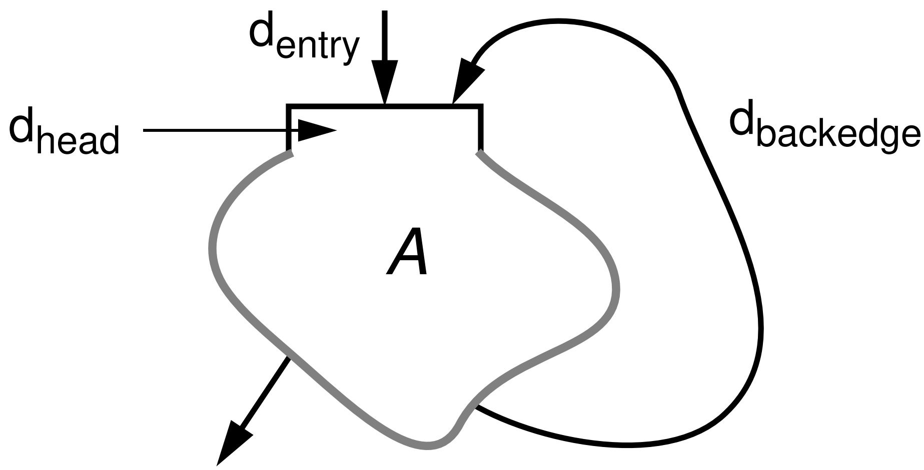
- e.g. reaching constants domain

Powerset lattice: set of all subsets of a set *S*, ordered somehow

- top & bottom = ∅ & *S*, or vice versa
- "a collecting analysis"
- isomorphic to tuple of booleans
  indicating membership in subset of elements of *S*

Analysis of loops in lattice model

Consider:



Assume A(d) computes info at backedge given d, info at head

Want solution to constraints:

$d_{head} = d_{entry} \cap d_{backedge}$

$d_{backedge} = A(d_{head})$

Let $F(d) = d_{entry} \cap A(d)$

Then want fixpoint of F:

$d_{head} = F(d_{head})$

Iterative analysis in lattice model

Iterative analysis computes fixpoint by iterative approximation:

$F^0 = d_{entry} \cap T$
$F^1 = d_{entry} \cap A(F^0) = F(F^0) = F(d_{entry})$
$F^2 = d_{entry} \cap A(F^1) = F(F^1) = F(F(F^0)) = F(F(d_{entry}))$
. . .
$F^k = d_{entry} \cap A(F^{k-1}) = F(F^{k-1}) = F(F(...(F(d_{entry}))...))$

until

$F^{k+1} = d_{entry} \cap A(F^k) = F(F^k) \quad = \quad F^k$

Is k finite?

If so, how big can it be?

Termination of iterative analysis

In general, k need not be finite

Sufficient conditions for finiteness:

- flow functions (e.g. F) are **monotonic**
- lattice is of finite height

A function F is monotonic iff:

$$d_1 \le d_2 \;\Rightarrow\; F(d_1) \le F(d_2)$$

- for application of DFA, this means that giving a flow function more conservative inputs leads to more conservative outputs, never more optimistic outputs

For monotonic F over domain D, the maximum number of times that F can be applied to itself, starting w/ any element of D, w/o reaching fixpoint, is height(D)-1

- start at top of D
- go down one level in lattice each application of F
- eventually must hit fixpoint or bottom (which is guaranteed to be a fixpoint), if D of finite height

Complexity of iterative analysis

How long does iterative analysis take?

`l`: depth of loop nesting

`n`: # of stmts in loop

`t`: time to execute one flow function

`k`: height of lattice

Another example: integer range analysis

For each program point,
for each integer-typed variable,
calculate (an approximation to) the set of integer values that can be taken on by the variable

- use info for constant folding comparisons,
  for eliminating array bounds checks,
  for (in)dependence testing of array accesses,
  for eliminating overflow checks

What domain to use?

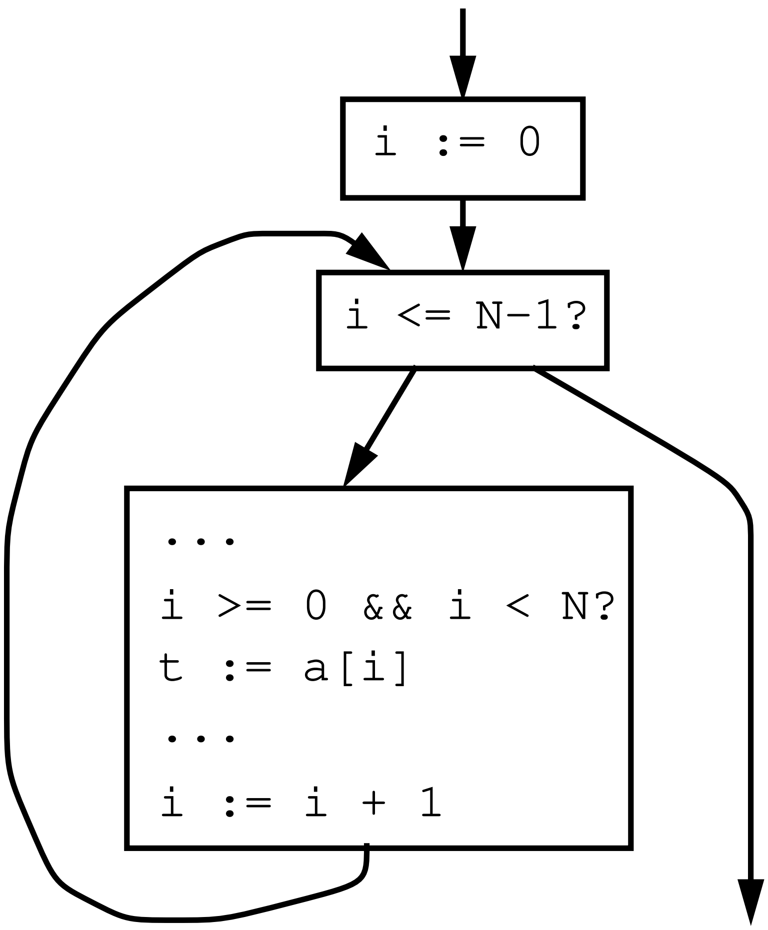
- what is its height?

What flow functions to use?

- are they monotonic?

Example

```
for i := 0 to N-1
  ... a[i] ...
end
```



Widening operators

If domain is tall, then can introduce artificial generalizations (called **widenings**) when merging at loop heads

- can guarantee a small number of widenings possible

Abstract interpretation

[Another name for dataflow analysis, with different connotations]

Describe analysis in terms close to semantics of operations, enabling careful reasoning about correctness of analysis

Step 1: define concrete semantics of operators in IR, in terms of concrete domain

For each analysis:

Step 2: define an abstract semantics of operators in IR, in terms of abstract domain

Step 3: define abstraction function relating elements of concrete domain to elements of abstract domain

Step 4: prove abstract semantics is safe approximation to concrete semantics

Concrete semantics

Define real semantics of evaluating the IR; an interpreter

Concrete domain: map from vars to (values or variables)

Concrete flow functions:

$\text{Sem}_{\texttt{x := } N}$:
  $\text{Info}_{succ} = \text{Info}_{pred}[\texttt{x} \to N]$

$\text{Sem}_{\texttt{x := \&y}}$:
  $\text{Info}_{succ} = \text{Info}_{pred}[\texttt{x} \to \texttt{y}]$

$\text{Sem}_{\texttt{x := y + z}}$:
  $\texttt{let}$ rhs $= \text{Info}_{pred}(\texttt{y}) + \text{Info}_{pred}(\texttt{z})$ $\texttt{in}$
    $\text{Info}_{succ} = \text{Info}_{pred}[\texttt{x} \to \text{rhs}]$

$\text{Sem}_{\texttt{*p := *q + *r}}$:
  $\texttt{let}$ rhs $= \text{Info}_{pred}(\text{Info}_{pred}(\texttt{q})) + \text{Info}_{pred}(\text{Info}_{pred}(\texttt{r}))$ $\texttt{in}$
    $\text{Info}_{succ} = \text{Info}_{pred}[\text{Info}_{pred}(\texttt{p}) \to \text{rhs}]$

Concrete control flow

$\text{Sem}_{\texttt{if x < y goto L1 else L2}}$:

$\texttt{if}$ $\text{Info}_{\text{pred}}(\texttt{x}) < \text{Info}_{\text{pred}}(\texttt{y})$
$\texttt{then}$ $\text{Info}_{\texttt{L1}} = \text{Info}_{\text{pred}}$
$\texttt{else}$ $\text{Info}_{\texttt{L2}} = \text{Info}_{\text{pred}}$

[Loops awkward in this format.
Recursion or machine configurations better.]

An abstraction: collecting semantics

Collecting semantics: abstract semantics whose domain is the set of possible values from the concrete domain

- collect together all values seen during any execution

Collecting domain: map from vars to sets of (values or variables)

- tuple domain of powerset domains, ordered by $\supseteq$

Collecting flow functions:

$\text{Sem}_{\texttt{x := } N}$:
  $\text{Info}_{succ} = \text{Info}_{pred}[\texttt{x} \rightarrow \{N\}]$

$\text{Sem}_{\texttt{x := \&y}}$:
  $\text{Info}_{succ} = \text{Info}_{pred}[\texttt{x} \rightarrow \{\texttt{y}\}]$

$\text{Sem}_{\texttt{x := y + z}}$:
  $\texttt{let}$ rhs = $\{ y+z \mid y \in \text{Info}_{pred}(\texttt{y}) \wedge z \in \text{Info}_{pred}(\texttt{z}) \}$ $\texttt{in}$
    $\text{Info}_{succ} = \text{Info}_{pred}[\texttt{x} \rightarrow \text{rhs}]$

$\text{Sem}_{\texttt{*p := *q + *r}}$:
  $\texttt{let}$ rhs = $\{ y+z \mid q \in \text{Info}_{pred}(\texttt{q}) \wedge r \in \text{Info}_{pred}(\texttt{r}) \wedge y \in \text{Info}_{pred}(q) \wedge z \in \text{Info}_{pred}(r) \}$,
    lhs = $\{ p \mid p \in \text{Info}_{pred}(\texttt{p}) \}$ $\texttt{in}$
    $\text{Info}_{succ} = \text{Info}_{pred}[\ldots, p_i \rightarrow \text{Info}_{pred}(p_i) \cup \text{rhs}, \ldots], \forall p_i \in \text{lhs}$

Collecting control flow

$\text{Sem}_{\texttt{if x < y goto L1 else L2}}$:

$\texttt{let}$ outcomes = { x < y | x ∈ $\text{Info}_{\text{pred}}(\texttt{x})$ ∧ y ∈ $\text{Info}_{\text{pred}}(\texttt{y})$ } $\texttt{in}$

$\texttt{if}$ true ∈ outcomes $\texttt{then}$ $\text{Info}_{\texttt{L1}} = \text{Info}_{\text{pred}}$

$\texttt{if}$ false ∈ outcomes $\texttt{then}$ $\text{Info}_{\texttt{L1}} = \text{Info}_{\text{pred}}$

Merge:

$\text{Info}_{\text{succ}} = \{\ldots, \texttt{x} \to \bigcup_{p \in \text{pred}} \text{Info}_p(\texttt{x}), \ldots\}$

Initial conditions: $\text{Info}_{\text{entry}} = \{\ldots, \texttt{x} \to \{\}, \ldots\}$

Becoming computable

Concrete semantics is uncomputable:
must run program until it halts to get final answer

Collecting semantics is also uncomputable:
must run program until it halts to collect all intermediate answers

Need to abstract away some of the precision of collecting interpretation to get a computable abstract interpretation

- go from infinitely tall powerset domain to finite-height domain

E.g. constant propagation domain, range analysis domain, domain of signs, domain of possible pointer targets, ...

Abstraction function and correctness

To connect concrete and abstract domain(s), need abstraction function

Abstraction function A takes an element of input domain and returns the most specific corresponding element of abstract domain

- for collecting semantics: $A(d_{\text{concrete}}) = \{d_{\text{concrete}}\}$
- for more abstract semantics:
  squash out precision in collecting semantics somehow

Safety rule: if concrete semantics produces element d_{concrete}, abstract semantics must produce some element d_{abstract} such that $A(d_{\text{concrete}}) \le d_{\text{abstract}}$

- true for collecting semantics
- relating collecting flow functions to abstract flow functions shouldn't be hard

Sharlit

A data flow analyzer generator [Tjiang & Hennessy 92]

- analogous to YACC

User writes basic primitives:

- control flow graph representation
  - nodes are instructions, not basic blocks
- domain (“flow value”) representation and key operations
  - `init`
  - `copy`
  - `is_equal`
  - `meet`
- flow functions for each kind of instruction
- action routines to optimize after analysis

Sharlit constructs iterative dataflow analyzer from these pieces

\+ easy to build, extend

– not highly efficient

Path compression

Can improve analysis efficiency by
summarizing effect of sequences of nodes

User can define path compression operations to collapse nodes together

- linear joining of sequential nodes
  $\Rightarrow$ summarizes effect of whole BB
  - presumes a fixed GEN/KILL bit-vector structure to be effective
- merge trees into extended BB's
- merge merges, loops as in interval analysis
  - simplifies reducible parts, applies iteration to nonreducible parts

\+ gets efficiency, preserves modularity & generality

– doesn't support data-dependent flow functions,
cannot simulate optimizations during analysis

Performance results for code quality of generated optimizer,
but not for compilation speed of optimizer

Vortex IDFA framework

Like Sharlit, except a run-time library rather than a compiler

User defines a subclass of `AnalysisInfo` to represent elements of domain

- `copy`
- `merge` (lattice g.l.b. operator)
- `generalizing_merge` (g.l.b. with optional widening)
- `as_general_as` (lattice ≤ operator)

User invokes `traverse` to perform analysis:

```
cfg.traverse(direction, is_iterative?,
  initial_analysis_info,
  λ(rtl, info){ rtl.flow_fn(info) })
```

Flow function returns an `AnalysisResult`:

- keep instruction and continue analysis w/ updated info(s)
- delete instruction/constant-fold branch
- replace instruction with instruction or subgraph

`ComposedAnalysis` supports running multiple analyses interleaved at each instruction

Features of Vortex IDFA

Big idea: separate analyses and transformations, make framework compose them appropriately

- don't have to simulate the effect of transformations during analysis
- can run analyses in parallel if each provides opportunities for the other
  - sometimes can achieve strictly better results this way than if run separately in a loop
- more general transformations supported (e.g. inlining) than Sharlit

Exploit inheritance & closures

Analysis speed is not stressed

- no path compression
- no "compilation" of analysis with framework

[Interprocedural analysis support discussed later]

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

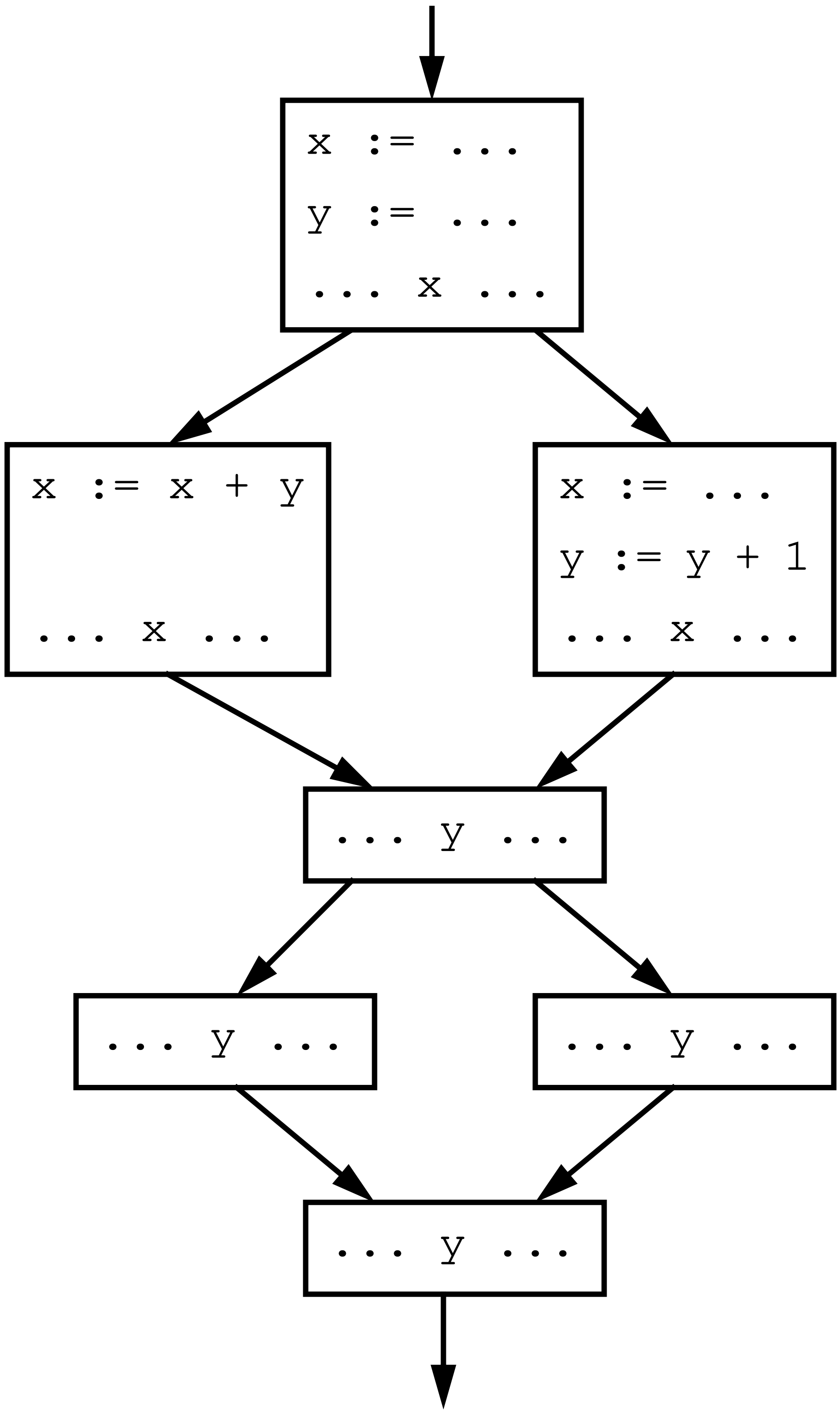
– ignores control flow
  - misses some optimization opportunities, since it assumes all paths taken
  - not executable by itself, since it doesn't include control dependence links

– 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]

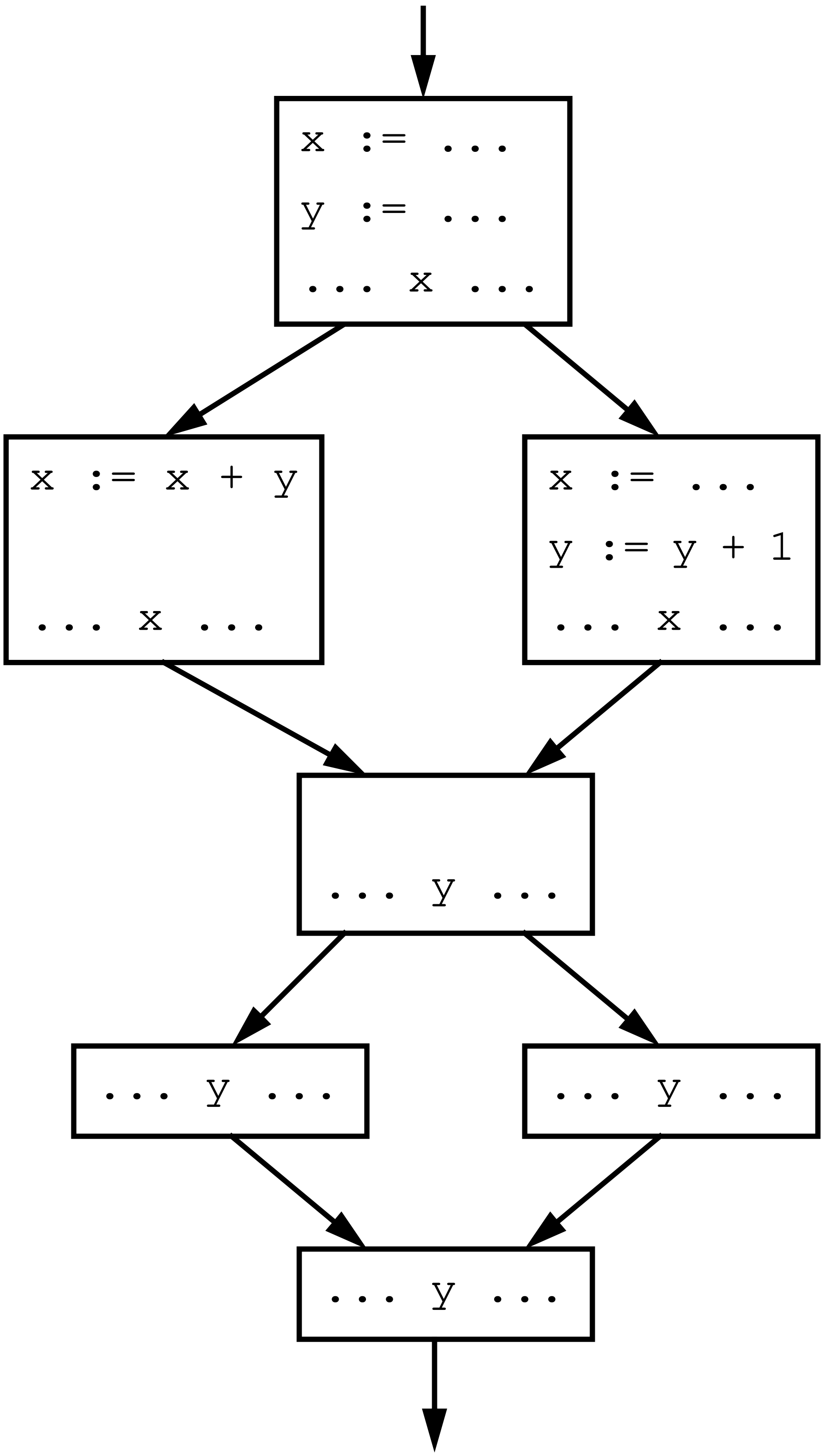
Invariant: at most one definition reaches each use

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



Comparison

+ 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:

$\text{AvailableExprs} = \{expr \to var \mid \forall expr \in \text{Expr}, \forall var \in \text{Var}\}$

- Var = set of all variables in procedure
- Expr = set of all right-hand-side expressions in procedure
- function? full relation?

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

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

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

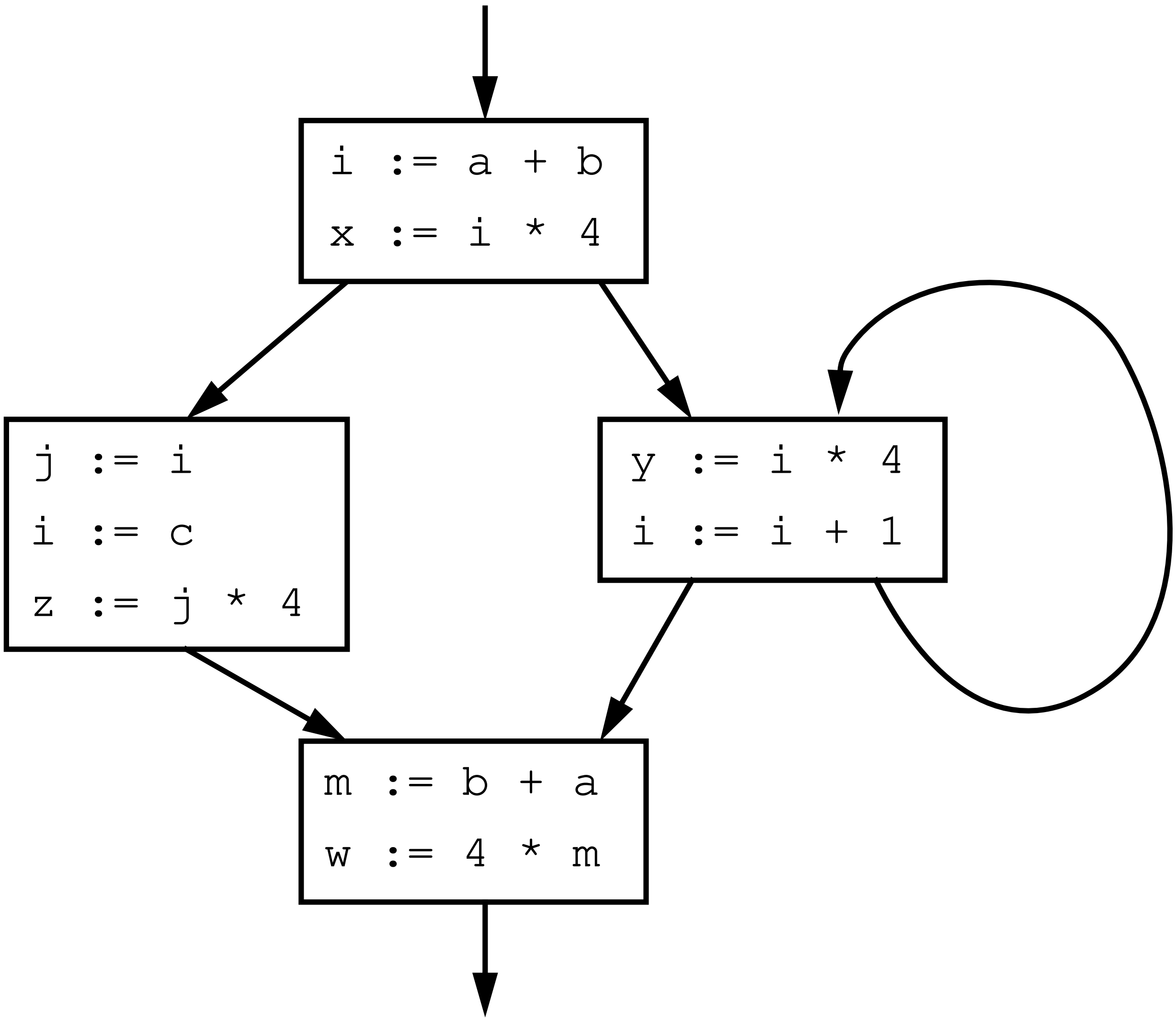
Constraints

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

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

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

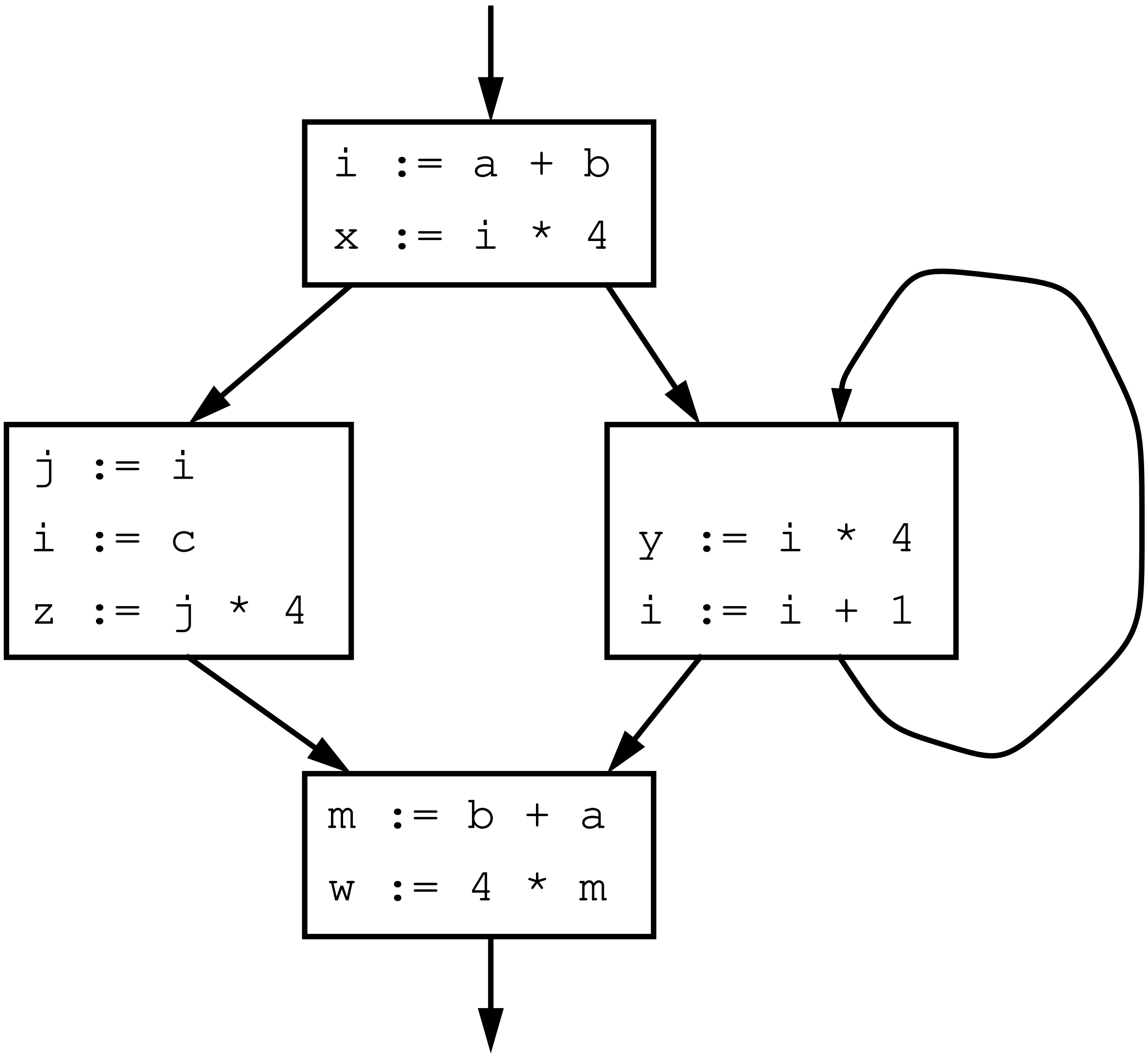
Step 2: do **copy propagation**

- 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: