CS 477: Dataflow Analysis and Abstract Interpretation

Sasa Misailovic

Based on previous slides by Saman Amarasinghe, Martin Rinard, and by Vikram Adve and Martin Vechev

University of Illinois at Urbana-Champaign

Forward Dataflow Analysis

Simulates execution of program forward with flow of control

Tuple **(G, (L, ≤), F, I)** – (graph, (lattice), transfer fs., initial val.)

For each node $n \in$ **G**, we have

- in_n – value at program point before n
- out_n – value at program point after n
- $f_n \in$ **F** – transfer function for n (given in_n, computes out_n)
- Signature of in_n, out_n, f_n : **L** $\rightarrow$ **L**

Requires that solution satisfies

- $\forall n.$ $\quad out_n = f_n(in_n)$
- $\forall n \neq n_0.$ $\quad in_n = \vee \{ out_m . m \text{ in } pred(n) \}$
- in_{n0} = **I**, summarizes information at the start of program

Dataflow Equations

Compiler processes program to obtain a set of dataflow equations

$$out_n := \quad f_n(in_n)$$
$$in_n \; := \; \vee \{ \, out_m \, . \text{ for each } m \text{ in } pred(n) \, \}$$

Conceptually separates analysis problem from program

Worklist Algorithm for Solving Forward Dataflow Equations

```
for each n do out_n := f_n(⊥)

in_n0 := I; out_n0 := f_n0(I)
worklist := N - { n_0 }

while worklist ≠ ∅ do
      remove a node n from worklist
      in_n  := ∨ { out_m . m in pred(n) }
      out_n := f_n(in_n)
      if out_n changed then
            worklist := worklist ∪ succ(n)
```

Correctness Argument

Why does the result satisfy dataflow equations?

- Whenever it processes a node n, algorithm sets $\mathtt{out}_n \mathtt{\ :=\ f}_n\mathtt{(in}_n\mathtt{)}$
  Therefore, the algorithm ensures that $out_n = f_n(in_n)$

- Whenever out_m changes, it puts succ(m) on worklist. Consider any node n $\in$ succ(m). It will eventually come off worklist and algorithm will set

  $$\mathtt{in}_n \mathtt{\ :=\ } \vee \mathtt{\ \{\ out}_m \mathtt{\ .\ m\ in\ pred(n)\ \}}$$

  to ensure that $in_n = \vee \{ out_m . m \text{ in } pred(n) \}$

- So final solution will satisfy dataflow equations

- Need also to ensure that the dataflow equalities correspond to the states in the program execution (this comes later!)

Termination Argument

Why does algorithm terminate?

Sequence of values taken on by IN_n or OUT_n is a chain. If values stop increasing, worklist empties and algorithm terminates.

If lattice has ascending chain property, algorithm terminates

- **Algorithm terminates for finite lattices**
- For lattices with infinite length, use **widening operator**
  - Detect lattice values that may be part of infinitely ascending chain
  - Artificially raise value to least upper bound of chain

Termination Argument (Details)

- For finite lattice **(L, ≤)**
- Start: each node n ∈ CFG has an initial IN set, called $\mathrm{IN}_0[n]$
- When F is **monotone,** for each n, successive values of IN[n] form a non-decreasing sequence.
  - Any chain starting at x ∈ L has at most c_x elements
  - x=IN[n] can increase in value at most c_x times
  - Then $C = \max_{n \in CFG} c_{IN[n]}$ is finite
- On every iteration, at least one IN[.] set must increase in value
  - If loop executes N × C times, all IN[.] sets would be T
  - The algorithm terminates in **O(N × C)** steps (but this is conservative)

Speed of Convergence

How quickly does the transfer function stabilize over backedge?

If the lattice has ascending chain property, then $\forall f \in F, \forall x \in L\ f^{[k]}$ stabilizes, where

$$f^{[k]} = \bigwedge_{i=0..k} f^i(x) \quad where\ f^0 = x, f^i = f \circ f^{i-1}(x)$$

F is bounded if for all *f*, the chain $\{f^{[k]}\}$ is finite, k, bounded if k≥length

K-boundness: $f^k \geq f^{[k]}$ (if L has height k, then F will be k-bounded)

Fast: (2-bounded) $f \circ f \geq f \wedge x$

Rapid (1-semibound): $\forall f \in F, \forall x, y \in L .\ f(x) \leq y \wedge x \wedge f(y)$
which ends up being $\forall f \in F, \forall x \in L .\ x \leq f(x) \wedge f(\top)$

Speed of Convergence

Loop Connectedness d(G): for a reducible CFG G, it is the maximum number of back edges in any acyclic path in G.

Kam & Ullman, 1976:

- The depth-first version of the iterative algorithm halts in at most d(G) + 3 passes over the graph
- If the lattice L has $\top$, at most d(G) + 2 passes are needed

In practice:

- d(G) < 3, so the algorithm makes less than 6 passes over the graph

For mode details, see also Properties of data flow frameworks, Marlowe and Ryder (1990)

General Worklist Algorithm

(Reminder)

```
for each n do out_n := f_n(⊥)

in_n0 := I; out_n0 := f_n0(I)
worklist := N - { n_0 }

while worklist ≠ ∅ do
      remove a node n from worklist
      in_n  := ∨ { out_m . m in pred(n) }
      out_n := f_n(in_n)
      if out_n changed then
            worklist := worklist ∪ succ(n)
```

Reaching Definitions Algorithm

(Reminder)

```
for all nodes n in N
   OUT[n] = emptyset; // OUT[n] = GEN[n];
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
      choose a node n in Changed;
      Changed = Changed - { n };

      IN[n] = emptyset;
      for all nodes p in predecessors(n)
      IN[n] = IN[n] U OUT[p];

      OUT[n] = GEN[n] U (IN[n] - KILL[n]);

      if (OUT[n] changed)
         for all nodes s in successors(n)
         Changed = Changed U { s };
```

Reaching Definitions

```
for all nodes n in N
    OUT[n] = emptyset;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
  choose a node n in Changed;
  Changed = Changed - { n };

  IN[n] = emptyset;
  for all nodes p in predecessors(n)
    IN[n] = IN[n] U OUT[p];

  OUT[n] = GEN[n] U (IN[n] - KILL[n]);

  if (OUT[n] changed)
    for all nodes s in succ(n)
       Changed = Changed U { s };
```

General Worklist

```
for each n do out_n := f_n(⊥)

in_n0 := I; out_n0 := f_n0(I)
worklist := N - { n_0 }

while worklist ≠ ∅ do
   remove a node n from worklist

   in_n  := ∨ { out_m . m in pred(n) }


   out_n := f_n(in_n)

    if out_n changed then
      worklist := worklist ∪ succ(n)
```

Reaching Definitions

P = powerset of set of all definitions in program (all subsets of set of definitions in program)

$\vee = \cup$ (order is $\subseteq$)

$\perp = \emptyset$

$I = in_{n0} = \perp$

F = all functions f of the form $f(x) = a \cup (x-b)$

- b is set of definitions that node kills
- a is set of definitions that node generates

General pattern for many transfer functions

- $f(x) = GEN \cup (x\text{-}KILL)$

Does Reaching Definitions Framework Satisfy Properties?

$\subseteq$ satisfies conditions for $\leq$

- **Reflexivity:** $x \subseteq x$
- **Antisymmetry:** $x \subseteq y$ and $y \subseteq x$ implies $y = x$
- **Transitivity:** $x \subseteq y$ and $y \subseteq z$ implies $x \subseteq z$

F satisfies transfer function conditions

- **Identity:** $\lambda x.\varnothing \cup (x - \varnothing) = \lambda x.x \in F$
- **Distributivity:** Will show $f(x \cup y) = f(x) \cup f(y)$

$$
\begin{aligned}
f(x) \cup f(y) &= (a \cup (x - b)) \cup (a \cup (y - b)) \\
&= a \cup (x - b) \cup (y - b) = a \cup ((x \cup y) - b) \\
&= f(x \cup y)
\end{aligned}
$$

Does Reaching Definitions Framework Satisfy Properties?

What about composition of F?

Given $f_1(x) = a_1 \cup (x-b_1)$ and $f_2(x) = a_2 \cup (x-b_2)$
we must show $f_1(f_2(x))$ can be expressed as $a \cup (x - b)$

$$
\begin{aligned}
f_1(f_2(x)) &= a_1 \cup ((a_2 \cup (x-b_2)) - b_1) \\
&= a_1 \cup ((a_2 - b_1) \cup ((x-b_2) - b_1)) \\
&= (a_1 \cup (a_2 - b_1)) \cup ((x-b_2) - b_1)) \\
&= (a_1 \cup (a_2 - b_1)) \cup (x-(b_2 \cup b_1))
\end{aligned}
$$

- Let $a = (a_1 \cup (a_2 - b_1))$ and $b = b_2 \cup b_1$
- Then $f_1(f_2(x)) = a \cup (x - b)$

General Result

All GEN/KILL transfer function frameworks satisfy the three properties:

- Identity
- Distributivity
- Composition

And all of them converge rapidly

Meet Over Paths* Solution

What solution would be ideal for a forward dataflow problem?

Consider a path p = $\mathbf{n_0}, n_1, \ldots, n_k, \mathbf{n}$ to a node n (note that for all i, $n_i \in \mathrm{pred}(n_{i+1})$)

The solution must take this path into account:

$$f_p(\perp) = (f_{nk}(f_{nk-1}(\ldots f_{n1}(f_{n0}(\perp)) \ldots)) \leq in_n$$

So the solution must have the property that

$$\vee\{f_p(\perp) \;.\; p \text{ is a path to } n\} \leq in(n)$$

and ideally

$$\vee\{f_p(\perp) \;.\; p \text{ is a path to } n\} = in(n)$$

**** Name exists for historical reasons; this will be a join-over-paths in our formulation for this problem. One can reformulate this with $\wedge$ ("meet") instead***

See Nielsen, Nielsen and Hankin book for more on "join" and Dragon book for the classical "meet" formalization

Soundness Proof of Analysis Algorithm

Property to prove:

For all paths p to n, $f_p(\perp) \leq in(n)$

Proof is by induction on length of p

- Uses monotonicity of transfer functions
- Uses following lemma

Lemma (we discussed the algorithm before):

Worklist algorithm produces a solution such that

$out(n) = f_n(in(n))$

if $n \in pred(m)$ then $out(n) \leq in(m)$

Proof

Base case: p is of length 1

- Then $p = n_0$ and $f_p(\perp) = \perp = in(n_0)$

Induction step:

- Assume theorem for all paths of length k
- Show for an arbitrary path p of length k+1

Induction Step Proof

$p = n_0, \ldots, n_k, n$

Must show $f_k(f_{k-1}(\ldots f_1(f_0(\perp)) \ldots)) \leq in(n)$

- By induction $(f_{k-1}(\ldots f_1(f_0(\perp)) \ldots)) \leq in(n_k)$
- Apply f_k to both sides, by monotonicity we get
  $$f_k(f_{k-1}(\ldots f_1(f_0(\perp)) \ldots)) \leq f_k(in(n_k))$$
- By lemma, $f_k(in(n_k)) = out(n_k)$
- By lemma, $out(n_k) \leq in(n)$
- By transitivity, $f_k(f_{k-1}(\ldots f_1(f_0(\perp)) \ldots)) \leq in(n)$

Distributivity

Distributivity preserves precision

If framework is distributive, then worklist algorithm produces the meet over paths solution

- For all n:

$$\vee\{f_p(\perp) \,.\, p \text{ is a path to } n\} = in_n$$

Soundness Proof of Analysis Algorithm

Connections between MOP and worklist solution:

- [Kildall, 1973] The **iterative worklist algorithm**: (1) converges and (2) computes a MFP (in our "join" case the least fixed point; in classical paper "meet", it computes the maximum fixed point) solution of the set of equations using the worklist algorithm

- [Kildall, 1973] **If F is distributive, MOP = MFP**

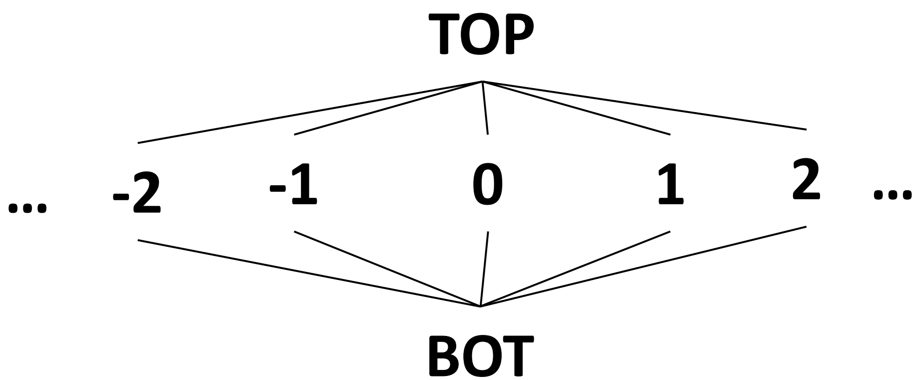
  $\vee \{f_p(\perp) \,.\, p \text{ is a path to } n\} = in_n$

- [Kam & Ullman, 1977] If **F is monotone**, **MOP ≤ MFP**
  (i.e. MFP is more conservative)

Note: if you reformulate the framework formulas with the "meet" operator, in that case MFP ≤ MOP

Lack of Distributivity Example

Constant Calculator: Flat Lattice on Integers



Actual lattice records a value for each variable

- Example element: [a→3, b→2, c→5]

Transfer function:

- If n of the form v = c, then $f_n(x) = x[v \rightarrow c]$
- If n of the form $v_1 = v_2 + v_3$, $f_n(x) = x[v_1 \rightarrow x[v_2] + x[v_3]]$

Lack of Distributivity Anomaly

a = 2
b = 3

a = 3
b = 2

[a→2, b→3]

[a→3, b→2]

[a→TOP, b→TOP]

c = a+b

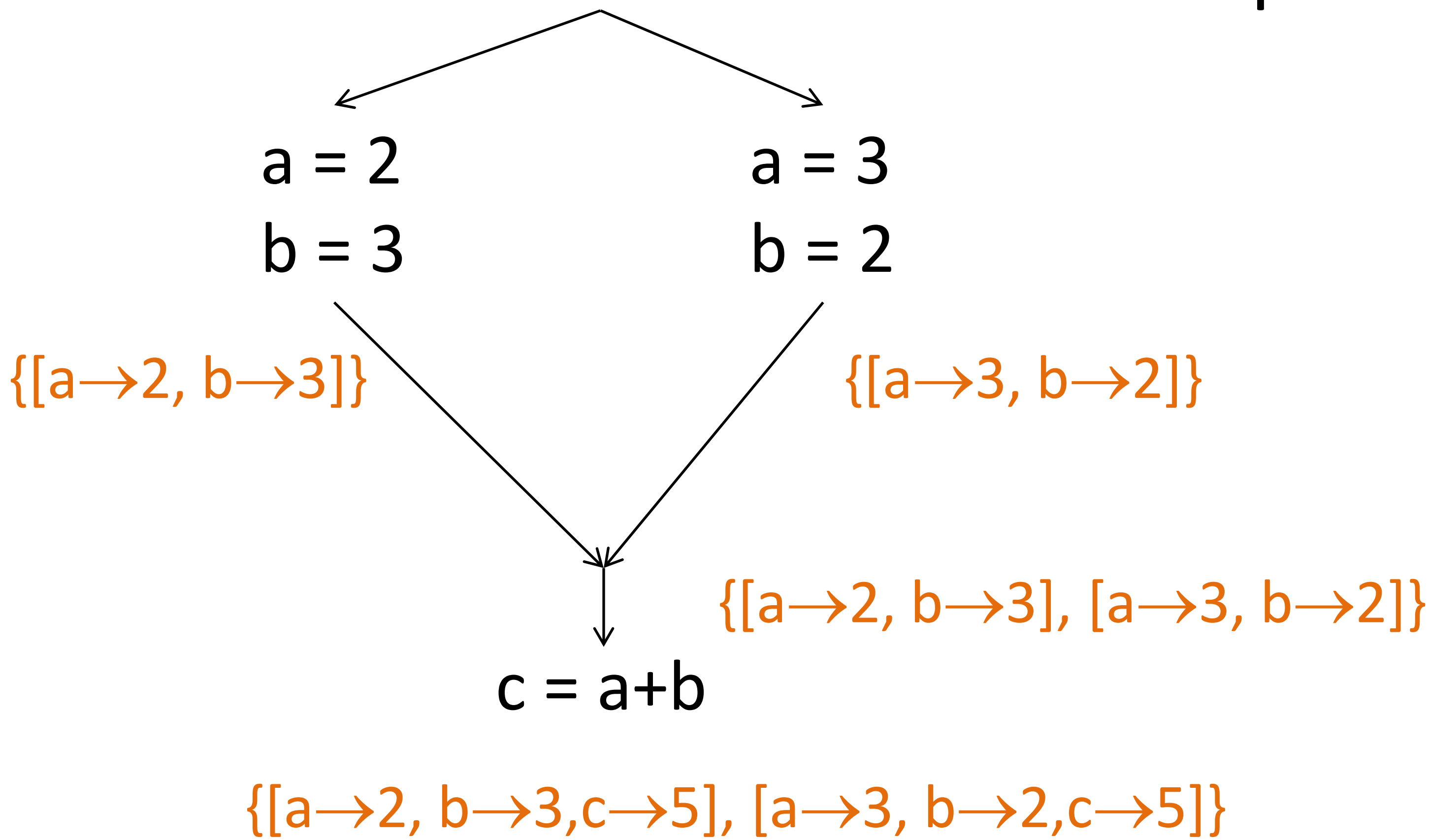
[a→TOP, b→TOP, c →TOP]

Lack of Distributivity Imprecision:
[a→TOP, b→TOP, c→5] more precise

What is the meet over all paths solution?

Make Analysis Distributive

Keep combinations of values on different paths



Discussion of the Solution

It basically simulates **all combinations** of values in **all executions**

- Exponential blowup
- Nontermination because of infinite ascending chains

Terminating solution:

- Use widening operator to eliminate blowup
  (can make it work at granularity of variables)
- However, loses precision in many cases
- Not trivial to select optimal point to do widening

Augmented Execution States

Abstraction functions for some analyses require augmented execution states

- **Reaching definitions:** states are augmented with definition that assigned each value

- **Available expressions:** states are augmented with expression for each value

Other Examples of Gen/Kill Analyses

(Optional)

Analysis: Available Expressions

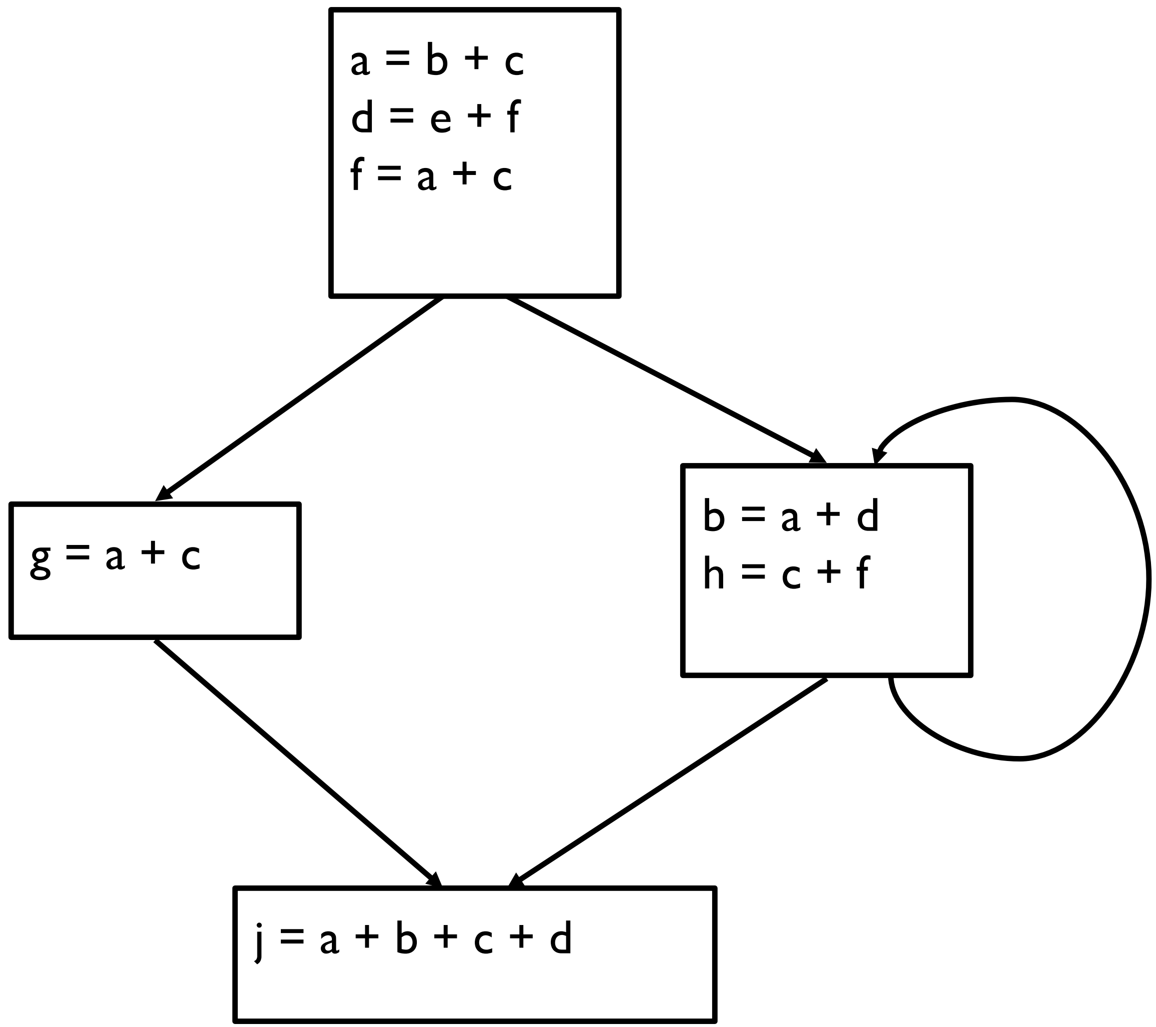
An expression x+y is available at a point p if

1. Every path from the initial node to p must evaluate x+y before reaching p,
2. There are no assignments to x or y after the expression evaluation but before p.

Available Expression information can be used to do global (across basic blocks) Subexpression Elimination

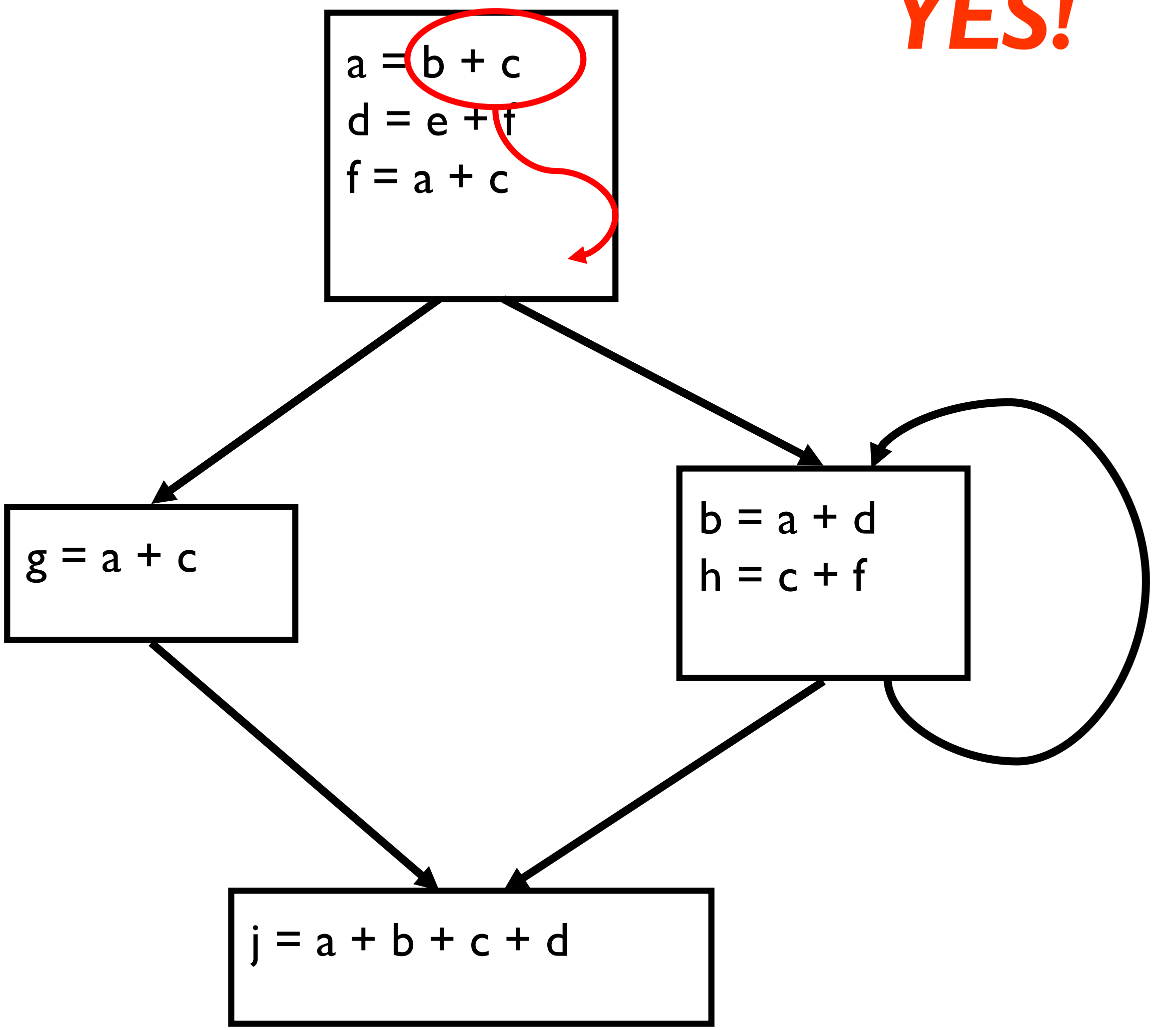
- If expression is available at use, no need to reevaluate it
- Beyond SSA-form analyses

Example: Available Expression



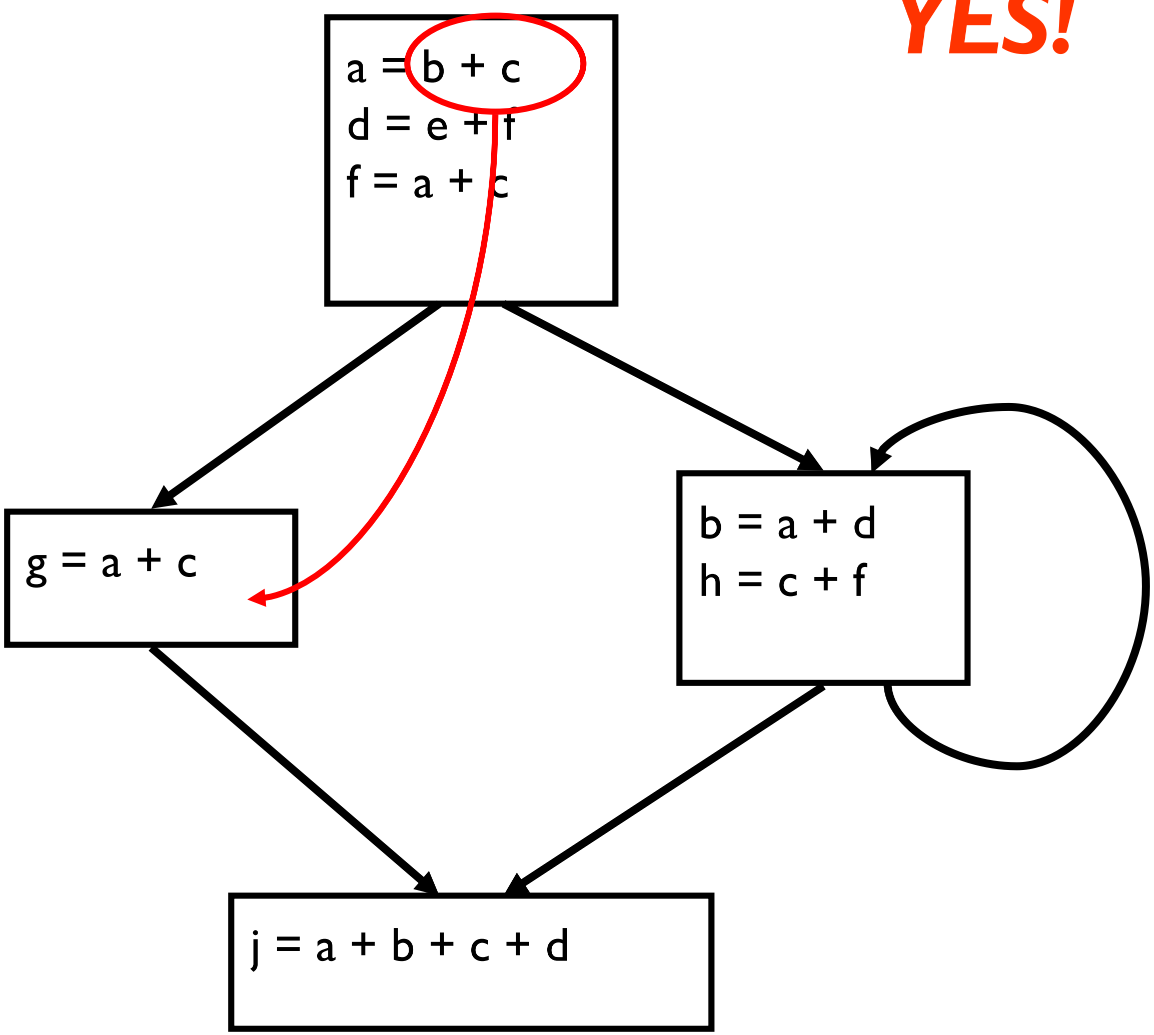
Is the Expression Available?

YES!



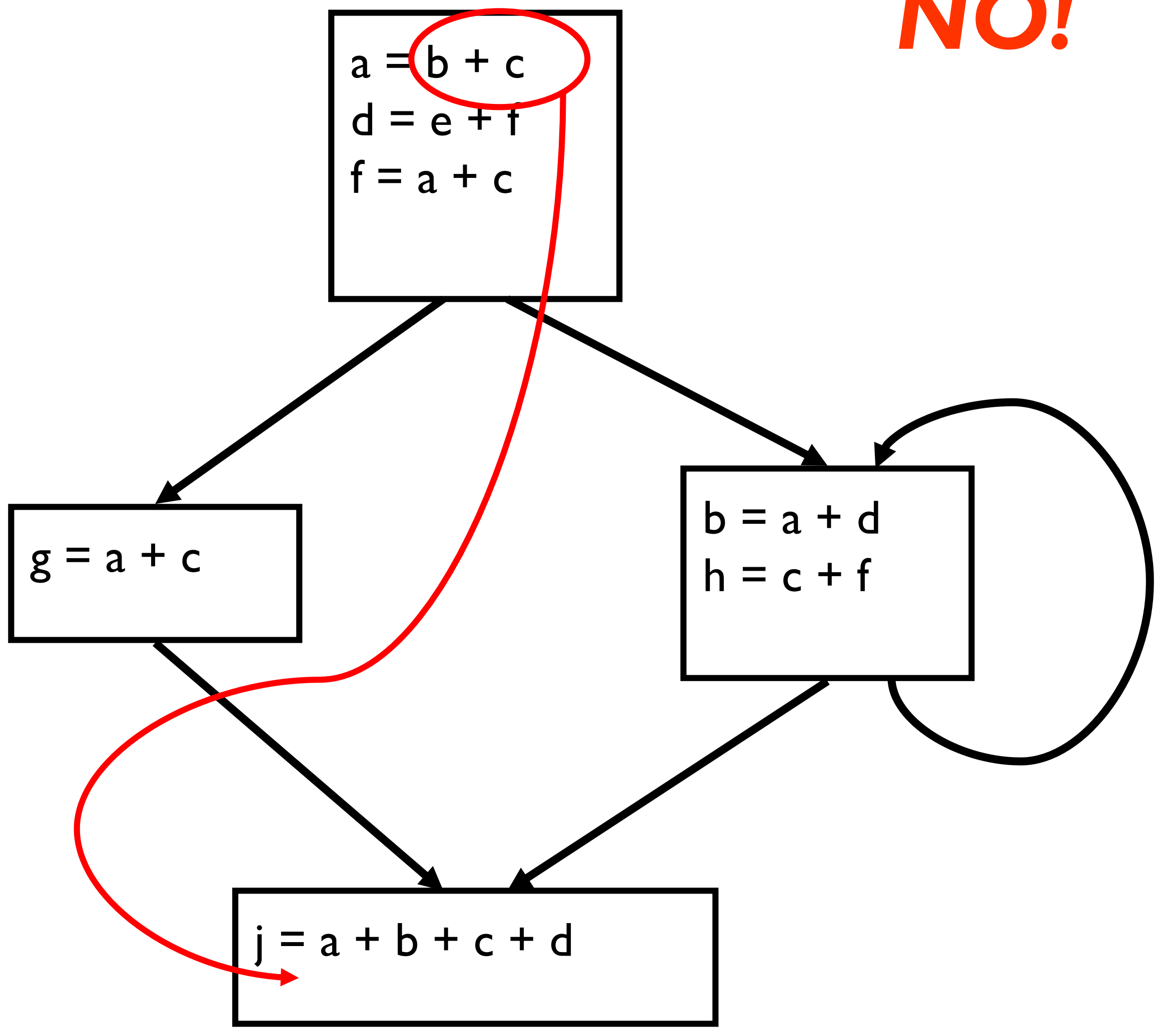
Is the Expression Available?

YES!



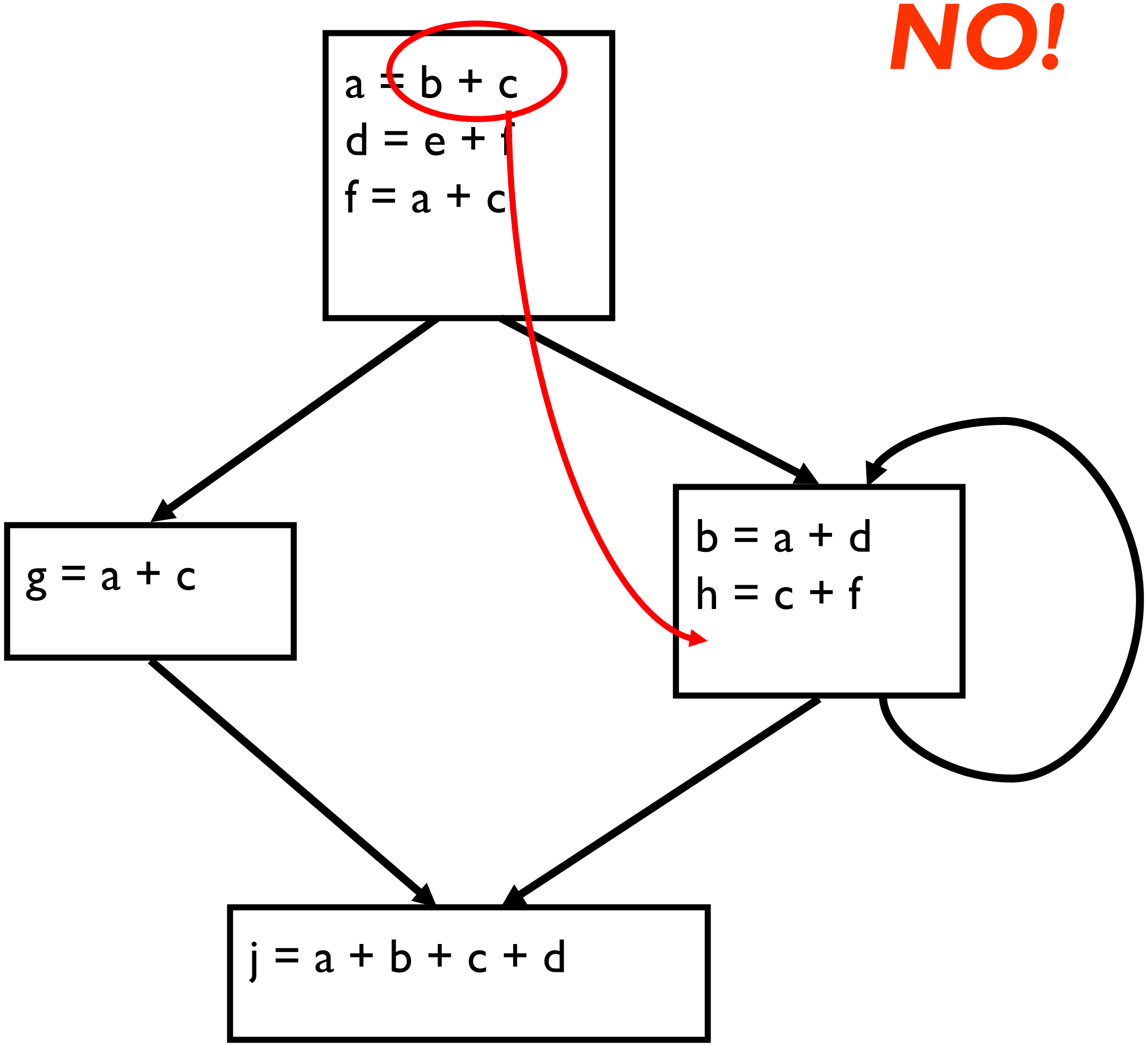
Is the Expression Available?

NO!



Is the Expression Available?

NO!



Available Expressions

P = powerset of set of all expressions in program (all subsets of set of expressions)

$\vee = \cap$ (order is $\supseteq$)

$\perp = P$

$I = in_{n0} = \emptyset$

F = all functions f of the form $f(x) = a \cup (x-b)$

- b is set of expressions that node kills
- a is set of expressions that node generates

Another GEN/KILL analysis

Concept of Conservatism

Reaching definitions use $\cup$ as join

- Optimizations must take into account all definitions that reach along **ANY path**

Available expressions use $\cap$ as join

- Optimization requires expression to be available along **ALL paths**

Optimizations must **conservatively take all possible executions into account.**

Analysis: Variable Liveness

A variable v is live at point p if

- v is used along some path starting at p, and
- no definition of v along the path before the use.

When is a variable v dead at point p?

- No use of v on any path from p to exit node, or
- If all paths from p redefine v before using v.

What Use is Liveness Information?

Register allocation.

- If a variable is dead, can reassign its register

Dead code elimination.

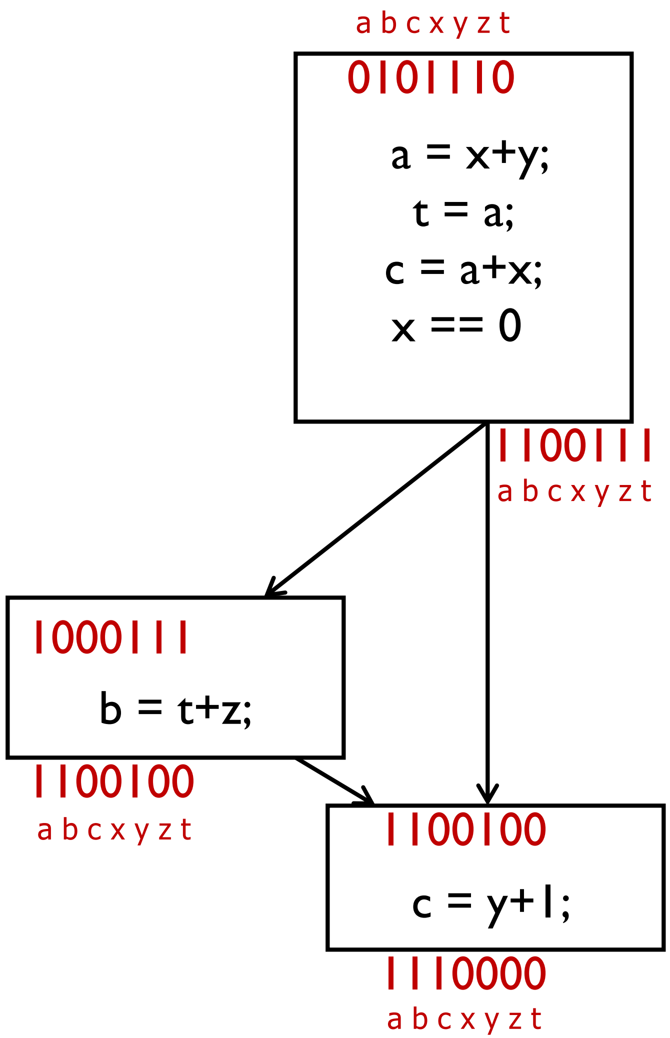
- Eliminate assignments to variables not read later.
- But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
- Can eliminate other dead assignments.
- Handle by making all externally visible variables live on exit from CFG

Conceptual Idea of Analysis

- Simulate execution
- But start from exit and go backwards in CFG
- Compute liveness information from end to beginning of basic blocks

Liveness Example

- Assume a,b,c visible outside method
  - So they are live on exit
- Assume x,y,z,t not visible outside method
- Represent Liveness Using Bit Vector
  - order is abcxyzt



Backward Dataflow Analysis

- Simulates execution of program backward against the flow of control
- For each node n, we have
  - in_n – value at program point before n
  - out_n – value at program point after n
  - f_n – transfer function for n (given out_n, computes in_n)
- Require that solution satisfies
  - $\forall n.\ in_n = f_n(out_n)$
  - $\forall n \notin N_{final}.\ out_n = \vee \{ in_m \ .\ m \text{ in } succ(n) \}$
  - $\forall n \in N_{final} = out_n = O$
  - Where O summarizes information at end of program

Worklist Algorithm for Solving Backward Dataflow Equations

```
for each n do in_n := f_n(⊥)
for each n ∈ N_final do out_n := O; in_n := f_n(out_n)
worklist := N - N_final

while worklist ≠ ∅ do
  remove a node n from worklist
  out_n := ∨ { in_m . m in succ(n) }
  in_n := f_n(out_n)
  if in_n changed then
     worklist := worklist ∪ pred(n)
```

Live Variables

P = powerset of set of all variables in program
(all subsets of set of variables in program)

$\vee = \cup$ (order is $\subseteq$)

$\perp = \emptyset$

$O = \emptyset$

F = all functions f of the form $f(x) = a \cup (x\text{-}b)$

- b is set of variables that node kills
- a is set of variables that node reads