COMP 512
Rice University
Spring 2015

Data-Flow Analysis

Dominators to Reaching Definitions

Citation numbers refer to entries in the EaC2e bibliography.

Data-flow Analysis

Definition

Data-flow analysis (DFA) *is a collection of techniques for compile-time reasoning about the run-time flow of values*

- We use the results of DFA to prove safety & identify opportunities
  - Not an end unto itself
- Almost always involves building a graph
  - Control-flow graph, call graph, or graphs derived from them
  - Sparse evaluation graphs to model flow of values *(efficiency)*
- Usually formulated as a set of *simultaneous equations*
  - Sets attached to nodes and edges
  - Often use sets with a lattice or semilattice structure

We have seen several data-flow problems.

Prior Examples

Computing LIVEOUT Sets

- Domain is the set of variable names in the procedure
- Data-flow equations define **LIVE** at the end of a block, **LIVEOUT**

Initialization: $\textbf{LIVEOUT}(n) = \emptyset, \forall n$

Fixed-point equations:

$$\textbf{LIVEOUT}(b) = \bigcup_{s \in succs(b)} (\textbf{UEVAR}(b) \cup (\textbf{LIVEOUT}(b) \cap \overline{\textbf{VARKILL}(b)}))$$

LIVE is a backward-flow problem

where

UEVAR(b) is the set of names used in b before definition in b

VARKILL(b) is the set of names defined in b

Prior Examples

Computing AVAIL Sets

- Domain is the set of expressions computed in the procedure
- Data-flow equations are more complex

Initialization:

$$\textbf{AVAIL}(n_0) = \emptyset$$

$$\textbf{AVAIL}(n) = \emptyset,\ \forall\ n \neq n_0$$

Fixed-point equations:

$$\textbf{AVAIL}(b) = \bigcap_{x \in pred(b)} (\textbf{DEEXPR}(x) \cup (\textbf{AVAIL}(x) \cap \overline{\textbf{EXPRKILL}(b)}))$$

AVAIL is a forward-flow problem

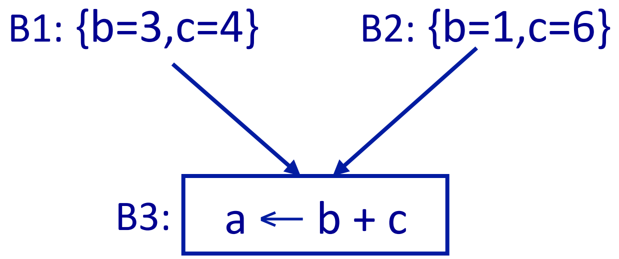
where

DEEPXR(*b*) is the set of expressions defined in *b* and not subsequently killed in *b*

EXPRKILL(*b*) is the set of expressions killed in *b* because one or more operand is redefined in *b*

Prior Examples

Global constant propagation



Function "f_3" models the effect of block B3

- $f_3(\{b=3,c=4\}) \Rightarrow \{a=7\}$
- $f_3(\{b=1,c=6\}) \Rightarrow \{a=7\}$
- $f_3(B1 \wedge B2) = f_3(\emptyset) \Rightarrow \{a=\perp\}$

Result depends on order of evaluation of the $\wedge$ operation and application of f

- First condition in admissibility

  $\forall f \in F, \forall x,y \in L, f(x \wedge y) = f(x) \wedge f(y)$
- Constant propagation is not admissible
  - ♦ Kam & Ullman time bound does not hold
  - ♦ There are tight time bounds, however, based on lattice height
  - ♦ Require a variable-by-variable formulation …

Because meet does not distribute over function application, constant propagation is not "admissible" in the Kam-Ullman sense.

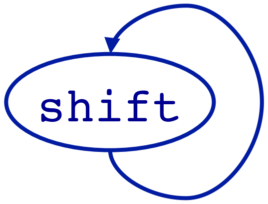
forward problem

Prior Examples

Interprocedural May Modify sets

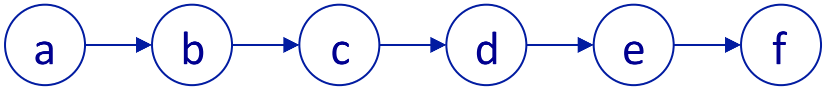
```
shift(a,b,c,d,e,f)
{
  local t;
  …
  call shift(t,a,b,c,d,e);
  f = 1;
  …
}
```

- Assume call-by-reference
- Compute the set of variables (in namespace of shift) that can be modified by a call to shift
- How long does it take?



Nothing to do with d(G)

- Iterations proportional to number of parameters
  - ♦ Not a function of the call graph
  - ♦ Can make example arbitrarily bad
- Proportional to length of chain of bindings…



Call-by-reference parameters plus recursion make the summary problems fail the Kam-Ullman “rapid” condition.

backward problem

GDFAP ≅ Global Data-Flow Analysis Problem

Proliferation of GDFAPs

In the late 1960's and the 1970's many data-flow problems were proposed

- GDFAP became the standard way to prove safety of a transformation
    - ♦ New transformation implied new GDFAP
    - ♦ Optimizing compilers spent a large fraction of compile time solving GDFAPs
    - ♦ Computers were relatively slow (1 – 10 MIPS) and small (16 to 32 MB)
    - ♦ Development of "frameworks" for GDFA
- Many papers showed a new GDFAP & a new transformation
    - ♦ Other applications arose for the GDFAP technology
    - ♦ See the papers on "DAVE" by Osterweil et al.

More GDFAPS: Very Busy Expressions

An expression *e* is <u>*very busy*</u> on exit from block *b*, *iff e* is evaluated & used along every path from *b* to n_f *and* evaluating *e* at the end of *b* would produce the same result as the next evaluation along those paths

The Plan

- Annotate each block *n* with a set *VERYBUSY(b)* that contains expressions
    - ♦ Solve data-flow equations (*standard iterative solver*)
- If *e* is in **VERYBUSY***(b)*, insert an evaluation at the end of *n* and eliminate the subsequent evaluations that it covers
    - ♦ If *e* is in **VERYBUSY***(b)* for successive blocks, want to insert it in the "right" block
    - ♦ Might be the last *b* (minimize register demand) or least frequently executed *b* (minimize dynamic number of evaluations) or …
- This optimization aims, primarily, to reduce code space

More GDFAPS: Very Busy Expressions

$|\textbf{VERYBUSY}| = |\textit{expressions}|$

Transformation: Hoisting

- *e* defined in <u>every</u> successor of *b*
- Evaluating *e* in *b* produces same result
- Saves code space, but shortens <u>no</u> path

Standard $f(x) = a \cup (b \cap c)$. $\wedge$ is $\cap$.

Data-flow problem: Very Busy Expressions

$$\textbf{VERYBUSY}(b) = \bigcap_{s \in succ(b)} (\textbf{UEEXPR}(s) \cup (\textbf{VERYBUSY}(s) \cap \overline{\textbf{EXPRKILL}(s)}))$$

$$\textbf{VERYBUSY}(n_f) = \emptyset$$

- **VERYBUSY**(*b*) contains expressions that are very busy at end of b
- **UEEXPR**(*b*) is the set of expressions used before they are killed in b
- **EXPRKILL**(*b*) is the set of expressions killed before they are used in b

VERYBUSY expressions is a **backward** flow problem

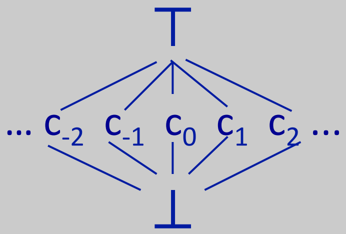
Same transfer function as **LIVE** and **AVAIL**, different meet operators.

$|CONSTANTS| = |variables|$

More GDFAPS: Constant Propagation (Classic formulation)

Transformation: Global Constant Folding

- Along every path to *p*, *v* has same known value
- Specialize computation at *p* based on *v*'s value



The Lattice *C*

Data-flow problem: Constant Propagation

Domain is the set of pairs $<v_i,c_i>$ where v_i is a variable and $c_i \in C$

$$\textbf{CONSTANTS}(b) = \wedge_{p \in preds(b)} f_p(\textbf{CONSTANTS}(p))$$

- $\wedge$ performs a pairwise meet on two sets of pairs
- $f_p(x)$ is a block specific function that models the effects of block p on the $<v_i,c_i>$ pairs in *x*

Form of *f* is quite different than in the other GDFAPs that we have seen

Constant propagation is a **forward** flow problem

More GDFAPs: Constant Folding

Meet operation is more complex than we have already seen

- $c_1 \wedge c_2 = c_1$ if $c_1 = c_2$, else $\perp$ (bottom & top as expected)

f_p does not fit into the mold of the functions in our Kam-Ullman rapid frameworks.

What about f_p ?

- If *p* has one statement then

  $x \leftarrow y$ with $CONSTANTS(p) = \{\ldots\langle x,l_1\rangle,\ldots\langle y,l_2\rangle\ldots\}$

  then $f_p(CONSTANTS(p)) = CONSTANTS(p) - \langle x,l_1\rangle + \langle x,l_2\rangle$

  $X \leftarrow y\ op\ z$ with $CONSTANTS(p) = \{\ldots\langle x,l_1\rangle,\ldots\langle y,l_2\rangle\ldots\rangle,\ldots\langle z,l_3\rangle\ldots\}$

  then $f_p(CONSTANTS(p)) = CONSTANTS(p) - \langle x,l_1\rangle + \langle x,l_2\ op\ l_3\rangle$

- If *p* has *n* statements then

  $f_p(CONSTANTS(p)) = f_n(f_{n-1}(f_{n-2}(\ldots f_2(f_1(CONSTANTS(p)))\ldots)))$

  where f_i is the function generated by the i^{th} statement in *p*

Constant propagation, in its more general forms, can become intractable because it encodes arithmetic.

Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

```
// find all the leaders, assume first op
// & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)
// build all the blocks
for i ← 0 to next – 1
    j ← leader[i] + 1
    while j ≤ n  and  j ∉ leader
        j ← j + 1
    last[i] ← j – 1
```

If **target**, **taken**, or **not_taken** are ambiguous, then we must include all labeled ops as leaders.

Sources of ambiguous targets:

- Fall-through branch path
- Jump to a register

No Ambiguity In ILOC:

All branches in **ILOC** have two explicit targets. Branches and jumps target a label rather than a register.

In the original compiler, jump to register was followed with an advisory list of labels generated when the **ILOC** was generated.

Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

```
// find all the leaders, assume first op
// & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)
// build all the blocks
for i ← 0 to next – 1
    j ← leader[i] + 1
    while j ≤ n  and  j ∉ leader
        j ← j + 1
    last[i] ← j – 1
```

EXAMPLE

```
0           a ← 4
1           t1 ← a * 4
2     L1:   t2 ← t1 / c
3           if t2 < w then goto L2
4           m ← t1 * k
5           t3 ← m + i
6     L2:   h ← i
7           m ← t3 – h
8           if t3 ≥ 0 then goto L3
9           goto L1
10    L3:   halt
```

LEADER	**0**					
LAST						

Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

```
// find all the leaders, assume first op
// & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)
// build all the blocks
for i ← 0 to next – 1
    j ← leader[i] + 1
    while j ≤ n  and  j ∉ leader
        j ← j + 1
    last[i] ← j – 1
```

EXAMPLE

```
0          a ← 4
1          t1 ← a * 4
2     L1:  t2 ← t1 / c
3          if t2 < w then goto L2
4          m ← t1 * k
5          t3 ← m + i
6     L2:  h ← i
7          m ← t3 – h
8          if t3 ≥ 0 then goto L3
9          goto L1
10    L3:  halt
```

LEADER	**0**	**6**	**4**	**9**	**10**	**2**
LAST						

Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

```
// find all the leaders, assume first op
// & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)
// build all the blocks
for i ← 0 to next – 1
    j ← leader[i] + 1
    while j ≤ n  and  j ∉ leader
        j ← j + 1
    last[i] ← j – 1
```

EXAMPLE

```
0          a ← 4
1          t1 ← a * 4
2     L1:  t2 ← t1 / c
3          if t2 < w then goto L2
4          m ← t1 * k
5          t3 ← m + i
6     L2:  h ← i
7          m ← t3 – h
8          if t3 ≥ 0 then goto L3
9          goto L1
10    L3:  halt
```

LEADER	**0**	**6**	**4**	**9**	**10**	**2**
LAST	**1**	**8**	**5**	**9**	**10**	**3**

Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

```
// find all the leaders, assume first op
// & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)
// build all the blocks
for i ← 0 to next – 1
    j ← leader[i] + 1
    while j ≤ n  and  j ∉ leader
        j ← j + 1
    last[i] ← j – 1
```

EXAMPLE

```
0         a ← 4
1         t1 ← a * 4
2   L1:   t2 ← t1 / c
3         if t2 < w then goto L2
4         m ← t1 * k
5         t3 ← m + i
6   L2:   h ← i
7         m ← t3 – h
8         if t3 ≥ 0 then goto L3
9         goto L1
10  L3:   halt
```

LEADER	**0**	**6**	**4**	**9**	**10**	**2**
LAST	**1**	**8**	**5**	**9**	**10**	**3**

Dominators

Definitions

In a flow graph, x *dominates* *y if and only if every path from the entry of the control-flow graph to the node for y includes x*

- By definition, *x* dominates *x*
- We associate a **DOM** set with each node
- |**DOM**(x)| ≥ 1

Immediate dominator

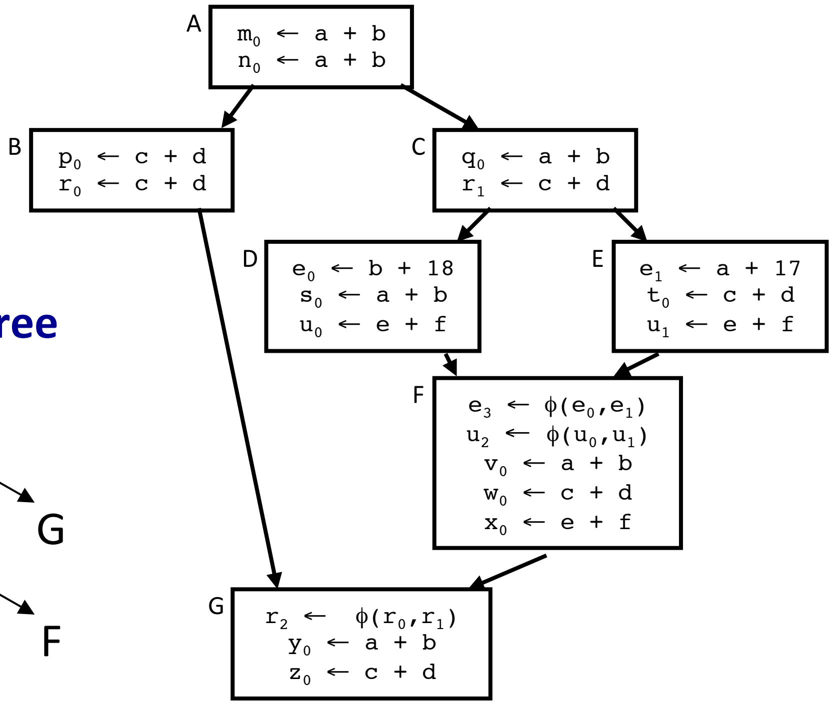
- For any node *x*, there must be a *y* in **DOM**(x) closest to *x*
  - ♦ Unless $x = n_0$, $x \neq$ **IDOM**(x)
- We call this *y* the immediate dominator of *x*
- As a matter of notation, we write this as **IDOM**(x)

Original idea: R.T. Prosser. “Applications of Boolean matrices to the analysis of flow diagrams,” *Proceedings of the Eastern Joint Computer Conference, Spartan Books, New York, pages 133-138, 1959.*

Dominators

Dominators have many uses in analysis & transformation

- Finding loops
- Building SSA form
- Making code motion decisions



A: $m_0 \leftarrow a + b$, $n_0 \leftarrow a + b$

B: $p_0 \leftarrow c + d$, $r_0 \leftarrow c + d$

C: $q_0 \leftarrow a + b$, $r_1 \leftarrow c + d$

D: $e_0 \leftarrow b + 18$, $s_0 \leftarrow a + b$, $u_0 \leftarrow e + f$

E: $e_1 \leftarrow a + 17$, $t_0 \leftarrow c + d$, $u_1 \leftarrow e + f$

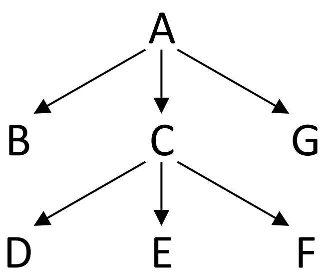
F: $e_3 \leftarrow \phi(e_0,e_1)$, $u_2 \leftarrow \phi(u_0,u_1)$, $v_0 \leftarrow a + b$, $w_0 \leftarrow c + d$, $x_0 \leftarrow e + f$

G: $r_2 \leftarrow \phi(r_0,r_1)$, $y_0 \leftarrow a + b$, $z_0 \leftarrow c + d$

Dominator sets

Block	**DOM**	**IDOM**
A	A	—
B	A,B	A
C	A,C	A
D	A,C,D	C
E	A,C,E	C
F	A,C,F	C
G	A,G	A

Dominator tree



Computing Dominators

Critical first step in SSA construction and in DVNT

- A node n dominates m iff n is on every path from n_0 to m
  - ♦ Every node dominates itself
  - ♦ n's *immediate dominator* is its closest dominator, **IDOM**(n)†

$$\textbf{DOM}(n_0) = \{ n_0 \}$$

$$\textbf{DOM}(n) = \{ n \} \cup \left(\bigcap_{p \in preds(n)} \textbf{DOM}(p) \right)$$

Initially, **DOM**(n) = N, $\forall\, n \neq n_0$.

Can do better.

Computing DOM

- These simultaneous set equations the data-flow problem
  - ♦ The simplest equations we have seen
  - ♦ Transfer function is the identity function
- Equations have a unique fixed point solution
- An iterative fixed-point algorithm will solve them quickly

† **IDOM**(n) ≠ n, unless n is n_0, by convention.

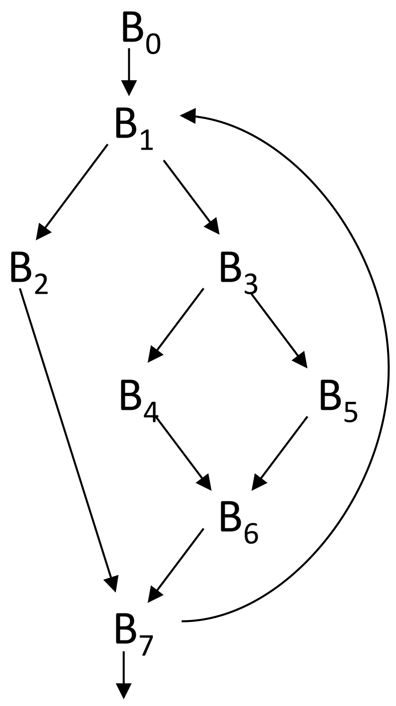
Round-robin Iterative Algorithm for DOM

```
$\mathbf{DOM}(n_0) \leftarrow n_0$

for x $\leftarrow n_1$ to $n_n$
    $\mathbf{DOM}(x) \leftarrow \{$ all nodes in graph $\}$

change $\leftarrow$ true

while (change)
    change $\leftarrow$ false
    for x $\leftarrow n_0$ to $n_n$
        $\mathbf{TEMP} \leftarrow \{ x \} \cup (\cap_{y \in pred(x)} \mathbf{DOM}(y))$
    if $\mathbf{DOM}(x) \neq \mathbf{TEMP}$ then
        change $\leftarrow$ true
        $\mathbf{DOM}(x) \leftarrow \mathbf{TEMP}$
```

Termination

- Makes sweeps over the nodes
- Halts when some sweep produces no change

DOM Example



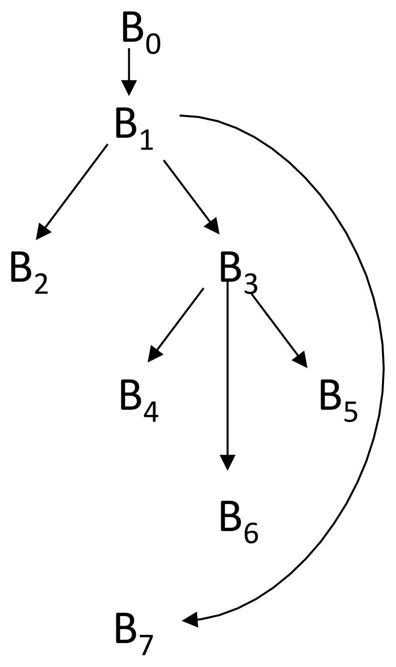
Flow Graph

Progress of iterative solution for DOM

Iter-ation	DOM(*n*)							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	*N*	*N*	*N*	*N*	*N*	*N*	*N*
1	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
2	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7

Example

If we have time, the last three slides show how to use **DOM** to improve **SVN**



Dominance Tree

Progress of iterative solution for DOM

Iter-ation	**DOM**(*n*)							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	*N*	*N*	*N*	*N*	*N*	*N*	*N*
1	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
2	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7

Results of iterative solution for DOM

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
IDOM	0	0	1	1	3	3	3	1

There are asymptotically faster algorithms.

With the right data structures, the iterative algorithm can be made extremely fast (competitive out to 10,000 or 20,000 nodes)

See Cooper, Harvey, & Kennedy [100], or § 9.5.2 in EaC2e.

Proliferation of GDFAPs

In the late 1960's and the 1970's many data-flow problems were proposed

- GDFAP became the standard way to prove safety of a transformation
  - ♦ New transformation implied new GDFAP
  - ♦ Optimizing compilers spent a large fraction of compile time solving GDFAPs
  - ♦ Computers were relatively slow (1 – 10 MIPS) and small (16 to 32 MB)
  - ♦ Development of "frameworks" for GDFA
- As transformations proliferated, need for a new paradigm emerged
  - ♦ One GDFAP that could be used for multiple transformations
  - ♦ Simplify the implementation
  - ♦ Reduce the time spent in analysis
  - ♦ The result was the development of information chains

In truth, the story is not that simple. Information chains did not arise overnight in response to an excessive number of GDFAPS; however, by the late 1980's they were being used to replace individual GDFAPs.

Information Chains

A tuple that connects 2 data-flow events is a *chain*

- Chains express data-flow relationships directly
- Chains provide a graphical representation
- Chains jump across unrelated code, simplifying search

event ≅ definition or use

We can build chains efficiently

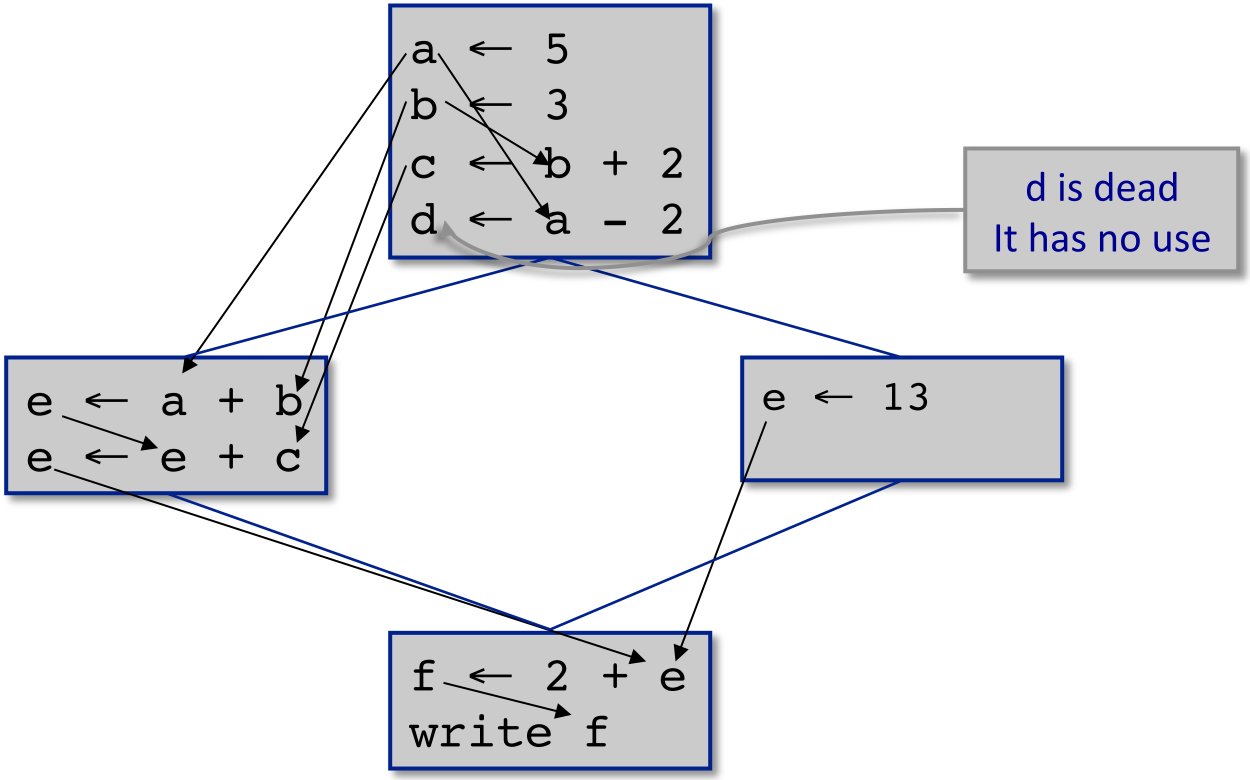
Four interesting types of chain

Source	Sink	Dependence Type
DEF	**USE**	*true, flow*
USE	**DEF**	*anti*
DEF	**DEF**	*output*
USE	**USE**	*input*

DEF-USE chains are the most common

Information Chains

Example



```
a ← 5
b ← 3
c ← b + 2
d ← a - 2
```

```
e ← a + b
e ← e + c
```

```
e ← 13
```

```
f ← 2 + e
write f
```

d is dead
It has no use

DEF-USE Chains

DEF-USE Chains form a sparse evaluation graph that we can use in many transformations.

For example, a **DEF** with no reachable use is *dead.*

Notation

Assume that, ∀ operation *i* and each variable *v*,

- **DEFS**(v,i) is the set of operations that may have defined v most recently before i, along some path in the CFG
- **USES**(v,i) is the set of operations that may use the value of v computed at i, along some path in the CFG

$$x \in \textbf{DEFS}(A,y) \Leftrightarrow y \in \textbf{USES}(A,x)$$

To construct DEF-USE chains, we solve *reaching definitions* (***YAGDFAP***)

- A definition d of some variable v *reaches* an operation i if and only if i reads v and there is a *v-clear* path from d to i
  - ♦ *v-clear* ⇒ no definition of v on the path
- Prior definition of v in same block ⇒ $|\textbf{DEFS}(v,i)| = 1$
- No prior definition ⇒ $|\textbf{DEFS}(v,i)| \geq 1$

Domain is |definitions|, same as number of operations

Reaching Definitions

The equations

$$\textbf{REACHES}(b) = \emptyset,\ \forall\ n \in N$$

$$\textbf{REACHES}(b) = \cup_{p\ \in preds(b)}\ (\textbf{DEDEF}(p) \cup (\textbf{REACHES}(p) \cap \overline{\textbf{DEFKILL}(p)}))$$

Form of *f* is same as in **LIVE**

- *REACHES(b)* is the set of definitions that reach block *b*
- *DEDEF(b)* is the set of definitions in *n* that reach the end of *b*
- *DEFKILL(b)* is the set of defs obscured by a new def in *b*

Computing *REACHES(b)*

- Use any data-flow method (*rapid framework*)
- Zadeck shows a simple linear-time algorithm

F.K. Zadeck, "Incremental data-flow analysis in a structured program editor," *Proceedings of the SIGPLAN 84 Conf. on Compiler Construction*, June, 1984, pages 132-143.

Building DEFS Sets

The Plan

1. Find basic blocks & build the CFG
2. ∀ block *b*, compute **REACHES**(*b*)

3. ∀ block *b*, ∀ operation *i*, ∀ referenced name *v*,

   Set **DEFS**(*v*,*i*) according to the earlier rule

      If there is a prior definition, *d*, of *v* in *b*

         DEFS(*v*,*i*) ← *d*

      Otherwise

         DEFS(*v*,*i*) ← {*d* | *d* defines *v* & $d \in$ **REACHES**(*b*) }

To build USES

- Invert **DEFS**, or
- Solve *reachable uses*, and use the analogous construction

Building DEF-USE Chains

Miscellany

- Domain of **REACHES** is the set of definitions ($\propto$ |operations|)
- Domain of **DEFS** & **USES** is also definitions
- Need a compact representation of **DEFS** & **USES**

Detecting Anomalies

- **DEFS**$(v,i) = \emptyset$ implies use of a <u>never initialized</u> variable
- Add a definition for each v to n_0 to detect larger set of anomalies
  - If initial def $\in$ **DEFS**(v,i) then $\exists$ a path to i with no initialization

NEXT LECTURE: using information chains & moving into **SSA**

Back to Redundancy Elimination

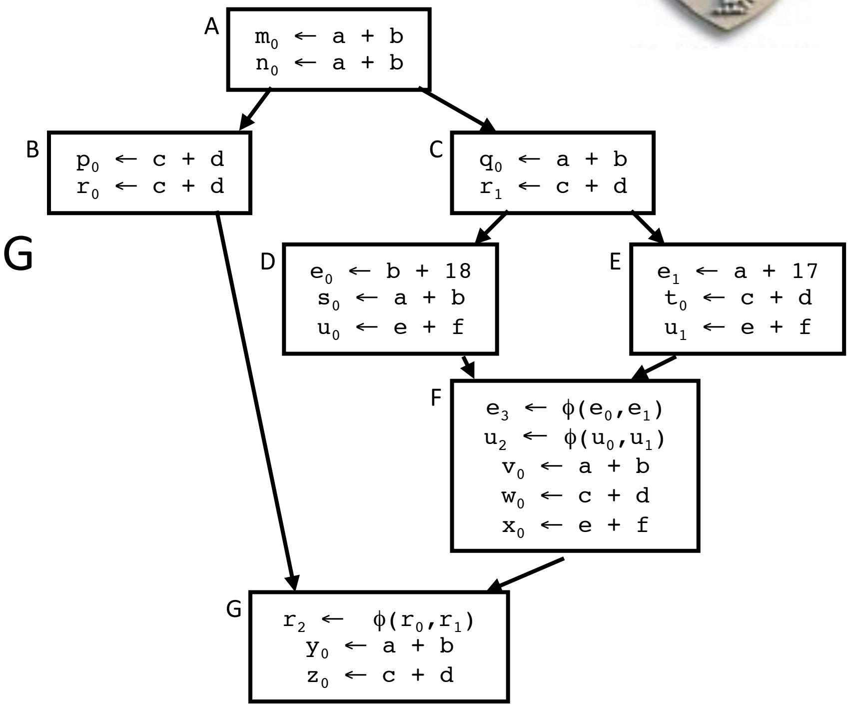
Dominators Can Improve Superlocal Value Numbering

SVN did not help with blocks F or G

- Multiple predecessors

- Must decide what facts hold in F and in G
  - For G, combine B & F?
  - Merging state is expensive
  - Fall back on what's known

- Can use table from **IDOM**(*x*) to start *x*
  - Use C for F and A for G
  - Imposes a **DOM**-based application order



A
$m_0 \leftarrow a + b$
$n_0 \leftarrow a + b$

B
$p_0 \leftarrow c + d$
$r_0 \leftarrow c + d$

C
$q_0 \leftarrow a + b$
$r_1 \leftarrow c + d$

D
$e_0 \leftarrow b + 18$
$s_0 \leftarrow a + b$
$u_0 \leftarrow e + f$

E
$e_1 \leftarrow a + 17$
$t_0 \leftarrow c + d$
$u_1 \leftarrow e + f$

F
$e_3 \leftarrow \phi(e_0,e_1)$
$u_2 \leftarrow \phi(u_0,u_1)$
$v_0 \leftarrow a + b$
$w_0 \leftarrow c + d$
$x_0 \leftarrow e + f$

G
$r_2 \leftarrow \phi(r_0,r_1)$
$y_0 \leftarrow a + b$
$z_0 \leftarrow c + d$

Leads to Dominator VN Technique (**DVNT**)

Dominator Value Numbering

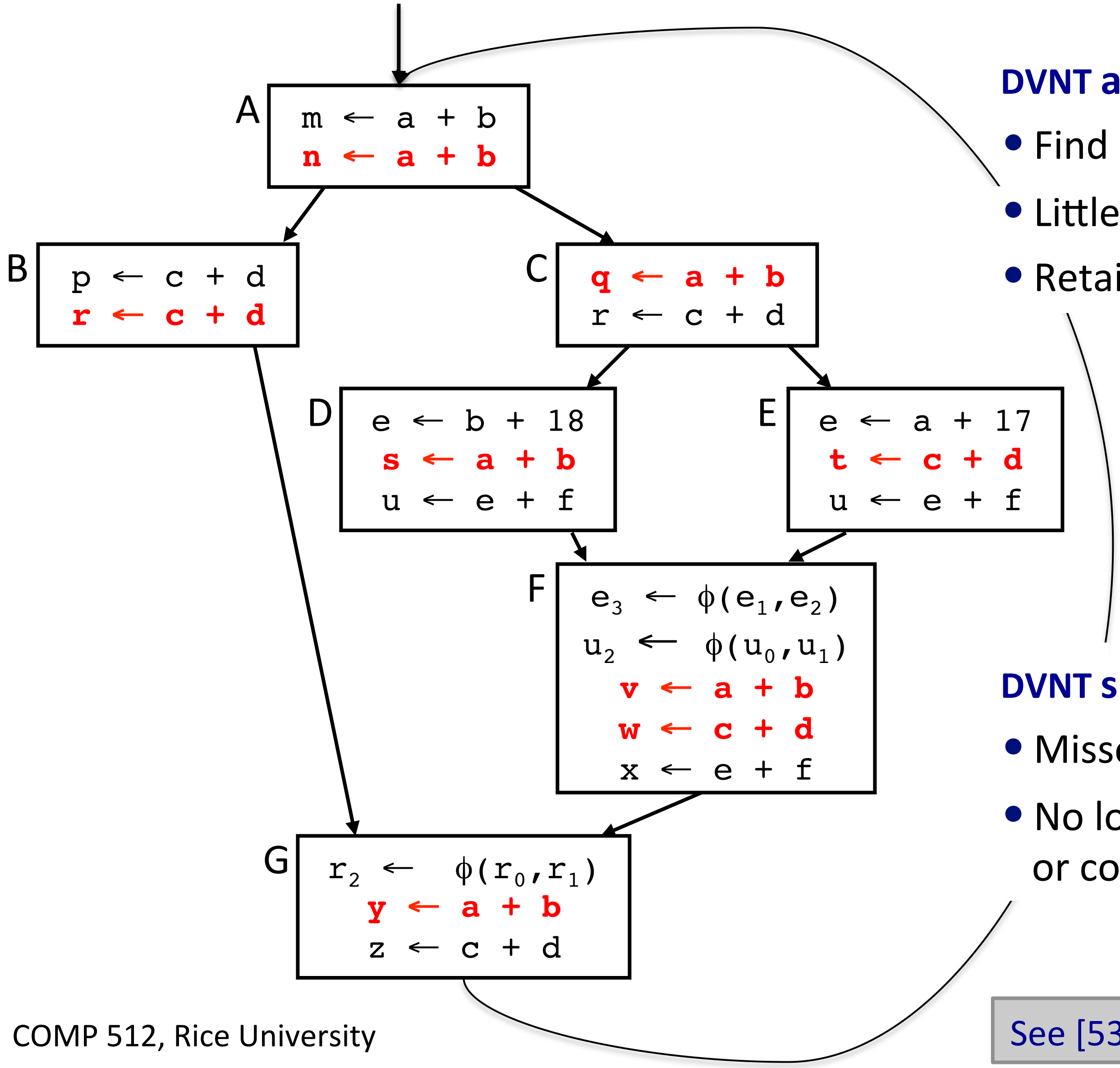
The DVNT Algorithm

- Use superlocal algorithm on extended basic blocks
  - ♦ Retain use of scoped hash tables
  - ♦ Need to use the **SSA** name space to avoid bookkeeping headaches
- Start each node with table from its **IDOM**
  - ♦ **DVNT** generalizes the superlocal algorithm
- No values flow along back edges (*i.e.,* around loops)
- Constant folding, algebraic identities as before

Larger scope leads to (*potentially*) better results

- ♦ **LVN** + **SVN** + good start for **EBBs** missed by **SVN**

Dominator Value Numbering



A

m ← a + b

n ← a + b

B

p ← c + d

r ← c + d

C

q ← a + b

r ← c + d

D

e ← b + 18

s ← a + b

u ← e + f

E

e ← a + 17

t ← c + d

u ← e + f

F

$e_3 \leftarrow \phi(e_1,e_2)$

$u_2 \leftarrow \phi(u_0,u_1)$

v ← a + b

w ← c + d

x ← e + f

G

$r_2 \leftarrow \phi(r_0,r_1)$

y ← a + b

z ← c + d

DVNT advantages

- Find more redundancy
- Little additional cost
- Retains *online* character

DVNT shortcomings

- Misses some opportunities
- No loop-carried redundancies or constants

See [53] or § 10.5.2 in EaC2e