

Data-flow analysis

Data-flow analysis

- *compile-time* reasoning about the *run-time* flow of values in the program
- represent facts about run-time behavior
- represent effect of executing each basic block
- propagate facts around control flow graph

Formulated as a set of simultaneous equations

- sets attached to the nodes and edges
- lattice to describe relation between values
- usually represented as bit or bit vector

Solve equations using iterative framework

- start with initial guess of facts at each node
- propagate until stabilizes at *maximal fixed point*
- desire *meet over all paths* solution (MOP)

Data-flow analysis

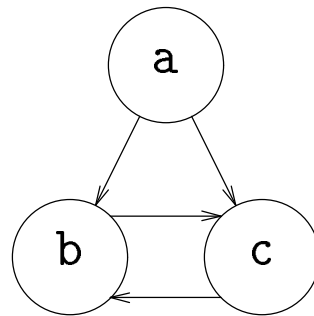
Limitations

1. *precision* — up to symbolic execution
 - no knowledge about control flow decisions
 - assume all paths are taken
2. *solution* — cannot afford MOP solution
 - class of problems where MOP = MFP
 - not all problems fit this category
3. *arrays* and *pointers* — difficult to analyze
 - precise techniques are *expensive*
 - imprecision rapidly adds up

Summary

For scalar values, we can quickly compute solutions to simple problems

Control-flow graph



Example flow graph

<i>Node Table</i>						<i>Edge Table</i>		
	Edges		Info			Number	From	To
Name	In	Out	Avail	Kill	Gen			
a	—	1	*	*	*	1	a	b
b	1	4	*	*	*	2	a	c
c	2	3	*	*	*	3	c	b
						4	b	c

Predecessors & successors have a natural (*cheap*) representation.

Available expressions

Definition

- An expression is *defined* at point p if its value is computed at p .
- An expression is *killed* at a point p if one of its argument variables is defined at p .
- an expression e is *available* at a point p in a procedure if every path leading to p contains a prior definition of e that is not killed between its definition and p .

Global common subexpression elimination

- If, at some definition point for $p \leftarrow e$, e is available with name x , we can replace the evaluation with a reference to x .
- requires a global naming scheme
- natural analog to parts of value numbering

Available expressions

For a block b

- let $AVAIL(b)$ be the set of expressions available on entry to b .
- let $KILL(b)$ be the set of expressions killed in b .
- let $GEN(b)$ be the set of expressions defined in b and not subsequently killed in b .

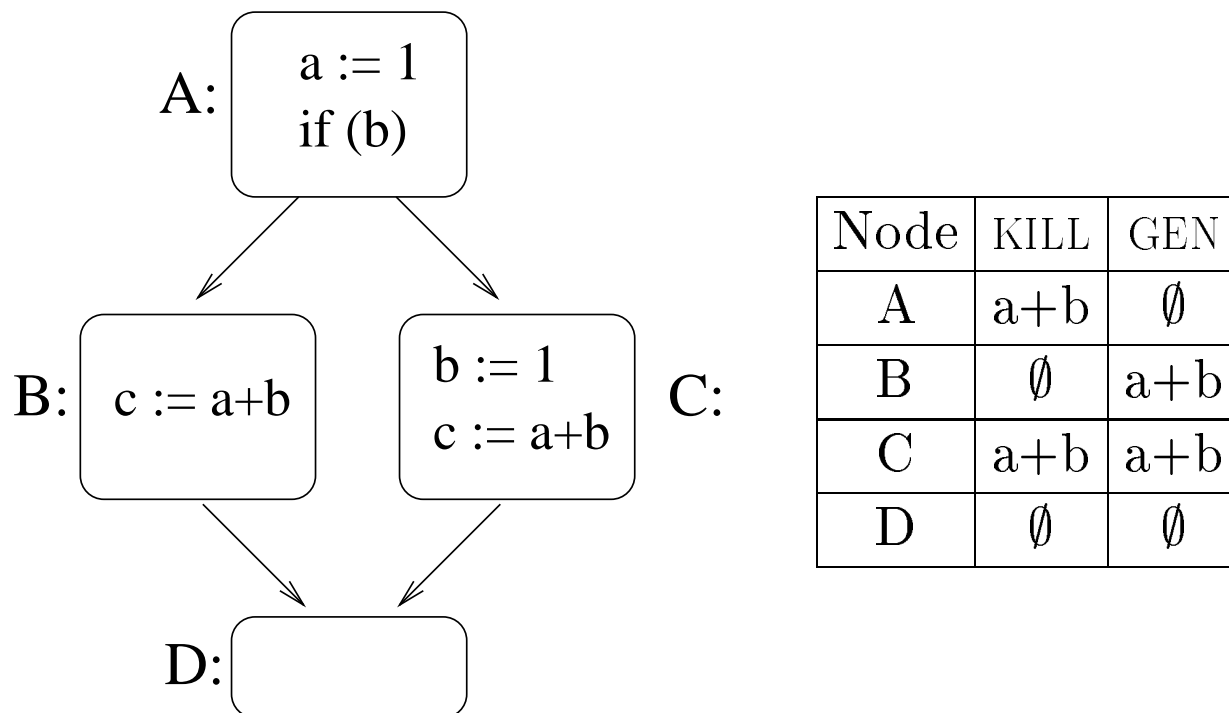
Note: $GEN(b)$ is $AVAILTAB$ from value numbering. $KILL(b)$ is harder to construct.

Now, $AVAIL$ can be defined as:

$$AVAIL(b) = \bigcap_{x \in pred(b)} (GEN(x) \cup (AVAIL(x) - KILL(x)))$$

Note: initializations must be conservative.

Available expressions example



$$\text{AVAIL}(\mathbf{A}) = \emptyset$$

$$\begin{aligned} \text{AVAIL}(\mathbf{B}) &= \text{GEN}(\mathbf{A}) \cup (\text{AVAIL}(\mathbf{A}) - \text{KILL}(\mathbf{A})) \\ &= \emptyset \cup (\emptyset - \{ a+b \}) = \emptyset \end{aligned}$$

$$\begin{aligned} \text{AVAIL}(\mathbf{C}) &= \text{GEN}(\mathbf{A}) \cup (\text{AVAIL}(\mathbf{A}) - \text{KILL}(\mathbf{A})) \\ &= \emptyset \cup (\emptyset - \{ a+b \}) = \emptyset \end{aligned}$$

$$\begin{aligned} \text{AVAIL}(\mathbf{D}) &= (\text{GEN}(\mathbf{B}) \cup (\text{AVAIL}(\mathbf{B}) - \text{KILL}(\mathbf{B}))) \cap \\ &\quad (\text{GEN}(\mathbf{C}) \cup (\text{AVAIL}(\mathbf{C}) - \text{KILL}(\mathbf{C}))) \\ &= (\{ a+b \} \cup (\emptyset - \emptyset)) \cap \\ &\quad (\{ a+b \} \cup (\emptyset - \{ a+b \})) \\ &= \{ a+b \} \end{aligned}$$

High-level view

Algorithm

1. build control flow graph *(cfg)*
2. initial (local) data gathering
3. propagate information around the graph
4. post-processing *(if needed)*

Example

<i>Facts</i>			
Node	AVAIL	KILL	GEN
a	—	—	$w+x, y \times z$
b	—	w	—
c	—	z	—

Iterative data-flow framework

A worklist iterative algorithm

```
worklist ← the set of all nodes
while( worklist ≠ ∅ )
    pick a node n from worklist
    remove n from worklist
    recompute AVAIL(n)
    if AVAIL(n) changed then
        worklist ← worklist ∪ successor(n)
```

Questions

- does this terminate?
- what answer does it compute?
- how fast (or *slow*) is it?

Gary Kildall “A unified approach to global program optimization”,
Proceedings of the First POPL, Boston, MA., October, 1973.

John Kam and Jeff Ullman, “Global data flow analysis and
iterative algorithms”, *JACM* 23(1), January, 1976.

Data-flow lattices

Definitions

1. a *semilattice* is a set L and a meet operation \wedge such that, $\forall a, b, c \in L$

(a) $a \wedge a = a$

(b) $a \wedge b = b \wedge a$

(c) $a \wedge (b \wedge c) = (a \wedge b) \wedge c$

2. \wedge imposes an order on L , $\forall a, b \in L$

(a) $a \geq b \Leftrightarrow a \wedge b = b$

(b) $a > b \Leftrightarrow a \geq b$ and $a \neq b$

3. a lattice may have a *bottom* element, denoted \perp

(a) $\forall a \in L, \perp \wedge a = \perp$

(b) $\forall a \in L, a \geq \perp$

4. a lattice may have a *top* element, denoted \top

(a) $\forall a \in L, \top \wedge a = a$

(b) $\forall a \in L, \top \geq a$

Data-flow lattices

How does this relate to data-flow analysis?

- choose a semilattice L to represent facts
- attach to each element of L a *meaning*
each $a \in L$ is a distinct set of known facts
- with each node n , associate a function
 $f_n : L \rightarrow L$ to model behavior of n

Example – AVAIL

- semilattice is 2^E , where E is the set of all expressions computed in the procedure,
and \wedge is \cap
– is \emptyset , \top is E
- for a node n , f_n has the form
$$f_n(x) = D_n \cup (x - N_n)$$
where $D_n = \text{GEN}_n$ and $N_n = \text{KILL}_n$
- the underlying graph is the flow graph
 $G = (N, E, n_0)$
 n_0 is the entry node

Data-flow analysis framework

Termination

- lattice of finite height
- node appears on worklist finite times
- iterative algorithm terminates

Correctness

- data-flow analysis results conservative
- iterative algorithm computes *maximum fixed point*
- equal to *meet over paths* for distributive frameworks

Speed

- propagate facts in reverse postorder
- define $d(G)$ as *loop connectedness*
maximum # of back edges in acyclic path in G
- stabilize in $d(G) + 3$ iterations
- in practice, $d(G)$ is less than 3 [Knuth]

Data-flow analysis

Representation

- represent a fact as a bit
- represent sets of facts as bit vectors
- meet function may be union or intersection
- operations on bit vectors

Key things to look for in a data-flow framework

- the domain and its size
- size of a single fact
- forward or backward problem
- model of characteristic function

Complexity

- distinguish bit-vector steps from logical steps
- watch out for complex mappings (GEN \rightarrow KILL)