

Why Global Dataflow Analysis?

Answer key questions at compile-time about the flow of values and other program properties over control-flow paths

Compiler fundamentals

What defs. of x reach a given use of x (and *vice-versa*)?

What $\{\langle \text{ptr}, \text{target} \rangle\}$ pairs are possible at each statement?

Scalar dataflow optimizations

Are any uses reached by a particular definition of x ?

Has an expression been computed on all incoming paths?

What is the innermost loop level at which a variable is defined?

Correctness and safety:

Is variable x defined on every path to a use of x ?

Is a pointer to a local variable live on exit from a procedure?

Parallel program optimization, program understanding, ...

Common Applications of Global Dataflow Analysis

Preliminary Analyses

- Pointer Analysis
- Detecting uninitialized variables
- Type inference
- Strength Reduction for Induction Variables

Static Computation Elimination

- Dead Code Elimination (DCE)
- Constant Propagation
- Copy Propagation

Redundancy Elimination

- Local Common Subexpression Elimination (CSE)
- Global Common Subexpression Elimination (GCSE)
- Loop-invariant Code Motion (LICM)
- Partial Redundancy Elimination (PRE)

Code Generation

- Liveness analysis for register allocation

General Approach to Dataflow Analysis

1. *Choose dataflow variables for problems of interest:*
 - Information “generated” in basic block B
 - Information “killed” in basic block B
 - Information flowing into B
 - Information flowing out of B
2. *Set up dataflow equations*
 - Transfer function for each block
 - Forward vs. backward problem
 - “Confluence” operator: \cup , \cap , other
3. *Solve iteratively*
 - Approximate execution
 - Always converges

Example: Reaching Definitions

Statement of Problem

- For each point p , what is the set of definitions that reach p ?

Definition of a Variable x

- A reference that *may store* a value into the storage location(s) named by x
- *Examples*: Assignment; FOR; I/O

Unambiguous definition : *guaranteed* to store to x

must

Ambiguous definition : *may store* to x

may

- *Ambiguity comes from aliases, procedure side effects, arrays*

What Reaching Means

- Definition d reaches point p if there is a path from the point after d to p such that d is not killed along that path.

Point: A location in a basic block just before or after some statement.

Path: A path from p_1 to p_n is a sequence of points p_1, p_2, \dots, p_n such that (intuitively) some execution can visit these points in order.

[See book for formal definition]

Kill: A definition d of variable x is killed on a path if there is an unambiguous definition of x on that path.

Dataflow Variables for Reaching Definitions

Dataflow variables (for each block B)

- $\text{Gen}(B) \equiv$ the set of defs in B that are not killed in B .
- $\text{Kill}(B) \equiv$ the set of all defs d that are killed by some def $d' \in B$
 - d need not be in B
 - If $d \in B$, then d' must occur on path from d to end of B
- $\text{In}(B) \equiv$ the set of defs that reach the point before first statement in B
- $\text{Out}(B) \equiv$ the set of defs that reach the point after last statement in B

The difference:

$\text{Gen}(B)$, $\text{Kill}(B)$ are local properties of block B alone.

$\text{In}(B)$, $\text{Out}(B)$ are global dataflow properties.

Dataflow Analysis for Reaching Definitions

Dataflow equations

$$In[S] = \text{globals, formals}$$

$$In[B] = \bigcup_{p:p \rightarrow B} Out[p], \quad B \neq S$$

$$Out[B] = Gen[B] \cup (In[B] - Kill[B])$$

Dataflow algorithms

Goal: solve these $2n$ simultaneous dataflow equations ($n = \#$ basic blocks)

- Block-structured graph (no GOTO; no BREAK from loops):
 - bottom-up evaluation, one scope at a time
- General flow-graphs:
 - iterative solution

Iterative Algorithm for Reaching Definitions

1. Initialize:

$$\begin{aligned} \text{in}[S] &= \text{globals}, \text{formals} \\ \text{in}[B] &= \emptyset && \forall B \neq S \\ \text{out}[B] &= \text{gen}[B] && \forall B \end{aligned}$$

2. Iterate until Out[B] does not change:

do

 change = false

 for each block B do

$$\text{In}[B] = \bigcup_{p:p \rightarrow B} \text{Out}[p]$$

 oldout = Out[B]

$$\text{Out}[B] = \text{Gen}[B] \cup (\text{In}[B] - \text{Kill}[B])$$

 if (oldout \neq Out[B]) change = true

 end

while (change == true)

What is the algorithm doing?

```

1 (d0)    X = ...
2        if (...)
3            ...
4        else
5 (d1)    X = ...
6        endif
7        ...
8 (d2)    X = ...
9        ...

```

```

1 (d0)    X = ...
2        while (...) {
3            ...
4            if (...) {
5 (d1)    X = ...
6            } else {
7                while (...) {
8 (d2)    X = ...
9                ...
10           }
11           ...
12        }
13 (d3)    X = ...
14        }
15        ...

```

Convergence of the Algorithm

- $Out[B]$ is finite
- $Out[B]$ never decreases for any B
⇒ must eventually stop changing
- At most n iterations if n blocks
⇐ Definitions need propagate only over acyclic paths

Available Expressions

Statement of Problem

- For each point p , what is the set of expressions available at p ?

Expression: We identify expressions *by structure*

Lexical names of variables (SSA makes this more complicated)

Operations (use reassociation, e.g., $x + y \equiv y + x$)

Available expression: Expression $e = x + y$ is available at point p if:

- every path to p evaluates e
- between the last such evaluation and p on each path, neither x nor y is modified.

Dataflow Variables for Available Expressions

Let \mathcal{U} = universal set of expressions in the program. Then:

$$in[B] = \{\epsilon \in \mathcal{U} \mid \epsilon \text{ is avail at entry to } B\}$$

$$out[B] = \{\epsilon \in \mathcal{U} \mid \epsilon \text{ is avail at exit from } B\}$$

$$e_gen[B] = \{\epsilon \in \mathcal{U} \mid \epsilon \text{ is generated by } B\}$$

$$e_kill[B] = \{\epsilon \in \mathcal{U} \mid \epsilon \text{ is killed by } B\}$$

Kill: Block B kills $x + y$ if it may assign to x or y , and it does not subsequently recompute $x + y$

Generate: Block B generates $x + y$ if it definitely evaluates $x + y$, and it does not subsequently modify x or y .

Dataflow Analysis for Available Expressions

Dataflow equations:

$$In[B] =$$

$$Out[B] =$$

Algorithm is identical to *Reaching Definitions* except:

- Confluence operator is \cap instead of \cup
- Algorithm must initialize sets as follows:

$$In[s] = \text{globals}, \text{formals}$$

$$Out[s] = e_gen[s]$$

$$Out[B] = \mathcal{U} - e_kill[B] \quad \forall B \neq s$$

Live Variables

Live Variables

Variable x is live at point p if x may be used along some path starting at p .

Use of a variable: A reference that *may read* the value of the variable.

Dataflow variables

$$def[B] = \{x \in \mathcal{V} \mid x \text{ is assigned in } B \text{ prior to use in } B\}$$

$$use[B] = \{x \in \mathcal{V} \mid x \text{ may be used in } B \text{ prior to being assigned in } B\}$$

$$in[B] = \{x \in \mathcal{V} \mid x \text{ is live at entry to } B\}$$

$$out[B] = \{x \in \mathcal{V} \mid x \text{ is live at exit from } B\}$$

Dataflow equations

$$In[B] =$$

$$Out[B] =$$

Def-Use and Use-Def Chains

Definitions

Use-Def chain or ud-chain: For each use u of a variable v , $\text{DEFS}(u)$ is the set of instructions that may have defined v last prior to u .

Def-Use chain or du-chain: For each def d of a variable v , $\text{USES}(d)$ is the set of instructions that may use the value of v computed at d

Note: $d \in \text{DEFS}(u)$ *iff* $u \in \text{USES}(d)$

Note: du-chains (or ud-chains) form a graph

Comparing with SSA

- Multiple defs reach each use, unlike SSA
- More edges in def-use graph than in SSA graph
- But fewer variable names, no ϕ functions

Computing and using du-chains and ud-chains

Construction

- Construct $DEFS(u)$ from the results of Reaching Definitions.
 - Then invert $DEFS$ to compute $USES$.
- ⇒ We can build chains very efficiently!

Some applications of chains:

- Building live ranges for graph-coloring register allocation
- Constant propagation
- Dead-code elimination
- Loop-invariant code motion

Efficient Orderings for Visiting Basic Blocks

Goal: Propagate information as far as possible in each iteration

Postorder and Reverse Postorder

- Depth-first spanning tree (DFST): tree constructed by Depth-first Search
- DFST has 3 kinds of edges: *tree edges*, *cross-edges*, *up-edges*
- Graph excluding up-edges is acyclic (DAG)
- *Postorder* (on original graph) \equiv postorder traversal of resulting DAG

Properties of Reverse Postorder

- If $B_1 \rightarrow B_2$, then B_1 is visited before B_2 , except for up-edges of DFST.
- If CFG is reducible, up-edges are exactly the back edges!
- In any case, max. # number of up-edges on any acyclic path is never more than maximum loop nesting depth

Efficiency of the Algorithm

Rule-of-thumb: Typically 5 iterations or less!
(when dataflow information propagates only over acyclic paths)

Efficient dataflow ordering

- Use Reverse Postorder (RPO) for “forward” dataflow problems
 - Use Postorder (PO) for “backward” dataflow problems
- ⇒ Information propagates “as far as possible” in each iteration, until it reaches a “retreating” DFS edge. It flows across the retreating DFS edge in the next iteration.

Rule of thumb

- Knuth [1971]: Max. #up-edges on each acyclic path is typically fewer than 3.

See Section 9.6.7 for more details.