

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

Dataflow Analysis: Our Objectives

1. To distinguish different types of dataflow problems
 - may v. must*
 - forward v. backward*
 - intersection v. union*
2. To set up and solve the dataflow equations for a basic dataflow problem
3. To identify dataflow problems needed for a given optimization

Preliminary definitions

Value, Storage location, variable, pointer : *these should be familiar*

Alias or alias pair : Two different names for the same storage location

Reference : An occurrence of a name in a program statement

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

Definition of a variable : A reference that *may store* a value into the storage location(s) named by the variable.

Examples: Assignment; FOR; input I/O

Unambiguous definition : *guaranteed* to store to X

must

Ambiguous definition : *may* store to X

may

Ambiguity comes from aliases, unpredictable side effects of procedure calls, arrays

Dataflow Analysis Basics

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

Kill of an Expression: An expression e is killed on a path if there is a possible definition of any of the variables of e on that path.

An Example Dataflow Problem: Reaching Definitions

Reaching Definitions

May or Must?

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.

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 that are killed in B
(i.e., on the path from entry to exit of B , if $d \notin B$;
or on the path from d to exit of B , if $\text{def } d \in B$).
- $\text{In}(B) \equiv$ 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

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

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

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:

$$\begin{aligned} \text{in}[B] &= \phi && \forall B \\ \text{out}[B] &= \text{gen}[B] && \forall B \end{aligned}$$

2. Iterate until $\text{Out}[B]$ does not change:

```
do
  change = false
  for each block B do
    In[B] =  $\bigcup_{p:p \rightarrow B} \text{Out}[p]$ 
    oldout = Out[B]
    Out[B] = Gen[B]  $\cup$  (In[B] - Kill[B])
    if (oldout != 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

See Section 10.9 for an example.

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 10.10 for more details.

Available Expressions

Definitions

Available expressions: $x + y$ is available at point p if:

- every path to p evaluates $x + y$
- between each such evaluation and p , neither x nor y is modified.

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 variables:

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

$$\begin{aligned} in[B] &= \{\epsilon \in \mathcal{U} \mid \epsilon \text{ is avail at entry to } B\} \\ out[B] &= \{\epsilon \in \mathcal{U} \mid \epsilon \text{ is avail at exit from } B\} \\ e_gen[B] &= \{\epsilon \in \mathcal{U} \mid \epsilon \text{ is generated by } B\} \\ e_kill[B] &= \{\epsilon \in \mathcal{U} \mid \epsilon \text{ is killed by } B\} \end{aligned}$$

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:

$$\begin{aligned} In[s] &= \phi \\ Out[s] &= e_gen[s] \\ Out[B] &= \mathcal{U} - e_kill[B] \quad \forall B \neq s \end{aligned}$$

Live Variables

Live Variables

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

Dataflow variables

$$\begin{aligned} def[B] &= \{x \in \mathcal{V} \mid x \text{ is assigned in } B \text{ prior to use in } B\} \\ use[B] &= \{x \in \mathcal{V} \mid x \text{ may be used in } B \text{ prior to being assigned in } B\} \\ in[B] &= \{x \in \mathcal{V} \mid x \text{ is live at entry to } B\} \\ out[B] &= \{x \in \mathcal{V} \mid x \text{ is live at exit from } B\} \end{aligned}$$

Dataflow equations

$$In[B] =$$

$$Out[B] =$$

General Approach to Dataflow Analysis

- Choose dataflow variables for problems of interest:

$Gen(B) \equiv$ “information” generated in block B

$Kill(B) \equiv$ “information” killed in block B

$In(B), Out(B)$

- Set up dataflow equations

Q. what is the transfer function for each block? E.g.,

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

Q. is it a forward vs. backward problem? E.g.,

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

Q. what is the “confluence” operator: \cup, \cap , other?

- Solve iteratively until convergence
Postorder or Reverse Postorder

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 $\text{DEFS}(u)$ 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