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
   - \( \text{may v. must} \)
   - \( \text{forward v. backward} \)
   - \( \text{intersection v. union} \)

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 \( \text{may read} \) the value of the variable.

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

Examples: Assignment; FOR; input I/O

Unambiguous definition: \( \text{guaranteed to store to} \ X \)

Ambiguous definition: \( \text{may store to} \ X \)

Ambiguity comes from aliases, unpredictable side effects of procedure calls
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 $p_1; p_2; \ldots; p_n$ 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.

Identifying Defs, Refs

**Examples**

1. $X = Y + 1$; // $r_{1,Y}$, $d_{1,X}$
2. $p = \text{cond? } &X : &Z$; // $d_{3,p}$ (what about $X$ and $Z$?)
3. $*p = Y + 1$; // $r_{4,Y}$, $d_{4,X}$, $d_{4,Z}$
4. // On line 54: list->next = new ListNode(...);
5. list->next->val = list->val + 1; // $r_{7,H_{54}}->val$, $d_{7,H_{54}}->val$

Principles of “naming” memory locations

- Variable names identify (sets of) memory locations
-Defs, refs apply to individual variables
-Arrays are usually named as a single variable
-Heap allocated objects can be named (i.e., treated as “dummy variables”) in different ways
- Most common: $H_k$, $k = \text{line number of malloc/new}$

An Example Dataflow Problem: Reaching Definitions

**May or Must**

$\forall p$, compute $\text{REACH}(p)$: the set of defs that reach point $p$.

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 $B$)**

- $\text{Gen}(B)$ $=$ the set of defs in $B$ that are not killed in $B$.
- $\text{Kill}(B)$ $=$ the set of all defs that are killed in $B$ (i.e., on the path from entry to exit of $B$, if def $d \in B$; or on the path from $d$ to exit of $B$, if def $d \in B$).
- $\text{In}(B)$ $=$ the set of defs that reach the point before first statement in $B$
- $\text{Out}(B)$ $=$ 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**

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

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

**Dataflow algorithms**

**Goal:** Solve these $2n$ simultaneous dataflow equations ($n = \#\text{basic blocks}$)

- 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:
   /* If there are globals or formals, in[s] ≠ φ */
   in[B] = φ ∀B
   out[B] = gen[B] ∀B

2. Iterate until Out[B] does not change:
   do change = false
      for each block B do
         In[B] = \bigcup_p: p \rightarrow B Out[p]
         oldout = Out[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 ...
10 if (...){...} else {...}
11 ...
12 ...
13 ...
14 ...
15 }
16 ...

Convergence of the Algorithm

OUT[B] must converge in a finite #iterations
- Out[B] is finite ∀B
- Out[B] never decreases for any B
  - Only KILL sets (constants) are ever subtracted from OUT sets
  - IN sets never decrease (if OUT sets never decrease)
  - But isn’t that a circular argument?

Acyclic Property
- Definitions need propagate only over acyclic paths
  - Each block only adds Gen[B], subtracts Kill[B]
  - ∪, – : only need to add, remove once
  - Must visit each block exactly once
  - Need one final iteration to check convergence

See Section 10.9 for an example.

Efficient Orderings for Visiting Basic Blocks

[Assume reducible graphs for now → Cycles “formed by” back edges]
1. No back edges: 2 iterations
2. 1 back edge (on any acyclic path): 3 iterations
3. k back edges on an acyclic path: k + 2 iterations
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 3 or fewer.

See Section 10.10 for more details.

### Available Expressions

**Definitions**
- Available expressions: $x + y$ is available at point $p$ if:
  - (a) every path to $p$ evaluates $x + y$
  - (b) between the last such evaluation and $p$ on each path, 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 $U = \text{universal set of expressions in the program}$. Then:
  - $\text{in}(B) = \{ e \in U | e \text{ is available at entry to } B \}$
  - $\text{out}(B) = \{ e \in U | e \text{ is available at exit from } B \}$
  - $\text{gen}(B) = \{ e \in U | e \text{ is generated by } B \}$
  - $\text{kill}(B) = \{ e \in U | e \text{ is killed by } B \}$

### Naming Expressions

**Examples**

```
1 a = x * y;  // eval e_1: x * y
2 b = x * y;  // eval e_1: x * y: redundant
3 x = 2;      // "kills" e_1
4 c = x * y;  // eval e_1: x * y
5 if (...) { x=5; t= x+y; }  // eval e_2: x+y
6 else { x=9; t= x+y; }      // eval e_2: x+y
7 x = x+y;    // eval e_2: x+y: redundant!
8 p = cond? &X : &Z;        // eval e_3: X+1, e_4: Y+1 may not be eval
9 ... = *p + 1;            // eval e_3: X+1, e_4: Y+1 may not be eval
10 ... = X + 1;            // eval e_3: X+1 may not be redundant
```
Dataflow Analysis for Available Expressions

Dataflow equations:

\[ \text{In}[B] = \]
\[ \text{Out}[B] = \]

Algorithm is identical to Reaching Definitions except:
- Confluence operator is \( \cap \) instead of \( \cup \)
- Algorithm must initialize sets as follows:
  \[ \text{In}[s] = \phi \]
  \[ \text{Out}[s] = \epsilon_{\text{gen}}[s] \]
  \[ \text{Out}[B] = U - \epsilon_{\text{kill}}[B] \quad \forall B \neq s \]

Live Variables

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

Dataflow variables

\[ \text{def}[B] = \{ x \in V \mid x \text{ is assigned in } B \} \]
\[ \text{use}[B] = \{ x \in V \mid x \text{ may be used in } B \} \]
\[ \text{in}[B] = \{ x \in V \mid x \text{ is live at entry to } B \} \]
\[ \text{out}[B] = \{ x \in V \mid x \text{ is live at exit from } B \} \]

Dataflow equations

\[ \text{In}[B] = \]
\[ \text{Out}[B] = \]

General Approach to Dataflow Analysis

1. Choose dataflow variables for problems of interest:
   \[ \text{Gen}(B) = \{ \text{“information” generated in block } B \} \]
   \[ \text{Kill}(B) = \{ \text{“information” killed in block } B \} \]
   \[ \text{In}[B], \text{Out}[B] \]

2. Set up dataflow equations
   - Q. what is the transfer function for each block? E.g.,
     \[ \text{Out}[B] = \text{Gen}(B) \cup (\text{In}[B] - \text{Kill}[B]) \]
   - Q. is it a forward vs. backward problem? E.g.,
     \[ \text{In}[B] = \bigcup_{p \rightarrow B} \text{Out}[p] \quad \text{or} \quad \text{Out}[B] = \bigcup_{B \rightarrow s} \text{In}[s] \]
   - Q. what is the “confluence” operator: \( \cap \), \( \cup \), other?

3. 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 \( \phi \) functions
Computing and using du-chains and ud-chains

**Construction**
- Construct $D_{efs}$ from the results of Reaching Definitions.
- Then invert $D_{efs}$ to compute $U_{ses}$.

⇒ 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