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 $\{<$ptr,target$>\}$ 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 *mus*

Ambiguous definition : *may* store to X *ma*

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 $p_1, p_2, \ldots p_n$ 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

```
X = Y + 1;                // r_1_Y,   d_1_X

p = cond? &X : &Z;        // d_3_p  (what about X and Z?)
*p = Y + 1;               // r_4_Y,   d_4_X,  d_4_Z

// On line 54:   list->next = new ListNode(...);
list->next->val = list->val + 1;  // r_7_H_54->val, d_7_H_5
```

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") i different ways
  - Most common: H$_k$, k = line number of malloc/new

An Example Dataflow Problem: Reaching Definitions

Reaching Definitions *May or Must*

$\forall p$, compute 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)

- `Gen(B)` $\equiv$ the set of defs in B that are not killed in B.
- `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 def $d \notin B$;
  or on the path from d to exit of B, if def $d \in B$).
- `In(B)` $\equiv$ the set of defs that reach the point before first statement in B
- `Out(B)` $\equiv$ the set of defs that reach the point after last statement in B

The difference:

`Gen(B)`, `Kill(B)` are *local* properties of block B alone.

`In(B)`, `Out(B)` are global dataflow properties.

Dataflow Analysis for Reaching Definitions

Dataflow equations

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

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

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:

```
/* 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] = ⋃_{p:p→B} Out[p]
        oldout = Out[B]
        Out[B] = Gen[B] ⋃ (In[B] − Kill[B])
        if (oldout != Out[B]) change = true
    end
while (change == true)
```


What is the algorithm doing?

```
(d0)    X = ...
        if (...)
            ...
        else
(d1)        X = ...
        endif
        ...
(d2)    X = ...
        ...
```

```
(d0) X = ...
     while (...) {
       ...
       if (...) {
(d1)     X = ...
       } else {
         while (...) {
(d2)       X = ...
           if (...) {...} else { ...
           ...
         }
         ...
       }
       ...
     }
       ...
```


Convergence of the Algorithm

OUT[B] must converge in a finite #iterations

- $Out[B]$ is finite $\forall B$
- $Out[B]$ 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 $Kill[B]$
  - $\bigcup$, $-$: 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 acycli 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 consructed 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

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

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

Naming Expressions

Examples

```
a = x * y;                    // eval e_1: x * y
b = x * y;                    // eval e_1: x * y: redundant
x = 2;                        // "kills" e_1
c = x * y;                    // eval e_1: x * y

if (...) { x=5; t= x+y; }     // eval e_2: x+y
else     { x=9; t= x+y; }     // eval e_2: x+y
x = x+y;                      // eval e_2: x+y: redundant!

p = cond? &X : &Z;
... = *p + 1;          // e_3: X+1, e_4: Y+1 may not be eval
... = X + 1;           // eval e_3: X+1 may not be redundant
```

Dataflow Analysis for Available Expressions

Dataflow equations:

$$In[B] \quad =$$

$$Out[B] \quad =$$

Algorithm is identical to *Reaching Definitions* except:

- Confluence operator is $\bigcap$ instead of $\bigcup$
- Algorithm must initialize sets as follows:

In[s] $= \phi$
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.

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

$$In[B] \quad =$$

$$Out[B] \quad =$$

General Approach to Dataflow Analysis

1. *Choose dataflow variables for problems of interest:*
   $Gen(B) \equiv$ "information" generated in block B
   $Kill(B) \equiv$ "information" killed in block B
   $In(B)$, $Out(B)$
2. *Set up dataflow equations*
   Q. what is the transfer function for each block? E.g.,
   $$Out[B] = Gen[B] \ \bigcup \ (In[B] - Kill[B])$$
   Q. is it a forward vs. backward problem? E.g.,
   $$In[B] = \bigcup_{p:p \to B} Out[p] \qquad \text{or} \qquad Out[B] = \bigcup_{s:B \to s} In[s]$$
   Q. what is the "confluence" operator: $\bigcup$, $\bigcap$, 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 ϕ functions

Computing and using du-chains and ud-chains

Construction

- Construct $\textsc{Defs}(u)$ from the results of Reaching Definitions.
- Then invert $\textsc{Defs}$ to compute $\textsc{Uses}$.

$\Longrightarrow$ 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