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 Basics

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

Entry(B): the point before first statement in B

Exit (B): the point after last statement in B

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.

An Example Dataflow Problem: 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:

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

Exercise

Find in, out, e_gen and e_kill for each block:

```
          int main(int argc, char** argv) {
B0:         int i, j, k, *pi = &i, *q, **pp = &pi;
            i = (argc > 1)? atoi(argv[1]) : 10;
B4:         j = i * i + 1;
            k = i + j;
            q = &k;
            if (j > i) {
B5:             int tmp=j; j = i; i = tmp;
            }
B6:         pp = &q;
            **pp = 0;
            if (i > 0)
B7:             printf("%d, %d, %d\n", i * i, j + i, j > i);
            return 0;
          }
```

Dataflow Analysis for Available Expressions

Dataflow equations:

$$In[B] = \bigcap_{p:p \to B} Out[p]$$

$$Out[B] = e\_gen[B] \bigcup (In[B] - e\_kill[B])$$

Initial values:

```
In[s]  = φ
Out[s] = e_gen[s]
Out[B]  = U  -  e_kill[B]          ∀B ≠ s
```

Iterative Algorithm for Available Expressions

1. Initialize:

```
Compute U, e_gen[B], e_kill[B], ∀B
in[B] = φ                        B = s
out[B] = e_gen[B]                B = s
out[B] = U - e_kill[B]           ∀B ≠s
```

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)
```

Efficient Orderings for Visiting Basic Blocks

Goal: Propagate information as far as possible in each iteration

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

$\Longrightarrow$ 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 fewer than 3.

See Section 10.10 for more details.

Example 2: Live Variables

Live Variables *May or Must?*

Variable x is live at point p if x [may? must?] 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

$$
\begin{aligned}
In[B] &= \\
Out[B] &=
\end{aligned}
$$

Example 3: 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.
- REACH Dataflow Problem: Compute the set of all defs (of each variable v) that [may? must?] reach the entry, exit of each basic block.

Dataflow variables (for each block B)

- `Gen(B)` $\equiv$ { $d \in B$ | d is not *subsequently*‡ killed in B }.
- `Kill(B)` $\equiv$ { d | d is not killed (or subsequently killed) in B }.
- `In(B)` $\equiv$ the set of defs that reach Entry(B)
- `Out(B)` $\equiv$ the set of defs that reach Exit(B)

‡ On path from d to Exit(B), if $d \in B$

Dataflow equations

$$In[B] =$$

$$Out[B] =$$

The primitives of the dataflow problem

References for Dataflow Analysis:

1. Muchnick, Chapter 8
2. ALS&U
3. *Flow Analysis of Computer Programs* by Matthew Hecht, North Holland Publishers, 1977.

The Primitives

1. Dataflow variables: (So far) Sets of things, *e.g.,* . . .

   Available expressions: sets of expressions

   Reaching definitions: sets of definitions

   Live variables: sets of variables

   $\Rightarrow$ What is an efficient representation for (large) sets, set operations?

2. "Confluence" or "meet" operator: E.g., $\bigcup, \bigcap$:

   $In[B] = \bigcup_{p:p\to B} Out[p], \qquad Out[B] = \bigcup_{s:B\to s} In[s]$

The primitives of the dataflow problem (cont'd)

3. Flow function (aka Transfer Function):

   Out[B] = function of In[B] *or*
   In[B] = function of Out[B]

   . . . *Example: Integer constant propagation*

   Set operations not enough: need arithmetic: $x = x + 2$

4. Solve equations iteratively until convergence

   keys to convergence: confluence operator and flow function

   key to speed: depth-first evaluation; bit-vector operations

Generalizing . . .

Q. Is there a general class of problems that can be solved in a similar framework?

Q. How could we formalize the properties of such a class?