E6998.3
Adv. Programming Languages & Compilers
Data-Flow Analysis
September 11, 2012

Lecture Outline

1. The data-flow analysis schema
2. Reaching definitions
3. Control-flow equations for reaching definitions
4. Iterative algorithm for reaching definitions
5. Live-variable analysis
6. Basic questions about the iterative algorithm
7. Semilattices
8. Data-flow analysis frameworks
9. The iterative algorithm for general frameworks
10. Meaning of a data-flow solution

1. The Data-Flow Analysis Schema

- A *data-flow value* at a program point represents the set of all possible program states that can be observed for that point, for example, all definitions in the program that can reach that point.

- Let IN[*s*] and OUT[*s*] be the set of data-flow values before and after a statement *s* in a program.

- A transfer function f_s relates the data-flow values before and after a statement *s*.

- In a forward data-flow problem

$$\text{OUT}[\ s\ ] = f_s(\text{IN}[\ s\ ])$$

  In a backward data-flow problem

$$\text{IN}[\ s\ ] = f_s(\text{OUT}[\ s\ ])$$

- A transfer function can be extended to a basic block by composing the transfer functions for all the statements in the block. Thus in a forward data-flow problem such as reaching definitions for a block B,

$$\text{OUT}[\ B\ ] = f_B(\text{IN}[\ B\ ])$$

- Given a flow graph, in a forward data-flow problem the IN set of a basic block B is computed from the OUT sets of B's predecessors:

$$\text{IN}[\ B\ ] = \cup_{P \text{ a predecessor of } B} \text{OUT}[\ P\ ]$$

- In a backward data-flow problem such as live variable analysis:

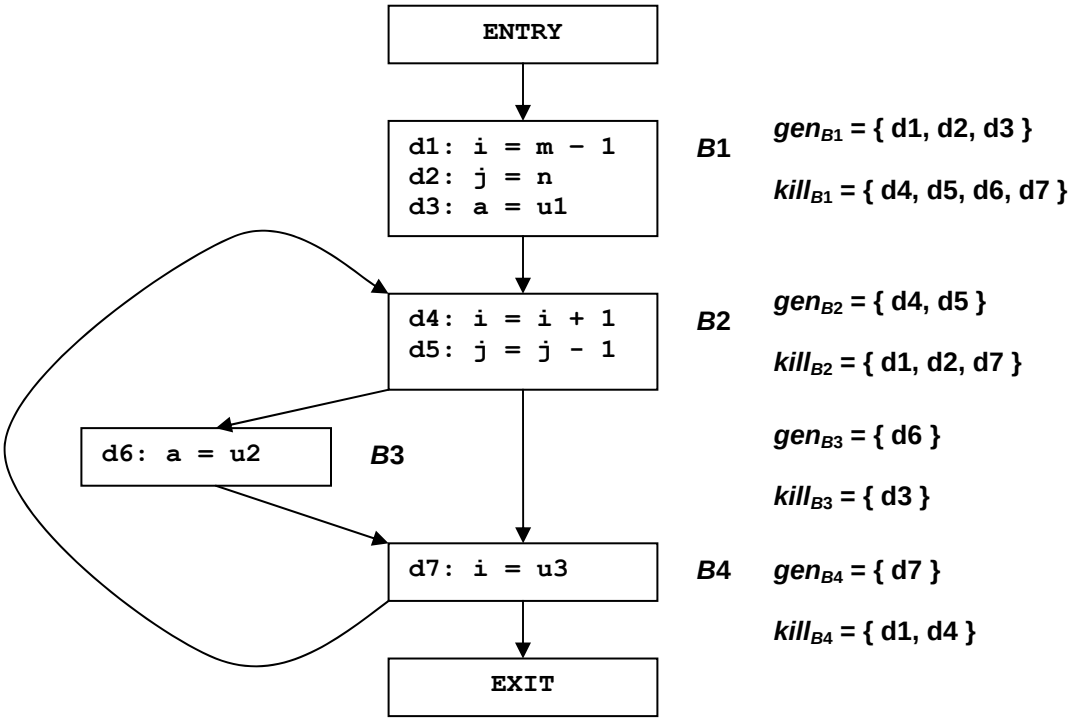
$$\text{IN}[\ B\ ] = f_B(\text{OUT}[\ B\ ])$$

$$\text{OUT}[\ B\ ] = \cup_{S \text{ a successor of } B} \text{IN}[\ S\ ]$$

- The *data-flow problem* for a flow graph is to compute the values of IN[B] and OUT[B] for all blocks B in the flow graph.

2. Reaching Definitions

- A definition *d reaches* a program point *p* if there is a path from the point immediately following *d* to *p* such that *d* is not killed along that path.
- Flow graph with *gen* and *kill* sets for each basic block:



- gen_B contains all definitions in block *B* that are visible immediately after block *B*.

- $kill_B$ is the union of all definitions killed by the statements in block *B*.

3. Control-Flow Equations for Reaching Definitions

- The reaching definitions problem is defined by the following control-flow equations:

$$\text{OUT}[\ \texttt{ENTRY}\ ] = \text{empty\_set}$$

For all blocks B other than **`ENTRY`**:

$$\text{OUT}[\ B\ ] = gen_B \cup (\ \text{IN}[\ B\ ] - kill_B\ )$$

$$\text{IN}[\ B\ ] = \cup_{P \text{ a predecessor of } B} \text{OUT}[\ P\ ]$$

4. Iterative Algorithm for Reaching Definitions

- Given a flow graph for which the *gen* and *kill* sets have been computed for each block, we can compute the set of definitions reaching the entry and exit of each block *B* using the following iterative algorithm:

```
OUT[ENTRY] = empty_set;
for (each block B other than ENTRY)
   OUT[B] = empty_set;
while (changes to any OUT occur)
   for (each block B other than ENTRY) {
      IN[B] = ∪P a predecessor of B OUT[P];
      OUT[B] = genB  ∪ (IN[B] – killB);
   }
```

- Example: Let us represent a set of definitions in the flow graph above by a bit vector. Thus **1110000** represents the set { d1, d2, d3 }. The following table represents the values taken on by the IN and OUT sets after each iteration of the while-loop of this algorithm. The superscript denotes the iteration. The initial values of 0UT, computed by the second statement of the algorithm, are indicated by the superscript 0.

Block B	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$
B1	000 0000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	111 0000	001 1100	111 0111	001 1110
B3	000 0000	001 1100	000 1110	001 1110	000 1110
B4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

IN and OUT sets for the basic blocks of flowgraph in Section 3

5. Live-Variable Analysis

- In live-variable analysis we want to determine for each variable *x* and each program point *p* whether the value *x* at *p* could be used along some path in the flow graph starting at *p*. If so, we say *x* is *live* at *p*; if not, *x* is *dead* at *p*. Live-variable analysis is crucial for register allocation.

- Live-variable analysis is an example of a backwards data-flow problem.

- Define def_B as the set of variables defined in *B* prior to any use of that variable in *B*. In the flow graph above def_{B2} is the empty set. Note that in the statement d_4: `i = i + 1`, the variable `i` is used on the right-hand side of the assignment before it is defined. Similarly, in the statement d_5: `j = j - 1`, the variable `j` is used on the right-hand side of the assignment before it is defined. It is for these reasons def_{B2} is empty.

  Define use_B as the set of variables whose values may be used in *B* prior to any definition of the variable. In the flow graph above use_{B2} = { `i, j` }.

- Data-flow equations for live-variable analysis:

$$\text{IN}[\ \textbf{EXIT}\ ] = \text{empty\_set}$$

For all blocks B other than **EXIT**:

$$\text{IN}[\ B\ ] = use_B \cup (\ \text{OUT}[\ B\ ] - def_B\ )$$

$$\text{OUT}[\ B\ ] = \cup_{S \text{ a predecessor of } B}\ \text{IN}[\ S\ ]$$

- Given a flow graph for which the *def* and *use* sets have been computed for each block, we can compute the set of variables live on entry and exit of each block B using the following iterative algorithm:

```
IN[EXIT] = empty_set;
for (each block B other than EXIT)
  IN[B] = empty_set;
while (changes to any IN occur)
  for (each block B other than EXIT)
   {
     OUT[B] = ∪S a successor of B IN[S];
     IN[B] = useB ∪ (OUT[B] − defB);
   }
```

- Unlike reaching definitions, the information flow for liveness travels backward in the data-flow graph, opposite to the direction of control flow.

- However, as for reaching definitions, live-variable analysis uses union as the “meet” operator. We are interested in whether any path with the desired properties exists, not whether something is true along all paths.

- Also as for reaching definitions, the solution to the data-flow equation is not necessarily unique. We want the solution with the smallest sets of live variables.

6. Basic Questions about the Iterative Algorithm

1) When is the iterative algorithm correct?

2) How precise is the solution?

3) Under what conditions does the iterative algorithm converge?

4) What is the meaning of the solution to the data-flow equations?

7. Semilattices

The following definitions are fundamental.

- A semilattice is a set *V* and a binary meet operator Λ such that for all *x*, *y*, *z* in *V*:
    1. x Λ *x* = *x* (meet is idempotent)
    2. *x* Λ *y* = *y* Λ *x* (meet is commutative)
    3. *x* Λ (*y* Λ *z*) = (*x* Λ *y*) Λ *z* (meet is associative)

- A semilattice has a top element TOP such that for all *x* in *V*, TOP Λ *x* = *x*.

- Optionally, a semilattice may have a bottom element BOTTOM such that for all x in V, BOTTOM Λ *x* = BOTTOM.

- The meet operator of a semilattice defines a partial order ≤ on V: $x \leq y$ if and only if $x \wedge y = x$.

- Greatest lower bound
  - A greatest lower bound (glb) of x and y is an element g such that
    1. $g \leq x$,
    2. $g \leq y$, and
    3. If z is any element such that $z \leq x$ and $z \leq y$, then $z \leq g$.

8. Data-Flow Analysis Frameworks

- A data-flow analysis framework (D, V, Λ, F) consists of
  1. A direction D of the the data flow. D can be forwards or backwards.
  2. A semilattice (V, Λ).
  3. A family F of transfer functions from V to V.

- The family F of transfer functions has two properties:
  1. F has an identity function I such that I(x) = x for all x in V.
  2. F is closed under composition; i.e., for f and g in F, the function h defined by h(x) = g(f(x)) is in F.

- A framework is monotone if for all x and y in V and f in F, $x \leq y$ implies $f(x) \leq f(y)$.

- A framework is distributive if for all x and y in V and f in F, $f(x \Lambda y) = f(x) \Lambda f(y)$.

9. The Iterative Algorithm for General Frameworks

- The iterative algorithm takes as input:
  - A data-flow graph with an ENTRY and EXIT node.
  - A direction, forward or backward.
  - A set of values V for IN and OUT.
  - A meet operator Λ on V such that (V, Λ) forms a semilattice.
  - A set of transfer functions for the blocks of the data-flow graph.
  - A constant value for the boundary condition: for forward frameworks v_{ENTRY}; for backward framewors v_{EXIT}.

- The iterative algorithm produces as output the sets IN[B] and OUT[B] for each block B in the data-flow graph.

- A maximum fixed point (MFP) is a solution with the property that in any other solution, the values of IN[B] and OUT[B] are ≤ the corresponding values of the MFP.

- The iterative algorithm for a forward data-flow problem:

```
OUT[EXIT] = v_ENTRY;
for (each block B ≠ EXIT)
   OUT[B] = TOP;
while (changes to any OUT occur)
   for (each block B ≠ ENTRY) {
      IN[B] = Λ_{P a predecessor of B} OUT[P];
      OUT[B] = f_B(IN[B]);
   }
```

- f_B is the transfer function for block B.

- The iterative algorithm for a backward data-flow problem:

```
IN[EXIT] = v_EXIT;
for (each block B ≠ EXIT)
   IN[B] = TOP;
while (changes to any IN occur)
   for (each block B ≠ EXIT) {
      OUT[B] = Λ_{S a successor of B} IN[S];
      IN[B] = f_B(OUT[B]);
   }
```

- f_B is the transfer function for block B.

- Important properties of the algorithm:
  - If the algorithm converges, the result is a solution to the data-flow equations.
  - If the framework is monotone, the solution is the MFP of the data-flow equations.
  - If the semilattice of the framework is monotone and of finite height, the algorithm is guaranteed to converge.

10. Meaning of a Data-Flow Solution

- For a forward data-flow framework, the IDEAL[B] solution for a block B is the meet over all feasible paths P of $f_P(v_{ENTRY})$.
  - Any answer greater than IDEAL is incorrect.
  - Any answer smaller than or equal to IDEAL is safe.

- MOP[B], the meet over all paths solution, assumes that every path in the data-flow graph can be taken, but some of these paths may not be feasible. Thus, MOP[B] ≤ IDEAL[B] for all blocks B.

- The MFP solution is always smaller than or equal to the MOP solution. Therefore, MFP ≤ MOP ≤ IDEAL.

11. Reference

- Compilers: *Principles, Techniques, & Tools* (second edition), Aho, Lam, Sethi, Ullman, Chapter 9.

aho@cs.columbia.edu