Lecture 4

Introduction to Data Flow Analysis

I. Structure of data flow analysis

II. Example 1: Reaching definition analysis

III. Example 2: Liveness analysis

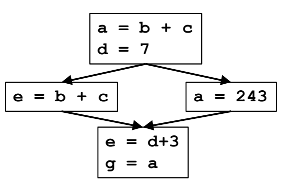
IV. Generalization

What is Data Flow Analysis?

- **Local analysis (e.g. value numbering)**
  - analyze effect of each instruction
  - compose effects of instructions to derive information from beginning of basic block to each instruction
- **Data flow analysis**
  - analyze effect of each basic block
  - compose effects of basic blocks to derive information at basic block boundaries
  - from basic block boundaries, apply local technique to generate information on instructions

What is Data Flow Analysis? (Cont.)

- **Data flow analysis:**
  - Flow-sensitive: sensitive to the control flow in a function
  - intraprocedural analysis
- **Examples of optimizations:**
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination

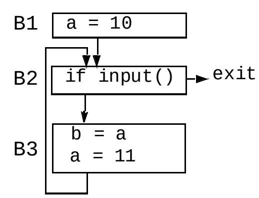


Value of x?

Which "definition" defines x?

Is the definition still meaningful (live)?

Static Program vs. Dynamic Execution



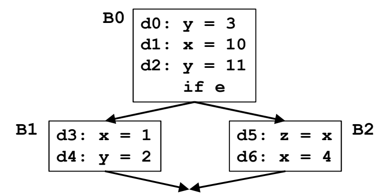
- **Statically**: Finite program
- **Dynamically**: Can have infinitely many possible execution paths
- **Data flow analysis abstraction:**
  - For each point in the program: combines information of all the instances of the same program point.
- **Example of a data flow question:**
  - Which definition defines the value used in statement "b = a"?

Effects of a Basic Block

- Effect of a statement: `a = b+c`
  - **Uses** variables (b, c)
  - **Kills** an old definition (old definition of a)
  - new **definition** (a)
- Compose effects of statements -> Effect of a basic block
  - A **locally exposed use** in a b.b. is a use of a data item which is not preceded in the b.b. by a definition of the data item
  - any definition of a data item in the basic block **kills** all definitions of the same data item reaching the basic block.
  - A **locally available definition** = last definition of data item in b.b.

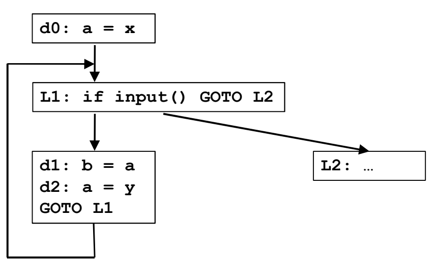
```
t1 = r1+r2
r2 = t1
t2 = r2+r1
r1 = t2
t3 = r1*r1
r2 = t3
if r2>100 goto L1
```


II. Reaching Definitions

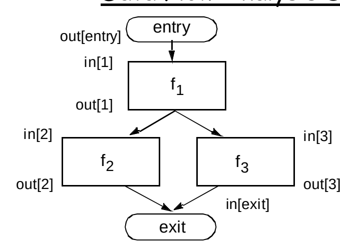


- **Every assignment** is a **definition**
- A **definition** *d* **reaches** a point *p* if **there exists** path from the point immediately following *d* to *p* such that *d* is **not killed** (overwritten) along that path.
- Problem statement
  - For each point in the program, determine if each definition in the program reaches the point
  - A bit vector per program point, vector-length = #defs

Reaching Definitions: Another Example

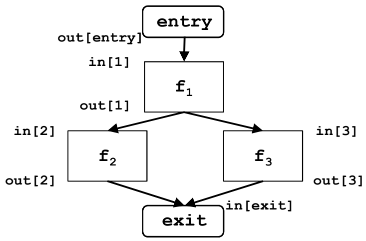


Data Flow Analysis Schema



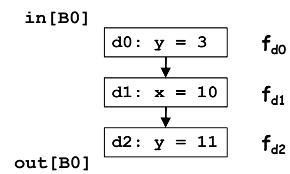
- Build a flow graph (nodes = basic blocks, edges = control flow)
- Set up a set of equations between in[b] and out[b] for all basic blocks b
  - Effect of code in basic block:
    - Transfer function f_b relates in[b] and out[b], for same b
  - Effect of flow of control:
    - relates out[b_1], in[b_2] if b_1 and b_2 are adjacent
- Find a solution to the equations

Data Flow Analysis Schema



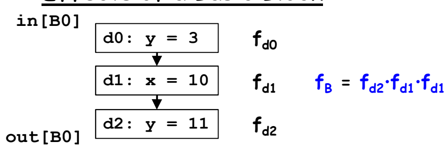
- Build a flow graph (nodes = basic blocks, edges = control flow)
- Set up a set of equations between in[b] and out[b] for all basic blocks b
  - Effect of code in basic block:
    - Transfer function f_b relates in[b] and out[b], for same b
  - Effect of flow of control:
    - relates out[b_1], in[b_2] if b_1 and b_2 are adjacent
- Find a solution to the equations

Effects of a Statement



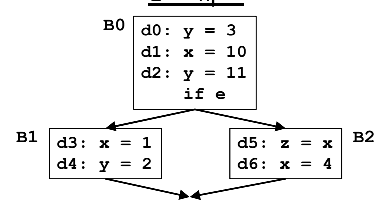
- f_s : A transfer function of a statement
  - abstracts the execution with respect to the problem of interest
- For a statement s (d: x = y + z)
  out[s] = f_s(in[s]) = Gen[s] U (in[s]-Kill[s])
  - **Gen[s]**: definitions generated: Gen[s] = {d}
  - **Propagated** definitions: in[s] - Kill[s],
    where **Kill[s]**=set of all other defs to x in the rest of program

Effects of a Basic Block



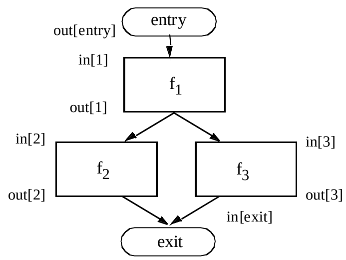
- Transfer function of a statement s:
  - out[s] = f_s(in[s]) = Gen[s] U (in[s]-Kill[s])
- Transfer function of a basic block B:
  - Composition of transfer functions of statements in B
- out[B] = f_B(in[B]) = $f_{d2} f_{d1} f_{d0}$(in[B])
  = Gen[d_2] U (Gen[d_1] U (Gen[d_0] U (in[B]-Kill[d_0]))-Kill[d_1])) -Kill[d_2]
  = Gen[d_2] U (Gen[d_1] U (Gen[d_0] - Kill[d_1]) - Kill[d_2]) U
  in[B] - (Kill[d_0] U Kill[d_1] U Kill[d_2])
  = Gen[B] U (in[B] - Kill[B])
  - Gen[B]: locally exposed definitions (available at end of bb)
  - Kill[B]: set of definitions killed by B

Example



- a **transfer function** f_b of a basic block b:
  OUT[b] = f_b(IN[b])
  incoming reaching definitions -> outgoing reaching definitions
- A basic block b
  - **generates** definitions: Gen[b],
    - set of locally available definitions in b
  - **kills** definitions: in[b] - Kill[b],
    where Kill[b]=set of defs (in rest of program) killed by defs in b
- **out[b] = Gen[b] U (in(b)-Kill[b])**

Effects of the Edges (acyclic)



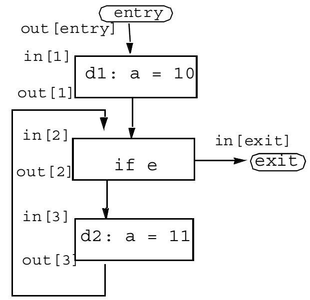
f	Gen	Kill
1	{1,2}	{3,4,6}
2	{3,4}	{1,2,6}
3	{5,6}	{1,3}

- $out[b] = f_b(in[b])$
- Join node: a node with multiple predecessors
- **meet** operator:

  $in[b] = out[p_1] \cup out[p_2] \cup \ldots \cup out[p_n]$, where

  $p_1, \ldots, p_n$ are all predecessors of b

Cyclic Graphs



- Equations still hold
  - $out[b] = f_b(in[b])$
  - $in[b] = out[p_1] \cup out[p_2] \cup \ldots \cup out[p_n]$, $p_1, \ldots, p_n$ pred.
- Find: fixed point solution

Reaching Definitions: Iterative Algorithm

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
   out[Entry] = ∅

// Initialization for iterative algorithm
   For each basic block B other than Entry
      out[B] = ∅

// iterate
   While (Changes to any out[] occur) {
      For each basic block B other than Entry {
         in[B] = ∪ (out[p]), for all predecessors p of B
         out[B] = f_B(in[B])     // out[B]=gen[B]∪(in[B]-kill[B])
      }
```

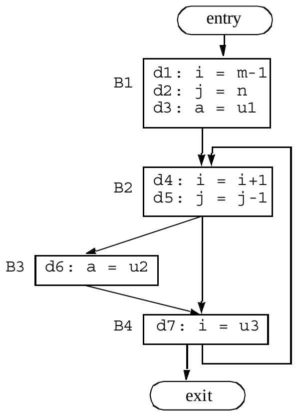

Reaching Definitions: Worklist Algorithm

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Initialize
    out[Entry] = ∅          // can set out[Entry] to special def
                            // if reaching then undefined use
    For all nodes i
        out[i] = ∅          // can optimize by out[i]=gen[i]
    ChangedNodes = N

// iterate
    While ChangedNodes ≠ ∅ {
        Remove i from ChangedNodes
        in[i] = U (out[p]), for all predecessors p of i
        oldout = out[i]
        out[i] = f_i(in[i])     // out[i]=gen[i]U(in[i]-kill[i])
        if (oldout ≠ out[i]) {
            for all successors s of i
                add s to ChangedNodes
        }
    }
```

Example



III. Live Variable Analysis

- **Definition**
  - A variable **v** is **live** at point *p* if
    - the value of **v** is used along some path in the flow graph starting at *p*.
  - Otherwise, the variable is **dead**.
- **Motivation**
  - e.g. register allocation

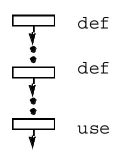
```
for i = 0 to n
  … i …
…
for i = 0 to n
  … i …
```

- **Problem statement**
  - For each basic block
    - determine if each variable is live in each basic block
  - Size of bit vector: one bit for each variable

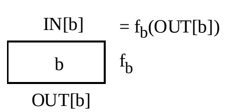
Effects of a Basic Block (Transfer Function)

- **Insight: Trace uses backwards to the definitions**

an execution path



control flow



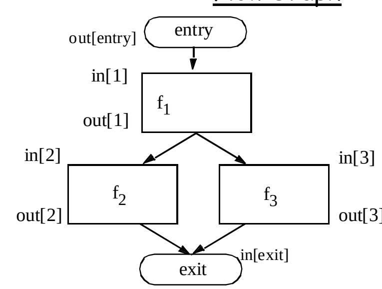
example

```
d3: a = 1
d4: b = 1

d5: c = a
d6: a = 4
```

- **A basic block b can**
  - generate live variables: **Use[b]**
    - set of locally exposed uses in b
  - propagate incoming live variables: **OUT**[b] - **Def[b]**,
    - where Def[b]= set of variables defined in b.b.
- **transfer function** for block b:
  in[b] = Use[b] U (out(b)-Def[b])

Flow Graph



f	Use	Def
1	{e}	{a,b}
2	{}	{a,b}
3	{a}	{a,c}

- in[b] = f_b(out[b])
- Join node: a node with multiple successors
- **meet** operator:
  out[b] = in[s_1] U in[s_2] U ... U in[s_n], where
  s_1, ..., s_n are all successors of b

Liveness: Iterative Algorithm

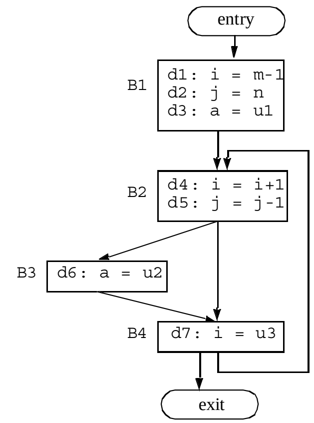
```
input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
   in[Exit] = ∅

// Initialization for iterative algorithm
   For each basic block B other than Exit
      in[B] = ∅

// iterate
   While (Changes to any in[] occur) {
      For each basic block B other than Exit {
         out[B] = ∪ (in[s]), for all successors s of B
         in[B] = f_B(out[B])     // in[B]=Use[B]∪(out[B]-Def[B])
      }
```


Example



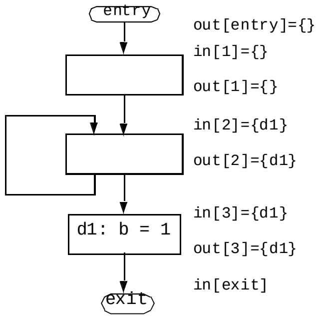
IV. Framework

	Reaching Definitions	Live Variables
Domain	Sets of definitions	Sets of variables
Direction	forward: $out[b] = f_b(in[b])$ $in[b] = \wedge\ out[pred(b)]$	backward: $in[b] = f_b(out[b])$ $out[b] = \wedge\ in[succ(b)]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$	$f_b(x) = Use_b \cup (x - Def_b)$
Meet Operation ($\wedge$)	$\cup$	$\cup$
Boundary Condition	$out[entry] = \emptyset$	$in[exit] = \emptyset$
Initial interior points	$out[b] = \emptyset$	$in[b] = \emptyset$

Thought Problem 1. "Must-Reach" Definitions

- **A definition D (a = b+c) <u>must</u> reach point P iff**
  - D appears at least once along on all paths leading to P
  - a is not redefined along any path after last appearance of D and before P
- **How do we formulate the data flow algorithm for this problem?**

Problem 2: A legal solution to (May) Reaching Def?



- Will the worklist algorithm generate this answer?

Questions

- **Correctness**
  - equations are satisfied, if the program terminates.
- **Precision: how good is the answer?**
  - is the answer ONLY a union of all possible executions?
- **Convergence: will the analysis terminate?**
  - or, will there always be some nodes that change?
- **Speed: how fast is the convergence?**
  - how many times will we visit each node?