Data flow analysis

- Framework for proving facts about program at each point
  - Point: entry or exit from block (or CFG edge)
  - Lots of “small facts”
  - Little or no interaction between facts
- Based on all paths through program
  - Includes infeasible paths

Infeasible Paths Example

```c
a = 1;
if (a == 0) {
  a = 1;
}
if (a == 0) {
  a = 2;
}
```

- Infeasible paths never actually taken by program, regardless of input
- Undecidable to distinguish from feasible

Data Flow-Based Optimizations

- Dead variable elimination
  ```c
  a = 3; print a; x = 12; halt
  a = 3; print a; halt
  ```

- Copy propagation
  ```c
  X = y; … use of x … … use of y
  ```

- Partial redundancy
  ```c
  a = 3*c + d; b = 3*c
  b = 3*c; a = b + d
  ```

- Constant propagation
  ```c
  a = 3; b = 2; c = a + b
  a = 3; b = 2; c = 5
  ```

Data Flow Analysis

- Define lattice to represent facts
- Attach meaning to lattice values
- Associate transfer function to each node
- (f: L → L)
- Initialize values at each program point
- Iterate through program until fixed point

Lattice-Related Definitions

- Meet function: u
  - Commutative and associative
  - x u x = x

- Unique bottom ? and top > element
  - x u ? = ?
  - x u > x

- Ordering: x v y if x u y = x
- Function f is monotone if 8 x, y:
  - x v y implies f(x) v f(y)
**Bit-Vector Lattice**

- Meet = bit-vector logical and

- Meet rules:
  - $111 \lor 111 = 111$
  - $011 \lor 111 = 111$
  - $100 \lor 101 = 101$
  - $011 \lor 011 = 011$
  - $011 \lor 110 = 110$
  - $011 \lor 010 = 010$
  - $100 \lor 000 = 100$
  - $001 \lor 110 = 011$
  - $010 \lor 100 = 100$

**Constant Propagation Lattice**

- Meet rules:
  - $a \lor a = a$
  - $a \lor ? = ?$
  - constant $\lor$ constant = constant (if equal)
  - constant $\lor$ constant = ? (if not equal)

- Define obvious transfer functions for arithmetic

**Iterative Data Flow Analysis**

- Initialize non-entry nodes to $>$
  - Identity element for meet function
  - If node function is monotone:
    - Each re-evaluation of node moves down the lattice, if it moves at all
    - If height of lattice is finite, must terminate

**Constant Propagation Example I**

- Two choices of “point”:
  - Compute at edges
    - maximal information
  - Compute on entry
    - must “meet” data from all incident edges
    - loses information

**Constant Propagation Example II**

- Vector for $(x,a,b,c)$
- Init values to $>$
- Iterate forwards

**Accuracy: MOP vs. MFP**

- We want “meet-over-all-paths” solution, but paths can be infinite if there are loops
- Best we can do in general:
  - **Maximum Fixed Point** solution = largest solution, ordered by $v$, that is fixed point of iterative computation
  - Provides “most information”
  - More conservative than MOP
**Distributive Problems**
- \( f \) is **distributive** iff
  - \( f(x \land y) = f(x) \land f(y) \)
- Doing meet early doesn’t reduce precision
- Non-distributive problems:
  - Constant propagation
- Distributive problems:
  - MFP = MOP
  - Reaching definitions, live variables

**Reaching Definitions**
- **Definition**: each assignment to variable
  - \( \text{defs}(v) \) represents set of all definitions of \( v \)
- Assume all variables scalars
  - No pointers
  - No arrays
- A definition **reaches** given point if there is a path to that point such that variable may have value from definition

**Data Flow Functions**
- \( \text{Kill}(S) \): facts not true after \( S \) just because they were true before
  - Example: redefinition of variable (assignment)
- \( \text{Gen}(S) \): facts true after \( S \) regardless of facts true before \( S \)
  - Example: assigned values not killed in \( S \)
- \( \text{In}(S) \): dataflow info on entry to \( S \)
  - If \( S \) has one predecessor \( P \), \( \text{In}(S) = \text{Out}(P) \)
  - Otherwise:
    - \( \text{In}(S) = \text{Out}(P) \setminus \text{Kill}(P) \)
  - Example: definitions that reach \( S \)
- \( \text{Out}(S) \): dataflow info on exit from \( S \)
  - \( \text{Out}(S) = \text{Gen}(S) \setminus \text{Kill}(S) \)
  - Example: reaching defs after \( S \)

**For Reaching Definitions**
- For reaching defs, \( u = \{ \)
  - \( \text{Gen}(d; v = \text{exp}) = \{ d \} \)
    - “on exit from block \( d \), generate new definition”
  - \( \text{Kill}(d; v = \text{exp}) = \text{defs}(v) \)
    - “on exit from block \( d \), definitions of \( v \) are killed”
  - Computing \( \text{In}(S) \)
    - If \( S \) has one predecessor \( P \), \( \text{In}(S) = \text{Out}(P) \)
    - Otherwise: \( \text{In}(S) = \text{Out}(P) \setminus \text{Kill}(P) \)
    - \( \text{Out} (\text{Entry}) = \{ \} \)

**For Reaching Definitions Example**

```plaintext
parameter a;
parameter b;
x = a*b;
y = a*b;
while (y > a+b) {
a = a+1;
x = a+b;
}
```

```
defs(x) =
defs(y) =
defs(a) =
defs(b) =
```

**Analysis Direction**
- **Forwards analysis**:
  - Start with Entry, compute towards Exit
- **Backwards analysis**:
  - Start with Exit, compute towards Entry
  - \( \text{In}(S) = \text{Out}(S) \setminus (\text{In}(S) \setminus \text{Kill}(S)) \)
  - \( \text{Out}(S) = \text{Out}(S) \setminus \text{Kill}(S) \)
- **Backwards problems**:
  - Live variables: which variables might be read before overwritten or discarded
Next Time

- More data flow analysis