More Dataflow Analyses

Last time: live variable analysis as a dataflow analysis

Dataflow analysis
- Compute facts (live vars) over a control-flow graph (CFG)
  Vertex(CFG) = statements
  Edges(CFG) = program points
- Can build CFG for Joos/IR/assembly

Live variables analysis (LVA): find set of vars live at each program point
(var is live at some program point := its current value may be needed)

in[n] = facts true on all in-edges (conservative estimation)
= vars that may be live before n executes

$$in[n] = use[n] \cup (out[n] \setminus def[n])$$

out[n] = facts true on all out-edges (conservative estimation)
= vars that may be live after n executes

$$out[n] = \bigcup_{n' > n} in[n']$$

iterative solving of equations

$$in[n] = use[n] \cup \left(\bigcup_{n' > n} in[n'] \setminus def[n] \right)$$

Algorithm: iterative solving

1. Initialize in[n] := ∅, for all n
2. Repeat until no change to in[•] is possible:
   For all n:

$$in[n] := use[n] \cup (\bigcup_{n' > n} in[n'] \setminus def[n])$$

Partial correctness: assume alg terminates.

Termination: in[n] can only grow.
in[n] can grow at most V times. (# vertices in CFG.)
in each iteration, at least 1 node's in[n] grows.

main loop executes at most V^2 iterations

Inefficiency: has to perform update even for CFG nodes whose equations are currently satisfied.

Algorithm: worklist

1. Initialize in[n] := ∅, for all n
2. Set worklist (usually FIFO queue) w := all nodes in $Nodes(\text{CFG})$
   Invariant: node n's equations are **not** currently satisfied ⇒ n ∈ w
3. While ∃ $n \in w$:
   $w := w \setminus \{n\}$
   $in[n] := use[n] \cup (\bigcup_{n' > n} in[n'] \setminus def[n])$
   If $in[n]$ changed, push predecessors of n onto w:
   $w := w \cup \{n' \mid n' < n\}$

Partial correctness. only need to show invariant is true by induction.

Termination. in[n] can only grow
main loop runs once every time a node is pushed to w. O(V)
once for every node in the beginning
once for every increase to the node's successor's in[·]

Complexity $O(V^2)$

A4: Detect "dead assignments" and emit warning

Ex/

```
public int f(int z) {
    while (z < 1000) {
        int y = 10;
        if (z > 0) {
            return y;
        } else {
            y = z;
            z = z + 1;
        }
    }
    int y = z;
    return y;
}
```

Want: Joos compiler emits a warning

For all assignment nodes n in CFG. if x = e, x ∉ out[n], emit warning

A4: Reachable Statement Analysis

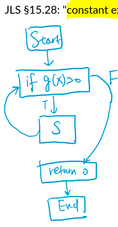
JLS 14.20: must check that all statements can potentially execute

Conservative approximation: overestimate reachability

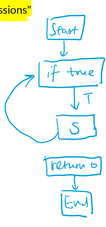
```
f();
x = y + 1;   reachable
return;
if (x > 1) g();   definitely unreachable ⇒ report error
```

CFG:

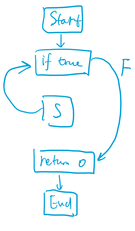
```
int f(int x) {
  while (g(x) > 0) {
    S
  }
  return 0;
}
```




```
int f(int x) {
  while (true) {
    S
  }
  return 0; // unreachable
}
```




```
int f(int x) {
  while (false) {
    S // unreachable
  }
  return 0;
}
```

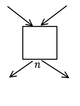


JLS §15.28: "constant expressions"

JLS: constant if-condition handled differently than constant loop condition

Equations for RSA

in[n] = facts true on all in-edges (conservative estimation)
= true/false: "program point before n may be reached"



Compiler reports error if ∃ statement n, in[n] = false

Forward analysis: out[n] computed via in[n]

$$in[n] = \begin{cases} true, & \text{if } n = Start \\ false, & \text{if } n \text{ doesn't have predecessors} \\ \bigvee_{n' < n} out[n'], & \text{otherwise} \end{cases}$$

$$out[n] = \begin{cases} false, & \text{if } n = \text{return;} \text{ or return } e; \\ in[n], & \text{otherwise} \end{cases}$$

JLS: Method return type is not void ⇒ Every finite-length execution path must return explicitly

```
int f(int x) {
  if (x > 0) {
    return f(x - 1);
  }
}
```
no return, report error

```
int f(int x) {
  if (x > 0) {
    f(x - 1);
  } else {
    return x;
  }
}
```
no return; report error

```
int f(int x) {
  while (true) {
    S
  }
}
```
legal

Can be implemented via RSA

Check in[End] = false

If in[End] ≠ false ⇒ Compiler emits error