Yet More Data Flow Analysis

- Last time:
  - The iterative worklist algorithm
- Today:
  - Live variable analysis
  - Constant propagation
  - algorithms
  - def-use chains

Live Variable Analysis

- Variable \( x \) is **live** at point \( p \) if:
  - used before being redefined along some path starting at \( p \)
    - backwards problem
  - \( U \) set(p):
    - variables that may be used starting at \( p \)
  - \( D \) set(p):
    - variables that may be defined in \( p \)

Use, Def, Live Variables:

<table>
<thead>
<tr>
<th>Example</th>
<th>Use</th>
<th>Def</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( x = 12 );</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: ( y = 14 );</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: ( z = x );</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: ( y = 15 );</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: ( q = z + z );</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: ( \text{halt} );</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Defining Live Variable Analysis

- Lattice elements =
  - \( \text{In(Exit)} \) =
  - \( \text{Out(v)} = \text{u}_{\text{In(SUCCESS)}}, \text{In(P)} \) =
  - \( u = \)
    - \( \text{In(v)} = \text{Out(v)} u (\text{Out(v)} - \text{D}(v)) \)
    - \( x \in \text{U} \)
      - \( \text{Out}(v) \text{ is used } \)
      - \( \text{Out}(v) \text{ is defined } \)
- \( x \text{ used in } \text{Out}(v) \) iff \( x \) may be used before defined
  - \( \text{Out}(v) \text{ is defined } \)
- \( x \text{ defined before used in } v \)
Iterative Worklist Algorithm, Live Variables

for v ∈ V
OUT(v) = ∅
IN(v) = Use(v)
worklist = V
while (worklist ≠ ∅)
for v ∈ worklist
oldin(v) = IN(v)
OUT(v) = up2SUCC(v)
IN(v) = Use(v)
(OUT(v) – Def(v))
if (oldin(v) ≠ IN(v))
worklist = PRED(v)

Live Variables Example

Entry
3: x=a*b;
5: if y > a+b
6: a = a+1;

Exit
4: y=a*b;
2: parameter b;
1: parameter a;
7: x = a+b;

Outline

Today:
- Live variable analysis
- Backwards problem
- Constant propagation
  - Algorithms
  - Def-use chains

Constant Propagation

- Discovers constant variables & expressions
- Propagates them as far forward as possible
- Uses:
  - Evaluate expressions at compile-time
  - Eliminates dead code
    - E.g., debugging code
  - Improves effectiveness of many optimizations
- Always a win

Constant Propagation Lattice, Revisited

Meet rules:
- a u > = a
- a u ? = ?
- constant u constant = constant (if equal)
- constant u constant = ? (if not equal)

Initialization:
- Optimistic assumption:
  - All variables unknown constant =>
  - Pessimistic assumption:
    - All variables not constant = ?

Wegman & Zadeck: TOPLAS 1991

- Relates & improves on previous constant propagation algorithms
- Sparsity
  - Improves speed
- Conditional:
  - Incorporates info from branches
Kildall’s Algorithm

- Worklist-based:
  - add successors of Entry
  - remove and examine a node from worklist
  - evaluate expressions to compute new In and Out
  - if the Out value changes,
    - add successors to worklist

Feeds simple constants:
- no information about direction of branches
- one value per variable along each path

Kildall’s Algorithm: Example

Reif & Lewis

- Kildall’s (SC):
  - at each node, computes value of all variables at entry and produces set of values for all variables at exit
- Reif & Lewis (SSC):
  - also finds simple constants, but faster
  - spare representation
  - original formulation based on def-use graph
  - revised version based on SSA form

Def-Use Graph

- Graph of def-use chains:
  - connection from definition site (assignment) to use site along path in CFG
  - does not pass through another definition
  - includes infeasible paths
  - misses some constants

Def-Use Graph: Example
Reif and Lewis

- **Worklist:**
  - Put root edges from def-use graph in worklist
  - if def site in roots can be evaluated to constant, assign that to variable, otherwise?
  - assign all other variables >
  - remove def-join edges from worklist:
    - propagate value of def to use using meet rules
    - if value is lowered, add node to worklist

Wegman & Zadeck: Conditional Definition

- **Conditional definition:**
  - keeps track of conditional branches
  - form of dead code elimination
  - constant expr in branch
  - mark appropriate branch as executable
  - use symbolic execution to mark edges
  - ignore non-executable edges at joins when propagating constants

Def-Use Chains: Problem

```c
switch (j)
    case x: i A 1;
    case y: i A 2;
    case z: i A 3;
switch (k)
    case x: a A i;
    case y: b A i;
    case z: c A i;
```

- worst-case size of graph = \(O(?)\)

Next Time

- “SSA is a better way”
- Dominance & dominance frontiers
- Control dependence

- Read ACDI Chapter 8, pp. 252—258