Advanced Compilers

CMPSCI 710

Spring 2003

Yet more data flow analysis

Emery Berger

University of Massachusetts, Amherst

Project Stuff

- 1 to 2 person teams
- Implement optimization/analysis in:
  - Jikes RVM (IBM's research Java compiler)
  - Broadway (UTexas "metacompiler")
  - other (subject to approval)
- Due dates:

02/11/03:	One-page project description.
02/25/03:	2-4 page project design.
03/25/03:	Project implementation review.
04/29/03:	Implementation due.
05/06-13/03:	In-class presentations.
05/13/03:	Project report.

Yet More Data Flow Analysis

- Last time:
  - The iterative worklist algorithm
- Today:
  - Live variable analysis
    - backwards problem
  - 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
- *Use(p)*:
  - variables that may be *used* starting at *p*
- *Def(p)*:
  - variables that may be *defined* in *p*

Use, Def, Live Variables: Example

	Use	Def	Live
`1: x = 12;`			
`2: y = 14;`			
`3: z = x;`			
`4: y = 15;`			
`5: q = z + z;`			
`6: halt;`			

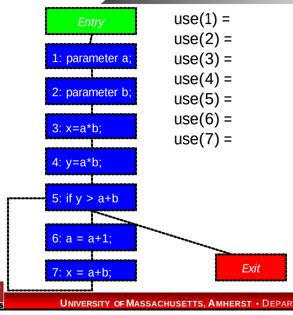
Defining Live Variable Analysis

- Lattice elements =
- *In(Exit)* =
- *Out(v)* = u$_{P \text{ in } \mathrm{SUCC(S)}}$*In(P)*
- u =
- *In(v)* = *Use(v)* u (*Out(v)* – *Def(v)*)
- x 2 *Use(v)* iff x may be used before defined
  - *Use(d: v = …x…) =*
  - *Use(d: if (…x…)) =*
- x 2 *Def(v)* iff x defined before used in v
  - *Def(d: v = exp) =*

Iterative Worklist Algorithm, Live Variables

```
for v 2 V
  OUT(v) = ∅
  IN(v) = Use(v)
worklist Ã V
while (worklist ≠ ∅)
  for v 2 worklist
    oldin(v) = IN(v)
    OUT(v) = up 2 succ(v) IN(p)
    IN(v) = Use(v) u (OUT(v) – Def(v))
    if (oldin(v) ≠ IN(v))
      worklist Ã worklist [ PRED(v)
```

Live Variables Example



use(1) =	def(1) =
use(2) =	def(2) =
use(3) =	def(3) =
use(4) =	def(4) =
use(5) =	def(5) =
use(6) =	def(6) =
use(7) =	def(7) =

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

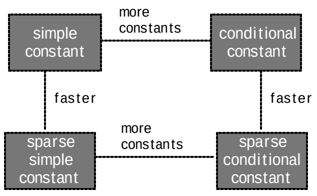
Constant Propagation Lattice, Revisited

```
        >
… -2 -1 0 1 2 …
        ?
```

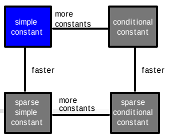
- Meet rules:
  - a u > = a
  - a u ? = ?
  - constant u constant = constant *(if equal)*
  - constant u constant = ? *(if not equal)*
- Initialization:
  - Optimistic assumption:
    - all variables unknown constant = >
  - Pessimistic assumption:
    - all variables not constant = ?

Wegman & Zadeck: TOPLAS 1991

- Relates & improves on previous constant propagation algorithms
- Sparsity
  - improves speed
- Conditional:
  - incorporates info from branches

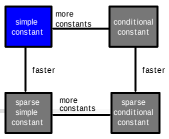


Kildall's Algorithm

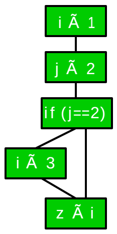


- Worklist-based:
  - add successors of Entry
  - remove and examine a node from worklist
  - evaluate expressions to compute new In and Out
  - if the Out value changes,
    - add successors to worklist
- Finds **simple constants:**
  - no information about direction of branches
  - one value per variable along each path

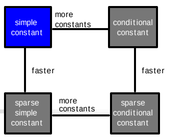
Kildall's Algorithm: Example



```
i Ã 1
j Ã 2
if (j == 2)
  i Ã 3
z Ã i
```

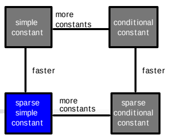


Kildall's Algorithm: Analysis



- In terms of N, E, V:
  - N = assignments + expressions in branches
    - for convenience: N = nodes in CFG
  - E = edges in CFG
  - V = variables
- Iterations = 2 * V * I (in-edges)
  - Runtime = iterations * operations
    =
  - Space = lattice values
    =

Reif & Lewis



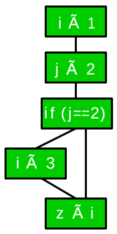
- Kildall's (SC):
  - at each node, computes value of all variables at entry and produces set of values for all variables at exit
- Reif & Lewis (SSC):
  - also finds simple constants, but faster
  - *sparse* representation
  - original formulation based on def-use graph
    - revised version based on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

Def-Use Graph

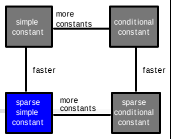
- Graph of **def-use chains:** connection from **definition site** (assignment) to **use site** along path in CFG
  - does not pass through another definition
- Includes infeasible paths
  - misses some constants

Def-Use Graph: Example

```
i Ã 1
j Ã 2
if (j == 2)
  i Ã 3
z Ã i
```



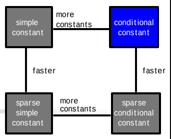
Reif and Lewis



- 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

Reif and Lewis: Example

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

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