Lecture notes by Edward Loper

Course: CIS 570 (Modern Programming Language Implementation)
Professor: E Christopher Lewis
Institution: University of Pennsylvania

http://www.cis.upenn.edu/~eclewis/cis570

Monday, January 15, 2001

Assignments:

- midterm
- final
- small assignments
- reading summaries (1-page)
- group project

Group project... can i do anything with compilers/linguistics? ...

Wednesday, January 17, 2001

Notes:

- Contact E. re wpe?
- Add self to mailing list?

Notes:

- dunkumware.com ?
- st david sq? lancaster?

1 Data-flow analysis

1.1 Control Flow Graph (CFG)

diagram where nodes are commands and edges are possible transitions between commands.

1.2 Liveness

A variable is "live" on an edge i→ j of a control flow graph iff the program's behavior at j or future(j) depends on the value of the variable at node i.

If two variables are live at mutually exclusive edges, then they can share a register.

Liveness is a property that flows "backwards" through CFGs: find usages of v, and follow CFG backwards (on all paths) until you find a definition of v. All edges traversed are live.

1.3 Back to CFGs

CFGs have a number of properties. These properties "flow" through the CFG in specific ways. E.g., liveness flows backwards through the CFG. Other properties flow in different ways.

Terms

- out-edge: an edge leading out of a node
- in-edge: an edge leading into a node
- successor node: a node connected by an out-edge
- predecessor node: a node connected by an in-edge
- pred[n] = set of predecessor nodes of n
- succ[n] = set of successor nodes of n
- def: a node that defines a variable
- use: a node that reads a variable
- defs of a variable = set of nodes
- defs of a node = set of variables

1.4 Back to liveness

Fomal definition of liveness:

v is live on edge e iff:

1. ∃ p ∈ paths from e to a USE of v
2. p does not go through any DEF of v

Defs for liveness:

- a variable is live-in at a node if it is live on any of the node's in-edges.
- a variable is live-out at a node if it is live on any of the node's out-edges.

Finding liveness:

1. if v $\in$ use[n], then v $\in$ live-in[n]
2. if n1$\rightarrow$ n2, v $\in$ live-in[n2], then v $\in$ live-out[n1]
3. if v $\in$ live-out[n] and v $\notin$ def[n], then v $\in$ live-in[n]

So: in[n] = use[n] $\cup$ (out[n]-def[n]) out[n] = $\cup${s $\in$ succ[n]} in[s]

Finding it

iterate the in and out equations until we get convergance. to make it faster:

- make sure we compute things in the right order.. namely, backwards.
- only consider basic blocks

Wednesday, February 7, 2001

send email to listsrv about project?

2 SSA

SSA can be used to make UD and DU chains more sparse. SSA is an "alternate" program representation.

memo to myself – can ssa be used in bytecode? last minute optimizations.. whee.

2.1 Alternate program representations

- can allow analyses and transformations to be simpler & more efficient/effective.
- may not be "executable"
- may make inefficient use of space

2.2 Static Single Assignment Form (SSA)

Idea: each assignment is to a uniquely named variable Property: each use has exactly 1 reaching def Effects: makes UD chains sparse

Transformation:

- rename each def
- rename uses reached by that def
- trivial for streight-line code
- for joins, we need a new operation ϕ. (special case: loops)

2.3 SSA vs UD/DU chains

Advantages of SSA:

- more compact
- easier to update/manipulate
- each use hase only 1 def
- value merging is explicit
- eliminates "false" dependencies

3 Transforming to SSA

- Insert ϕ-functions
- Rename variables

3.1 Insert ϕ-functions

basic rule: If x→ z and y→ z converge at z, and x and y contain defs for v, then ϕ for v is inserted at z.

placing approaches:

- minimal = as few as possible, subject to basic rule
- briggs-minimal = minimal, but v must be live across some basic block. Intuition – if a variable v is assigned and used in the same basic block, and never used again, then don't bother with it.

- pruned = same, except dead ϕ's not listed. this will have a subset of the ϕ functions of briggs-minimal.

Machinery

domination:

- d dom i if all paths from entry$\rightarrow$ i include d
- (d sdom i) iff (d dom i) and (d $\neq$ i)

dominance frontier:

- dominance frontier of d is the set of nodes that are "just barely" not dominated by d.
- dominance frontier of a set of nodes is the union of the dominance frontiers of the nodes.
- to find dominance frontier: find set of nodes dominated by d, then go "one beyond" (including loop up to d but not loops to anything else that d dominates).
- dominance frontier nodes are nodes where value from d merges with some other value.

Using dominance frontiers

1. find dominance frontier of the defs of a var. Then add everything from the dominance frontier to the set of defs, and find the dominance frontier of that.. repeat.

Theorem: iterated dominance frontier = set of nodes that require ϕ-functions for v.

3.2 Rename Variables

4 Using SSA

4.1 Constant Propagation

- Simple: Constant for all paths through a program
- Simple sparse
- Conditional: Constant for all actual paths through a program
- Conditional sparse

Thursday, April 5, 2001

5 Interprocedural Analysis

DLLs with summaries?