CS 412/413

Introduction to
Compilers and Translators

Andrew Myers
Cornell University

Lecture 28: Loop optimizations

7 April 00

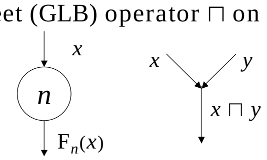
Administration

- HW4 due Monday
- Prelim 2 next Thursday
- MMII graphical user interface package released for PA5

Last time

Dataflow analysis framework:

1. Lattice of dataflow information values L with order $\sqsubseteq$, top $\top$
2. Monotonic flow functions $F_n : L \rightarrow L$
3. Meet (GLB) operator $\sqcap$ on L



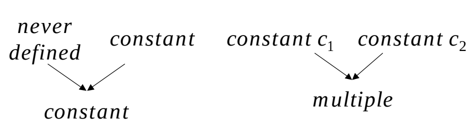
Constant propagation

- Idea: propagate and fold integer constants in one pass

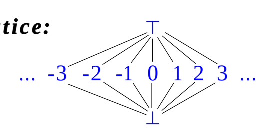
```
x = 1;          x = 1;
y = 5+x;   ⇒    y = 6;
z = y*y;        z = 36;
```

- Information about a single variable:
    i. Variable never defined
    ii. Variable has single constant value
    iii. Variable has multiple values

One-variable Const. Prop.



Full lattice:

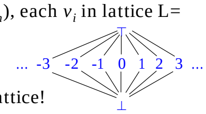


Rest of defn.

- Flow function for x = x OP c_1:
  $F_n(\top) = \top$
  $F_n(\bot) = \bot$
  $F_n(c_2) = c_2 \text{ OP } c_1$
- Flow function is monotonic: iterative solution works
- What about multiple variables $x_1 \ldots x_n$? Want tuple $(v_1, \ldots v_n)$,

Multiple vars

- Dataflow value is a tuple $(v_1,\ldots v_n)$, each v_i in lattice L=



- Set of tuples $(v_1,\ldots,v_n)$ is also a lattice!

  $(v_1,\ldots,v_n) \sqsubseteq (v'_1,\ldots,v'_n) \Leftrightarrow \forall_i\, v_i \sqsubseteq v'_I$

  $(v_1,\ldots,v_n) \sqcap (v'_1,\ldots,v'_n) = (v_1 \sqcap v'_1,\ldots,v_n \sqcap v_n)$
- For any two lattices L_1, L_2, have *product lattice* $L_1 \times L_2$ with component-wise ordering

  $(v_1, v_2) \sqsubseteq (v'_1, v'_2) \Leftrightarrow v_1 \sqsubseteq v'_1 \;\&\; v_2 \sqsubseteq v'_2$
  - Is this really a lattice?
- Dataflow values are in $L \times \ldots \times L = L^n$

Flow functions

- Consider $x_1 = x_2$ OP x_3

  $F(x_1, \top, x_3) = (\top, \top, x_3)$

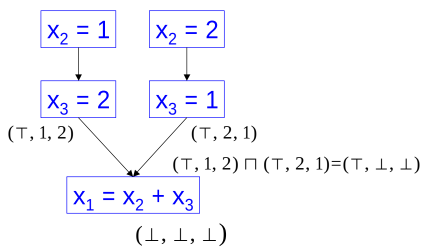
  $F(x_1, x_2, \top) = (\top, x_2, \top)$

  $F(x_1, \bot, x_3) = (\bot, \bot, x_3)$

  $F(x_1, x_2, \bot) = (\bot, x_2, \bot)$

  $F(x_1, c_2, c_3) = (c_2 \text{ OP } c_3, c_2, c_3)$
- Monotonic? Distributes over $\sqcap$?

Not MOP!



$F((\top, 1, 2) \sqcap (\top, 2, 1)) \neq F(\top, 1, 2) \sqcap F(\top, 2, 1)$

Loops

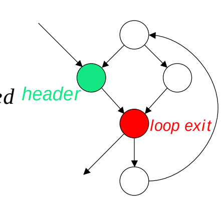
- Most execution time in most programs is spent in loops: 90/10 is typical
- Most important targets of optimization: loops
- Loop optimizations:
  - loop-invariant code motion
  - loop unrolling
  - loop peeling
  - strength reduction of expressions containing induction variables
  - removal of bounds checks
- When to apply loop optimizations?

High-level optimization?

- Loops may be hard to recognize in IR or quadruple form -- should we apply loop optimizations to source code or high-level IR?
  - Many kinds of loops: while, do/while, continue
  - loop optimizations benefit from other IR-level optimizations and vice-versa -- want to be able to interleave
- Problem: identifying loops in call-flow graph

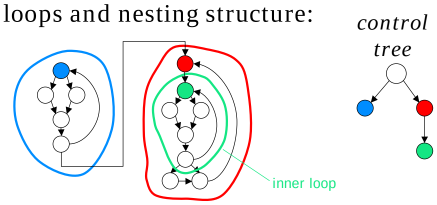
Definition of a loop

- A *loop* is a set of nodes in the control flow graph, with one distinguished node called the *header* (entry point)
- Every node is reachable from header, header reachable from every node: *strongly-connected component*
- No entering edges from outside except to header
- nodes with outgoing edges: *loop exit nodes*



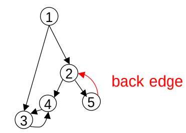
Nested loops

- Control-flow graph may contain many loops, and loops may contain each other
- *Control-flow analysis* : identify the loops and nesting structure:



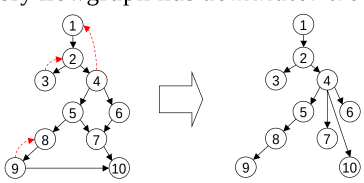
Dominators

- CFA based on idea of *dominators*
- Node A *dominates* node B if the only way to reach B from start node is through A
- Edge in flowgraph is a *back edge* if destination dominates source
- A loop contains at least one back edge



Dominator tree

- Domination is transitive; if A dominates B and B dominates C, then A dominates C. ***A*** *immediately dominates* ***B*** if domination not implied transitively
- Every flowgraph has *dominator tree*



Finding dominators

- Goal: for every node in flowgraph, find its set of dominators
- Properties of dominators:
  1. Every node dominates itself
  2. A node **B** is dominated by another node **A** if **A** dominates all of the predecessors of **B**

Dominator data-flow analysis

- Forward analysis; out[n] is set of nodes dominating n
- "A node **B** is dominated by another node **A** if **A** dominates *all* of the predecessors of **B**"

$$in[n] = \bigcap_{n' \in pred[n]} out[n']$$

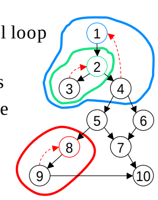
- Every node dominates itself:

$$out[n] = in[n] \cup \{n\}$$

- Formally: L = sets of nodes ordered by $\subseteq$, flow functions $F_n(x) = x \cup \{n\}$, $\top = \{\text{all } n\}$
  ⇒ Standard iterative analysis converges on MOP soln

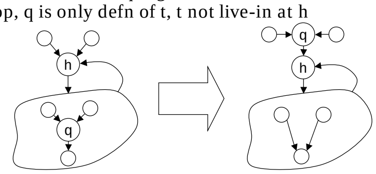
Completing control-flow analysis

- Dominator analysis gives all back edges
- Each back edge n→h has an associated *natural loop* with h as its header: all nodes reachable from h that reach n without going through h
- For each back edge, find its natural loop
- Nest loops based on subset relationship between natural loops
- Exception: natural loops may share same header; merge them into larger loop.
- Build control tree using nesting relationship



Loop-invariant hoisting

- *Idea:* move computations that always give the same result out of the loop: only compute once!
- Hoisting quadruple q: t = a + b. Use *reaching definitions* analysis to see if a, b are invariant (conservatively)
- Must also ensure q is guaranteed to be executed by loop, q is only defn of t, t not live-in at h



Induction variables

- *Induction variables* are variables with value $ai + b$ on the i^{th} iteration of a natural loop, for constants a & b
- Various optimizations can exploit information about induction variables:
  - strength reduction
  - array bounds check elimination
  - loop unrolling

Identifying induction variables

- ***Basic induction variables:*** only one definition of the form i = i + K
- ***Derived induction variables:*** one definition of the form j = i * M + N

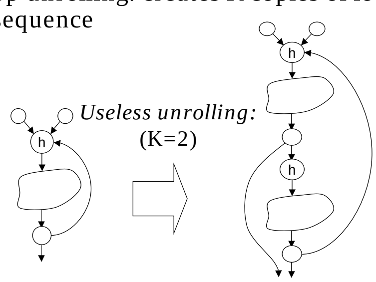
```
j = 3;
for (i = 0; i < n; i++) {
  j = j +1;
  k = i*4 + 8;
  m = k*12 + 1;
  ...
}
```


Strength reduction

- Every derived induction variable k can be written as $a*i + b$, a and b constants, i some basic induction variable
- For all distinct (a,b) pairs:
  - insert before loop header $k' = b$
  - insert after loop header $k' = k' + a$
  - Replace definition of any k whose formula is $a*i + b$ with $k = k'$
- Result: multiplication(s) replaced by single addition
- Additional optimizations facilitated: copy/constant propagation, dead/useless variable elimination, dead code elimination

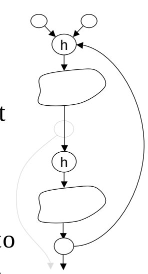
Loop unrolling

- Loop unrolling: creates K copies of loop in sequence



Using induction variables

- When loop test expression depends on induction variable (e.g. i < n), can use one loop test to ensure that entire unrolled loop will succeed (i+K-1 < n): remove all interior loop tests
- Additional loop is needed to "finish up" 0..K-1 iterations



Useful unrolling

Summary

- Constant propagation: not all lattice elements are sets; not all analyses give MOP solution.
- Optimizing loop code is critical to good performance
- Loops can be identified automatically in control-flow graph using dominator data-flow analysis; allows interleaving of loop optimizations
- Induction variables enable many loop optimizations: loop unrolling, strength reduction, array bounds checks.