More Dataflow Analysis

March 5, 2001

CS 132: Compiler Design

Review

- Dataflow analyses are static (compile-time) attempts to describe dynamic (run-time) values and computations.
  - Necessarily conservative approximations
  - Depends on (conservative) control-flow graph
- General organization
  - Describe desired property via equations for each program point.
  - Solve the equations.

Review: Liveness Analysis

- A variable is said to be *live* at a program point if its contents affects the result (visible behavior) of the rest of the program.
  - Conservative approximation: variable is live if its value might be accessed in the future, according to the conservative control-flow graph.
- If a variable is not live, we can trash its value with impunity.

Review: Liveness Analysis

- For each instruction s:
  - Define $def(s)$ to be variables potentially modified by s.
  - Define $use(s)$ to be the variables accessed by s.

$$livein(s) = use(s) \cup (liveout(s) \setminus def(s))$$

$$liveout(s) = \bigcup_{p \in succ(s)} livein(p)$$

$$liveout(terminal\ node) = \emptyset$$

Review: Liveness Analysis

		DEF	USE	LIVE
				x
1.	w ← x+1	w	x	
				w
2.	w ← w+1	w	w	
				w
3.	q ← 4	q	∅	
				w
4.	x ← w+2	x	w	
				x,w
5.	w ← x+w	w	x,w	
				w
6.	return w	∅	w	
				∅

Review: Liveness Analysis

- Consequences:
  - Line 3 is *dead code* and can be deleted.
  - Could rename `x` to `z` on lines 4 and 5 only.
  - If we did so, could rename `w` to `x` everywhere.

		LIVE
		x
1.	w ← x+1	
		w
2.	w ← w+1	
		w
3.	q ← 4	
		w
4.	x ← w+2	
		x,w
5.	w ← x+w	
		w
6.	return w	
		∅

Digression: Mathematics of Dataflow Analysis

- Definition:
  - A *complete lattice* is a partial order $(L,\sqsubseteq)$ where every subset $S \subseteq L$ has a least upper bound $\bigsqcup S$ and greatest lower bound $\bigsqcap S$.
    - This includes the empty set, so L must have a minimum $\perp$ and a maximum.
  - A fixed point of a function $F : L \rightarrow L$ is a value x such that $F(x)=x$.
  - The *least fixed point* of $F : L \rightarrow L$, denoted $lfp\ F$, is a fixed point *satisfying* $(lfp\ F) \sqsubseteq x$ for every fixed point x of F.
  - We define the *greatest fixed point* ($gfp\ F$) similarly.

Digression: Mathematics of Dataflow Analysis

- Propositions
  - Every monotone function on a complete lattice has least and greatest fixed-points.
    - Recall: F is monotone iff $\forall x,y \in L.\ (x \sqsubseteq y) \Rightarrow (F(x) \sqsubseteq F(y))$
  - If F is monotone and the sequence

    $$\perp \sqsubseteq F(\perp) \sqsubseteq F(F(\perp)) \sqsubseteq F^3(\perp) \sqsubseteq F^4(\perp) \sqsubseteq \ldots$$

    attains a maximum, then this maximum is $lfp(F)$
    - Proof: Every element of this sequence must be $\sqsubseteq lfp(F)$, and a maximum of this sequence is a fixed point.
  - Similarly, if $\forall x,y \in L.\ (x \sqsubseteq y) \Rightarrow (F(x) \sqsupseteq F(y))$ and

    $$\top \sqsupseteq F(\top) \sqsupseteq F(F(\top)) \sqsupseteq F^3(\top) \sqsupseteq F^4(\top) \sqsupseteq \ldots$$

    attains a minimum, then this minimum is $gfp(F)$.

Digression: Mathematics of Dataflow Analysis

- Tuple of sets for $(in(1),\ldots,in(6),out(1),\ldots out(6))$ form a complete lattice.
  - Component-wise union, intersection, subset
  - Minimum element is contains 12 empty sets.
- The function which performs one iteration of the liveness analysis is then a monotone function on 12-tuples of sets.
  - Every solution to the equations is a fixed point of this function.
- Thus if iteration starting with all sets empty terminates, we have the least fixed point.
  - But we already know that the iterative process has to terminate because the possible sets are bounded.
  - Least fixed point corresponds to solution with the smallest sets.

Available Expressions

- Definition:
  - An expression is *available* at a program point if, on every path from the beginning to this point,
    1. The expression is computed at least once
    2. There is no assignment to this expression's variables since the expression was last computed.
- Goal
  - Find all available expressions
    - i.e., at each point find the expressions which are guaranteed to have been computed already.
- Answer
  - Set of expressions in the program for each point

Available Expressions

- For each instruction s, define $gen(s)$ to be the set of *expressions* whose result is accessible after this instruction.
  - Singleton set if s is a definition that does not overwrite its free variables
  - Empty set otherwise
- For each instruction s, define $kill(s)$ to be the set of all expressions whose value may be changed by this statement
  - All expressions involving t if the temporary t is modified.
  - All expressions involving memory if s is a store
  - Empty set otherwise (if s only concerns control flow)

Available Expressions

- For each instruction s, define $in(s)$ to be the set of expressions whose values are guaranteed to be stored somewhere before s executes.
- For each instruction s, define $out(s)$ to be the set of expressions whose values are guaranteed to be stored somewhere after s executes.

$$in(1) = \varnothing$$
$$in(s) = \bigcap_{p \in pred(s)} out(p)$$
$$out(s) = gen(s) \cup (in(s) \setminus kill(s))$$

Solution by Iteration

- Note that this analysis computes intersections rather than unions.
  - Each iteration can make the sets smaller.
- So, we start out by initializing all the sets to be the set of *all* expressions
  - Except *in*(1) which must stay empty
- Computes the *greatest* fixed point
  - Finds as many as possible expressions that are guaranteed available.

Application

- Common Subexpression Elimination
  - If `e` is available at the statement `t ← e` we know the value of e has already been computed and is still around --- somewhere.
  - Create a new temporary `t'`.
  - Search backwards to all the places where it might have been computed (to statements of the form `w ← e`) and insert the copy `t' ← w` afterwards.
  - Then replace the original statement by `t ← t'`
  - Relies on copy propagation to eliminate redundant copies.
  - Warning: can't do this if `e` has side-effects.

Implementing Iteration

- Example set representations
  - Sorted list
    - Union takes time proportional to the size of sets *involved*
  - Binary string
    - One bit for each possible set member
    - Intersection and union by logical and, or
      - Takes time proportional to the *maximum possible* set size

Implementing Iteration

- Make nodes be basic blocks
  - Compute *gen* and *kill* or *def* and *use* for each block

    $$use(s1; s2) = use(s1) \cup (use(s2) \setminus def(s1))$$

    $$def(s1; s2) = def(s1) \cup def(s2)$$
  - Use the same iterative equations to compute *in* and *out* for each block
  - Use these to (in one pass) compute *in* and *out* for each statement, as desired.
  - Advantage: fewer nodes in control-flow graph
    - Hence, less work per iteration

Implementing Iteration

- Order the nodes
  - Number of iterations (but not final answer) depends on order in which we visit the nodes.
- Idea:
  - (Assume forward dataflow analysis, so *in* computed from the *out* of the predecessors)
  - If there were no loops (i.e., if the graph were a DAG) then we could topologically sort the graph
    - Process every node before looking at its successors
    - One iteration would suffice.
  - Choose order that approximates topological sort
    - Reverse postorder graph traversal (DFS)
  - Flip order for backward analysis
  - Out of luck for bidirectional analysis

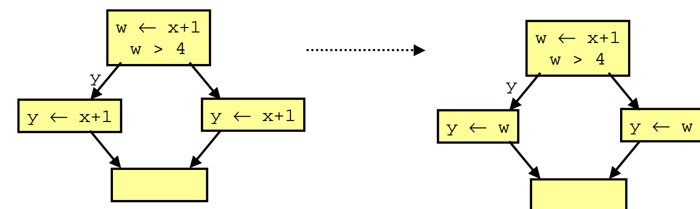
Implementing Iteration

- Observation:
  - May happen that most of the sets have converged, but some small part of the program is more complicated and isn't done.
  - General iteration algorithm will still recompute sets for the entire program.
- Worklist algorithm
  - Just keep track of all the nodes that might change
  - e.g., if a node is recalculated and its *out* set changes, then add all its successors to worklist. (Assuming forward analysis.)
  - Only recalculate for nodes on the worklist.
    - Optimization: always pick the worklist node that's earliest in the DFS order

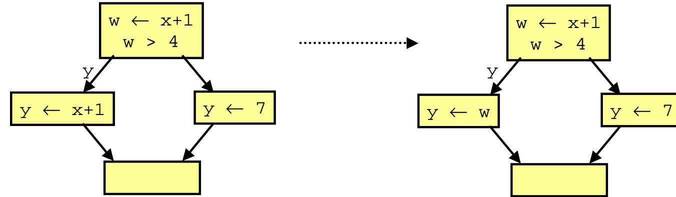
Partial Redundancy Elimination

- A partial redundancy is a computation that is performed more than once on *some* path through the control flow graph.
- *Partial redundancy elimination* is an optimization that moves, copies, and deletes computations so as to minimize partial redundancies
  - Guaranteed not to add more computations to any path.

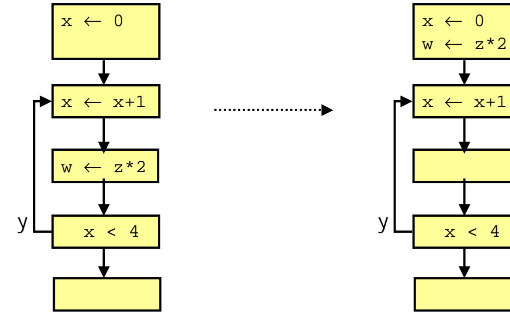
Example 1



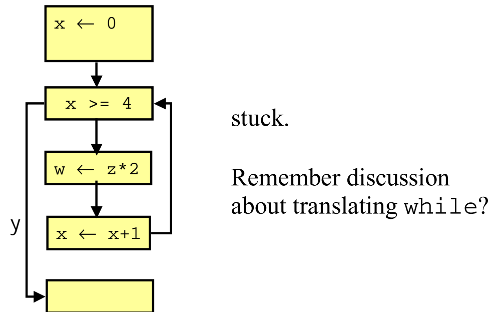
Example 2



Example 3



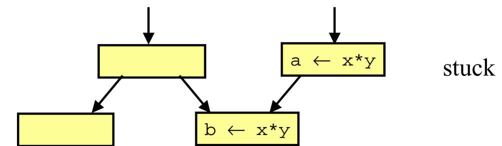
Example 4



stuck.

Remember discussion about translating `while`?

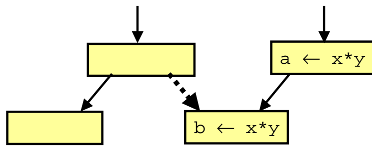
Example 5



stuck

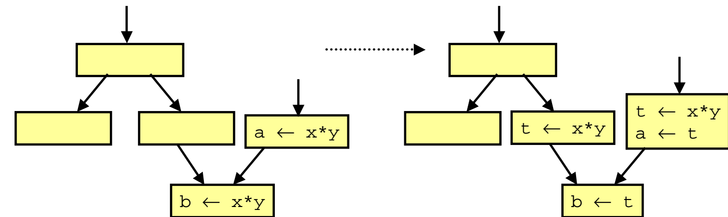
Definition

- An edge in the flow graph is said to be a *critical edge* if it goes from a node with multiple successors to a node with multiple predecessors

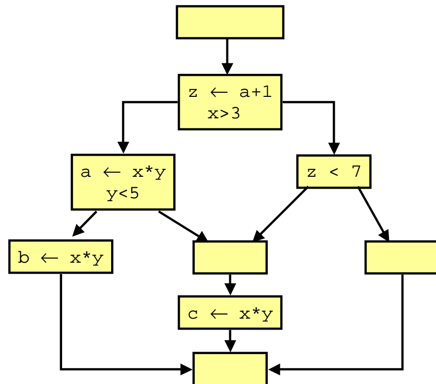


Removing Critical Edges

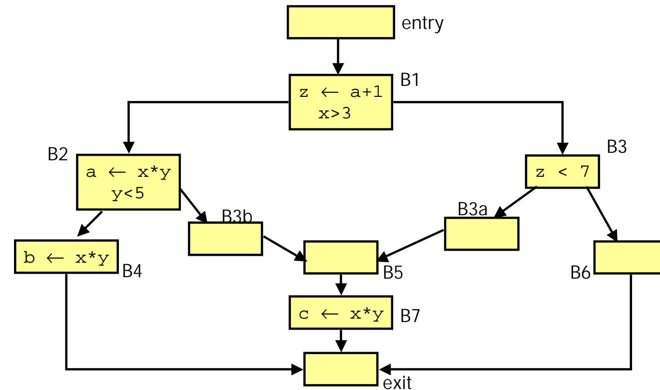
- We can "split" all critical edges by inserting empty basic blocks.
  - Improves results of PRE.



Big Example



Big Example: No Critical Edges



Step 1

- An expression is said to be *locally transparent* in a basic block if the block does not alter the value of that expression.
  - Conservative approximation: check that block contains no assignments to variables appearing in the expression
  - Notation: $TransLoc(b)$ denotes the set of locally transparent expressions for the basic block b.

$TransLoc(\text{B2}) = \{\texttt{x*y}\}$
$TransLoc(b) = \{\texttt{x*y, a+1}\}$ for all other blocks b

Step 2

- An expression is *locally anticipatable* in a block iff
  - it is computed in the block
  - and, its computation could be moved to the beginning of the block without change.

$AntLoc(\text{B1}) = \{\texttt{a+1}\}$
$AntLoc(\text{B2}) = \{\texttt{x*y}\}$
$AntLoc(\text{B4}) = \{\texttt{x*y}\}$
$AntLoc(\text{B7}) = \{\texttt{x*y}\}$
$AntLoc(b) = \varnothing$ for all other blocks b

Step 3

- An expression is *globally anticipatable* at a program point iff
  - every path leading from the point eventually computes the expression, and
  - the expression yields the same value along the entire path

$AntOut(\text{exit}) = \varnothing$
$AntOut(b) = \bigcap_{p \in succ(b)} AntIn(p)$
$AntIn(b) = AntLoc(b) \cup (TransLoc(b) \cap AntOut(b))$

$AntIn(\text{entry,B1}) = \{\texttt{a+1}\}$
$AntIn(\text{B2,B2a,B3a,B4,B5,B7}) = \{\texttt{x*y}\}$
$AntIn(\text{B6,exit}) = \varnothing$

Step 4

- An expression is said to be *earliest* at a block boundary if there is a path from the entry to here containing no block where the expression is globally anticipatable and yields the same value as it would here.

$EarlIn(\text{entry}) = \text{All expressions}$
$EarlIn(b) = \bigcup_{p \in pred(b)} EarlOut(p)$
$EarlOut(b) = \overline{TransLoc(b)} \cup (\text{EarlIn}(b) \setminus AntIn(b))$

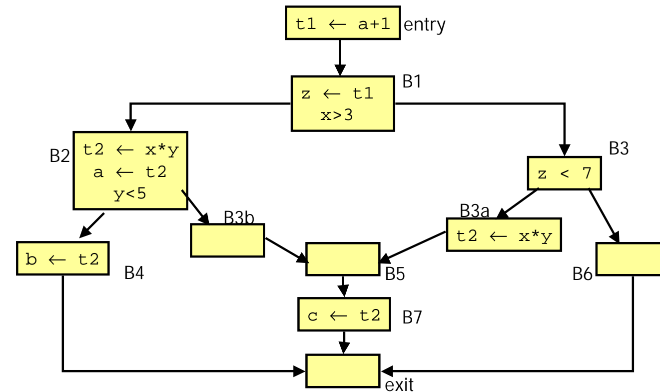
$EarlIn(\text{entry,exit}) = \{\texttt{a+1,x*y}\}$
$EarlIn(\text{B1,B2,B3,B3a,B6}) = \{\texttt{x*y}\}$
$EarlIn(\text{B2a,B4,B5,B7}) = \{\texttt{a+1}\}$

PRE Transformation

- For each expression e, allocate a fresh variable `t`.
- At every block b where $e \in AntIn(b) \cap EarlIn(b)$, insert the assignment `t` $\leftarrow$ e
- Replace every original occurrence of e by t.

- The result is "computationally optimal"
  - No more computations on any path than in the original program.
  - Any other re-arrangement would require more computations on some path.
  - NB: Does not necessarily remove *all* partial redundancies!

Result



Refinements

- Lazy Code Motion
  - Refinement of PRE
  - Moves computations as late as possible.
  - Can be computed with just 3 more (unidirectional) dataflow problems.
- Combine with other optimizations/transformations
  - Strength reduction (in a weak form)
  - Reassociation (`a ← b+i; c ← a-i; d ← b+i`)
- Some improvements reported by using SSA form
- Use profiling information to guide movement
- Restructure/duplicate code if useful

Example 2