Global Optimization
Basic tool: iterative algorithms over graphs

- Dead code elimination
- Reduction in strength
- Constant folding
- Loop-invariant code motion
- Common subexpression elimination
- Can be performed
definition/information, various optimizing transformations
and uses of variables beyond basic blocks. With use-
to generate better code, need to examine definitions

Data Flow Analyses
The Flow Graph

Nodes are basic blocks

Edges are transitions (conditional/unconditional jumps)

For every node B (basic block) we define the sets

Pred(B) and Succ(B) which describe the graph

Global information reaching B is computed from the sets

Pred(B) or Succ(B) (forward propagation)

Informa\?

Within a basic block we can easily (in a single pass)

compute local information, typically a set

of variables that are assigned a value:
def(B)

of variables that are operands:
use(B)

Global information on all nodes is comprised from the following:

• The Flow Graph

• Nodes are basic blocks

• Edges are transitions (conditional/unconditional jumps)

• For every node B (basic block) we define the sets

• Pred(B) and Succ(B) which describe the graph

• Variables that are assigned a value:
def(B)

• Variables that are operands:
use(B)

• Global information on all nodes is comprised from the following:

• The Flow Graph

• Nodes are basic blocks

• Edges are transitions (conditional/unconditional jumps)

• For every node B (basic block) we define the sets

• Pred(B) and Succ(B) which describe the graph

• Variables that are assigned a value:
def(B)

• Variables that are operands:
use(B)
Example: Live Variables Analysis

**Definition:** A variable is alive if its current value is used subsequently in the computation. If it is not used in any successor of block \( B \), it is not live on exit from block \( B \).

\[
\text{Livein}(B) = \text{Liveout}(B) \cap \text{use}(B) \text{ for } B \text{ any block}
\]

\[
\text{Liveout}(B) = \text{Succ}(B) \cup \text{Livein}(B) \text{ for } B \text{ any block}
\]

**Use:** If a variable is not alive on exit from a block, it does not need to be saved (stored in memory).

**Definition:** A variable is alive at location if it is current value is used subsequently in the computation. If it is not used on exit from block \( B \), nothing is alive.

\[
\text{Liveout}(B \text{ exit}) = \emptyset
\]
Liveness Conditions

\[ z - 2 \]

\[ x \]

\[ \exists x \cdot \]

\[ \cdots =: \ z \]

\[ \text{Liveness } h', h \]

\[ 1 + h \cdot \cdots \]
Example: Reaching Definitions

Definition:

The set of computations (quadruples) that may be used at a location.

Use: compute use-definition relations.

Definition: The set of computations (quadruples) that may be used at a location.

Nothing reaches the entry of the program.

\( \emptyset = (\text{entry}) \cap \text{In} \)

A computation reaches the exit if it reaches the entry and is not recomputed in the block, or if it is computed locally.

\( \text{Out} (B) = \text{In} (B) \setminus (\text{gen} (B) \cap (B) \cup \text{kill} (B)) \)

In the block, or if it is computed locally.

A computation reaches the entry of a block if it reaches the exit of a predecessor.

\( (\text{Out} (B)) \cap \text{In} (B) \cap \text{entry} \subset (B) \cup \text{predecessor} \)

\( (d) \cup \text{Out} (B) \cap (B) \cap \text{predecessor} \subset (B) \cup \text{entry} \)

Lecture 10: Global Optimization

A. Pnueli

Honors Compilers, NYU, Spring, 2007
Iterative Solution

Note that the equations are monotonic: if $Out(B)$ increases, $In(B')$ increases for some successor.

General approach: start from lower bound (e.g. empty sets), iterate until nothing changes.

Initially $in(b) = \emptyset$ for all $b$, $out(b) = gen(b) - kill_a(b)$

while change loop

forall $b \in \text{blocks}$ loop

change := false;

forall $p \in \text{pred}(b)$

in(b) := \bigcup_{p \in \text{pred}(b)} out(p);

oldout := out(b);

out(b) := in(b) \cup gen(b) - kill_a(b);

if oldout \neq out(b) then change := true; endif;

endloop;

endloop;

end loop;
Better algorithms use node orderings.

\[\text{while not empty(queue)}\]
\[\text{loop} \]
\[\text{dequeue}(b); \]
\[\text{recompute} \text{out}(b); \]
\[\text{if} \text{out}(b) \text{has changed then enqueue all of } b\text{'s successors;}\]
\[\text{enqueue all of } b\text{'s successors;}\]
\[\text{end loop;}\]

\[\text{Empty: Instead of recomputing all block, keep a queue of}\]
\[\text{nodes that may have changed. Iterate until queue is}\]
\[\text{empty.}\]
Example: Available Expressions

Definition: Computation (triple, e.g. x+y) that may be available at a point because previously computed.

Local Information:
Use: common subexpressions elimination

Example: Available Expressions

\[(q)\text{\texttt{kill}} \cup (q)\text{\texttt{gen}} \cap (q)\text{\texttt{exp}} \subseteq (q)\text{\texttt{in}} \cap (q)\text{\texttt{out}}\]

\[\text{in } \bigcup_{p \in \text{pred}(b)} (q)\text{\texttt{pred}} \subseteq (q)\text{\texttt{in}} \cap (q)\text{\texttt{exp}} \subseteq (q)\text{\texttt{exp}} \text{\texttt{kill}} \subseteq (q)\text{\texttt{exp}} \text{\texttt{gen}} \subseteq (q)\text{\texttt{in}} \cap (q)\text{\texttt{out}}\]

All predecessors

Computation is available on entry if it is available on exit from

\[\text{exp}_{\text{gen}}(b) \subseteq \text{set of expressions computed in } b\]

\[\text{exp}_{\text{kill}}(b) \subseteq \text{set of expressions whose operands are modified in } b\]

\[\text{Out}(b) \subseteq \text{in}(b) \cup (q)\text{\texttt{gen}} \cap (q)\text{\texttt{exp}} \text{\texttt{kill}} \subseteq (q)\text{\texttt{in}} \cap (q)\text{\texttt{out}}\]

Honors Compilers, NYU, Spring, 2007
Iterative Solution

Approach as above: Start from lower bound (e.g. empty sets).

Iterate until nothing changes.

Note that the equations are monotonic: if \(\text{Out}(B)\) increases,
\(\text{Out}(B)\) does not decrease for any successor.

Initially in \(\emptyset = (q)\) for all.

end loop;

end loop;

\(\text{if oldout} \neq \text{out} \text{ then change} := \text{true} \end{\text{if}}\)

\((q)\)\text{exp}\exp_{\text{kill}}(\text{in}) =: (\text{out})\\text{exp}_{\text{gen}}(\text{out}) \cap (\text{in})\\text{exp}_{\text{gen}}(\text{out}) =: (\text{out})\cup (\text{in})\\text{exp}_{\text{pre}}d\exists d
d\text{forall } b \in \text{blocks Loop}
\text{change} := \false;\)
\text{while change Loop}
\text{change} := \true;
\text{forall } b \in \text{blocks Loop}
\text{oldout} := \text{out}(b);
\text{out}(b) := \text{in}(b) \\text{exp}_{\text{gen}} \\text{exp}_{\text{kill}}(\text{in})
\text{if } \text{oldout} \neq \text{out}(b) \text{ then change} := \text{true} \end{\text{if}};
\text{endif};
\text{endloop};
\text{endloop};
Use-definition Chaining

- The value computed by $b$:
  - $(b)_{np}$: set of occurrences that may use the value of $b$.

- Inverse map:
  - $o_{pn}$: set of quadruples that may have computed the value of $o$.

- $o$ of value that may have generated the value:
  - $(o)_{pn}$: set of quadruples that may have generated the value of $o$. (operand in a quadruple) to the quadruple occurrence (expression).

- The closure of available expressions: map each expression to the set of quadruples that may use it.
Finding Loops in Flow-Graph
A computation $A$ is loop invariant if $A$ is the only assignment to the target variable in the loop, and $Q$ dominates all exits from the loop, and there is no use of the target variable that has another definition within the loop.

An exception may now be raised before the loop if:

- There is no use of the target variable within the loop, and
- $Q$ dominates all exits from the loop, and $Q$ is the only assignment to the target variable in the loop, and
- A quadruple $Q$ that is loop invariant can be moved to the pre-header of the loop if $Q$ dominates all exits from the loop, and
- $Q$ is the only assignment to the target variable in the loop, and
- There is at most one computation of $x$ and $y$ within the loop, and
- There is no use of the target variable that has another definition within the loop.

$x$ and $y$ are constant, or $A$ computation $(x \circ y)$ is loop invariant within a loop if $A$ is loop invariant.
Important for loops over multidimensional arrays.

Can be removed.

Generalization to polynomials in $j$: all multiplications

If $j$ increments by $p$, $k$ increments by $p \cdot c_0$.

Addition.

Can compute $k := k + c_0$, replacing multiplication by constants. $k$ is a basic induction variable.

Where $j$ is the loop variable, $k = 0$, $1$, $\ldots$, $c_0$ and $c_1$ are arithmetic series: $k = j \cdot c_0 + c_1$.

An induction variable in a loop takes values that form

$\ldots$
Remove original assignment to $k$

Insert after incrementing:

$k := k + c_0 \cdot incr$

Insert in loop pre-header:

$k := c_0 \cdot incr + c_1$

Note that $c_0 \cdot incr$ is a static constant.

$(k, c_0 \cdot incr, c_0 \cdot incr + c_1)$

Any variable that has a single assignment of the form

$k := c_0 \cdot incr + c_1$

is an induction variable. With triple

$(k, c_0 \cdot incr, j_0)$

For every induction variable, establish a triple

$(var, incr, init)$

A. Pnueli

Lecture 10: Global Optimization

Honors Compilers, NYU, Spring, 2007
Global Constant Propagation

Domain is set of values (not bit-vector).

Status of a variable is \{c, non-const, unknown\}

Like common sub-expression elimination, but instead of intersection, define a merge operation:

- \text{Merge}(c, \text{unknown}) = c
- \text{Merge}(\text{non-const, anything}) = \text{non-const}
- \text{Merge}(c_1, c_2) = \text{if } c_1 = c_2 \text{ then } c_1 \text{ else non-const}

Initially all variables are unknown, except for explicit constant assignments.

Initially all variables are unknown, except for explicit constant assignments.