To generate better code, need to examine definitions and uses of variables beyond basic blocks. With use-definition information, various optimizing transformations can be performed.

**Basic tools: iterative algorithms over graphs**

- Dead code elimination
- Reduction in strength
- Constant folding
- Loop-invariant code motion
- Common subexpression elimination

**Flow Graph**

The nodes are basic blocks, edges are transfers (conditional/unconditional jumps), and we describe the graph for every node $B$ (basic block) we define the sets

**Example: Live Variables Analysis**

- **Liveout** $(B)$ = $\cap \{S \mid \text{Succ}(B) \}
- \{S \}

- **Livein** $(B)$ = $\{S \in \text{Pred}(B) \}$

We exclude variables which are defined before they are used.

A variable is live on entry if it is live on exit or used within $B$.

A variable is live on exit if it is live in any successor of $B$.

We assume that a variable is used subsequently in the computation value is used subsequently in the computation.

A variable is live on exit if it is live in any successor of $B$.

A variable is live on entry if it is live on exit from a block, it is defined or used in a basic block.

Variables that are assigned a value:

- $\text{def}(B)$
- $\text{use}(B)$

Variables that are operands:

- $\text{use}(B)$
- $\text{def}(B)$

Global information reaching $B$ is computed from the use-definition information on all $\text{Pred}(B)$ (forward propagation) or $\text{Succ}(B)$ (backward propagation) on the flow graph.

Global information in $B$ is computed from the information on all $\text{Pred}(B)$ (forward propagation) or $\text{Succ}(B)$ (backward propagation) on the flow graph.

To generate better code, need to examine definitions and uses of variables beyond basic blocks. With use-definition information, various optimizing transformations can be performed.
Lecture 10: Global Optimization

A. Pnueli

Liveness Conditions

\[ z := ::: x + 1 \]

\[ x; y \text{ live} \]

\[ z \text{ live} \]

Example: Reaching Definitions

**Denition:**

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

**Use:**

Compute use-definition relations.

**Definition:**

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

Better algorithms use node orderings. Instead of recomputing all block, keep a queue of nodes that may have changed. Iterate until queue is empty.

**Work-pile Algorithm**

- Nothing reaches the entry of the program.
  \[ \emptyset = \text{in}(B) \]
- Nothing reaches the exit of the program.
  \[ \text{out}(B) = \text{out}(B) \]
- A computation reaches the exit if it reaches the entry and is not recomputed.
  \[ \text{in}(B) \cap \text{out}(B) = \text{out}(B) \in \text{m} \]
- A computation reaches the entry of a block if it reaches the exit of a predecessor.
  \[ \text{in}(B) \cap \text{out}(B) = \text{out}(B) \in \text{m} \]
- Use: compute use-definition relations.

Iterative Solution

\[ \text{end loop: } \]

\[ \text{if } \text{old} \text{out}(B) \neq \text{new} \text{out}(B) \text{ then change: end if; } \]

\[ \text{while not empty(queue) loop } \]

\[ \text{dequeue}(B) ; \]

\[ \text{recompute out}(B) ; \]

\[ \text{if out}(B) \text{ has changed then enqueue all of B's successors; } \]

\[ \text{end loop; } \]

Better algorithms use node orderings. Instead of recomputing all block, keep a queue of nodes that may have changed. Iterate until queue is empty.

**Work-pile Algorithm**

- Nothing reaches the entry of the program.
  \[ \emptyset = \text{in}(B) \]
- Nothing reaches the exit of the program.
  \[ \text{out}(B) = \text{out}(B) \]
- A computation reaches the exit if it reaches the entry and is not recomputed.
  \[ \text{in}(B) \cap \text{out}(B) = \text{out}(B) \in \text{m} \]
- A computation reaches the entry of a block if it reaches the exit of a predecessor.
  \[ \text{in}(B) \cap \text{out}(B) = \text{out}(B) \in \text{m} \]
- Use: compute use-definition relations.

**Definition:**

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

Example: Reaching Definitions

\[ z - 2 \]

\[ 1 + \bar{x} \]

Liveness Conditions

Restrictions: Start from lower bound (e.g. empty sets).

General approach: start from lower bound (e.g. empty sets).

Iterative approach: start from lower bound (e.g. empty sets).

Note that the equations are monotonic: if \( \text{out}(B) \) increases, then \( \text{out}(B) \) increases for all \( B \) of some successor.

Initial: \( \text{in}(B) = \text{out}(B) = \emptyset \).

While loop: change := true;

forall \( B \) in blocks loop:

\( \text{in}(B) := \text{sp}_{\text{pred}}(B) \text{ out}(p) \);

oldout := \( \text{out}(B) \);

\( \text{out}(B) := \text{in}(B) \mid \text{gen}(B) \text{ kill } a(B) \);

if oldout \# \( \text{out}(B) \) then change := true;
endif;
endloop;
endloop;
Example: Available Expressions

**Definition:**
A computation (triple $e.g. x+y$) that may be available at a point because previously computed.

**Use:**
- Common subexpressions elimination
- Available expressions

**Finding Loops in Flow-Graph**
- A dominator of a node $n$ dominates all the predecessors of $n$.
- For all $B$, the initial condition should be $\{n \rightarrow \} = (g) \forall n \in B$
- The value computed by the entry point of the program.

**Use-definition Chaining**
- The closure of available expressions.

**Finding Loops in Flow-Graph**
- A dominator of a node $n$ dominates all the predecessors of $n$.
- The initial conditions should be $\{n \rightarrow \} = (g) \forall n \in B$
- Use-definition Chaining
- The closure of available expressions.

**Example: Available Expressions**

**Definition:**
A computation (triple $e.g. x+y$) that may be available at a point because previously computed.

**Use:**
- Common subexpressions elimination
- Available expressions

**Finding Loops in Flow-Graph**
- A dominator of a node $n$ dominates all the predecessors of $n$.
- The initial conditions should be $\{n \rightarrow \} = (g) \forall n \in B$
- Use-definition Chaining
- The closure of available expressions.
**Loop Optimization**

- **Strength Reduction**
  - Specialized loop optimization: formaldifferentiation
  - Aninductionvariableinalooptakesvaluesthatformanarithmeticseries:
    
\[
    k = j_0 + j_1 \]

  Where \( j_0 \) is the loop variable \( j_0 = 0, 1, 2, \ldots \), \( c_0 \) and \( c_1 \) are constants.
  - An induction variable in a loop takes values that form an arithmetic series:
    \( k = 0 + c_0 + \ldots + c_1 \).
  - An inductionvariable is invariant:
  - Specalized loop optimization: formaldifferentiation
  - Important for loops over multidimensional arrays.

**Global Constant Propagation**

- Domain is set of values (not bit-vector).
- Initially all variables are unknown, except for explicit constants.
- Initially all variables are unknown, except for explicit constants.
  - Merge of \( c \) andunknown \( c \) is non-const.
  - Merge ofunknown \( c \) andnon-const \( c \) is non-const.
  - Merge ofanything andunknown \( c \) is non-const.

**Induction Variables**

- For every induction variable, establish a triple \((\text{var}; \text{incr}; \text{init})\).
  - Insert after incrementing \( j \): 
    
\[
    k := k + c_0 \text{ incr } j + c_0 \text{ init } c_1 \]

**Global Optimization**

- Strength Reduction
  - Specialized loop optimization: formaldifferentiation
  - Aninductionvariableinalooptakesvaluesthatformanarithmeticseries:
    
\[
    k = j_0 + j_1 \]

  Where \( j_0 \) is the loop variable \( j_0 = 0, 1, 2, \ldots \), \( c_0 \) and \( c_1 \) are constants.
  - An induction variable in a loop takes values that form an arithmetic series:
    \( k = 0 + c_0 + \ldots + c_1 \).
  - An inductionvariable is invariant:
  - Specalized loop optimization: formaldifferentiation

An exception may now be raised before the loop.

- There is no use of the target variable that has another definition.
- \( Q \) is the only assignment to the target variable in the loop, and \( Q \) dominates all exits from the loop, and
- The pre-header of the loop iff \( Q \) is an induction variable.

A quadruple \( Q \) that is loop invariant can be moved to

- These conditions are invariant:
  - There is an arithmetic loop invariant
  - If and \( y \) are constant, or
  - A computation with respect to any loop

**Loop Optimization**

- Remove original assignment to \( k \).
- Insert after incrementing \( j \):
  
\[
    k := k + c_0 \text{ incr } j + c_0 \text{ init } c_1 \]

**Global Optimization**

- Strength Reduction
  - Specialized loop optimization: formaldifferentiation
  - Aninductionvariableinalooptakesvaluesthatformanarithmeticseries:
    
\[
    k = j_0 + j_1 \]

  Where \( j_0 \) is the loop variable \( j_0 = 0, 1, 2, \ldots \), \( c_0 \) and \( c_1 \) are constants.
  - An induction variable in a loop takes values that form an arithmetic series:
    \( k = 0 + c_0 + \ldots + c_1 \).
  - An inductionvariable is invariant:
  - Specalized loop optimization: formaldifferentiation

An exception may now be raised before the loop.

- There is no use of the target variable that has another definition.
- \( Q \) is the only assignment to the target variable in the loop, and \( Q \) dominates all exits from the loop, and
- The pre-header of the loop iff \( Q \) is an induction variable.

A quadruple \( Q \) that is loop invariant can be moved to

- These conditions are invariant:
  - There is an arithmetic loop invariant
  - If and \( y \) are constant, or
  - A computation with respect to any loop