Local optimizations

Consider the expression: $a + a \times (b - c) + (b - c) \times d$

**Tree**

```
+   *
  +   *
    +   -
      a   d
    *   -
      a   b
  *   *
    a   c
  -   -
    b   c
```

**Directed acyclic graph**

```
+   *
  +   *
    +   -
      a   d
    *   -
      a   b
  *   *
    a   c
  -   -
    b   c
```

---

**Local optimizations**

**Common subexpressions (CSE)**

- portion of expressions
- repeated multiple times
- computes same value
- can reuse previously computed value

**Directed acyclic graph (DAG)**

- program representation
- nodes can have multiple parents
- no cycles allowed
- exposes common subexpressions

**Building a DAG for an expression**

- maintain hash table for leafs, expressions
- unique name for each node — its *value number*
- reuse nodes found in hash table
Directed acyclic graphs

What about assignment?

- complicates detection of common subexpressions
- identical expression $\rightarrow$ different value
- must ensure each value has a unique node

One solution - renaming

- add subscripts to variable names (e.g., $x \rightarrow x_i$)
- increment subscript of name if target (LHS) of assignment
- variables references use new subscript

Example

$$a_1 = a_0 + b_0$$

Can apply to entire basic block

Directed acyclic graph example

<table>
<thead>
<tr>
<th>Code</th>
<th>After Renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b + c$</td>
<td>$a_0 = b_0 + c_0$</td>
</tr>
<tr>
<td>$b = a - d$</td>
<td>$b_1 = a_0 - d_0$</td>
</tr>
<tr>
<td>$c = b + c$</td>
<td>$c_1 = b_1 + c_0$</td>
</tr>
<tr>
<td>$d = a - d$</td>
<td>$d_1 = a_0 - d_0$</td>
</tr>
</tbody>
</table>
Common subexpressions

Going beyond basic blocks

- can no longer build DAGs
- must consider control flow

Examples

- possible kill

  \[
  \begin{align*}
  c &= a + b \\
  \text{if (\ldots)} & \quad \begin{align*}
  a &= \ldots \\
  d &= a + b
  \end{align*}
  \end{align*}
  \]

- possible gen

  \[
  \begin{align*}
  \text{if (\ldots)} & \quad \begin{align*}
  c &= a + b \\
  d &= a + b
  \end{align*}
  \end{align*}
  \]

We handle these conditions using data-flow analysis

Data-flow analysis

Data-flow analysis

- \textit{compile-time} reasoning about the \textit{run-time} flow of values in the program
- represent facts about run-time behavior
- represent effect of executing each basic block
- propagate facts around control flow graph

Formulated as a set of simultaneous equations

- sets attached to the nodes and edges
- lattice to describe relation between values
- usually represented as bit or bit vector

Limitations

- answers must be conservative
- often need to approximate information
- assume all possible paths can be taken
Data-flow analysis

Algorithm

1. build control flow graph (CFG)
2. initial (local) data gathering
3. propagate information around the graph
4. post-processing (if needed)

Example control flow graph

$$\begin{align*}
a &= 1 \\
\text{if (b) then} & \quad c := a+b \\
\text{else} & \quad b = 1 \\
& \quad c := a+b
\end{align*}$$

Available expressions

Definition

- An expression is defined at point $p$ if its value is computed at $p$.
- An expression is killed at a point $p$ if one of its argument variables is defined at $p$.
- An expression $e$ is available at a point $p$ in a procedure if every path leading to $p$ contains a prior definition of $e$ that is not killed between its definition and $p$.

Global common subexpression elimination

- If, at some definition point for $p = e$, $e$ is available with name $x$, we can replace the evaluation with a reference to $x$.
- requires a global naming scheme
- natural analog to parts of value numbering
Available expressions

For a block $b$

- let $\text{GEN}(b)$ be the set of expressions defined in $b$ and not subsequently killed in $b$.
- let $\text{KILL}(b)$ be the set of expressions killed in $b$.
- let $\text{IN}(b)$ be the set of expressions available on entry to $b$.
- let $\text{OUT}(b)$ be the set of expressions available on exit to $b$.

$\text{IN}$ and $\text{OUT}$ represent global information and can be calculated as:

$$\text{OUT}(b) = \text{GEN}(b) \cup (\text{IN}(b) - \text{KILL}(b))$$

$$\text{IN}(b) = \bigcap_{x \in \text{pred}(b)} (\text{OUT}(x))$$

$\text{AVAIL}$ is simply $\text{IN}$. Its calculation can be combined as:

$$\text{AVAIL}(b) = \bigcap_{x \in \text{pred}(b)} (\text{GEN}(x) \cup (\text{AVAIL}(x) - \text{KILL}(x)))$$

Available expressions example

AVAIL(A) = \emptyset
AVAIL(B) = \text{GEN}(A) \cup (\text{AVAIL}(A) - \text{KILL}(A)) = \emptyset \cup (\emptyset - \{a+b\}) = \emptyset
AVAIL(C) = \text{GEN}(A) \cup (\text{AVAIL}(A) - \text{KILL}(A)) = \emptyset \cup (\emptyset - \{a+b\}) = \emptyset
AVAIL(D) = (\text{GEN}(B) \cup (\text{AVAIL}(B) - \text{KILL}(B))) \cap (\text{GEN}(C) \cup (\text{AVAIL}(C) - \text{KILL}(C)))
= (\{a+b\} \cup (\emptyset - \emptyset)) \cap (\{a+b\} \cup (\emptyset - \{a+b\})) = \{a+b\}
Solving data-flow equations

Iterative algorithm

change = true;
while (change)
    change = false;
    for each basic block      // faster in reverse PostOrder:
        solve data-flow equations for b
        if (old ≠ new)
            change = true;
    end for
end while

Speed of solution

- node may change only if some predecessor changes
- try to visit node after all its predecessors
- reverse PostOrder propagates information quickly
- programs usually converge after 3–4 passes
- use bitvectors for more efficiency

PostOrder and reverse PostOrder

Step1: PostOrder

main()
    count = 1;
    Visit (root);

Visit(n)
    mark n as visited
    for each successor s of n not yet visited
        Visit(s);
    PostOrder(n) = count;
    count = count + 1;

Step 2: Reverse PostOrder (rPostorder)

for each node n
    rPostOrder(n) = NumNodes - PostOrder(n)

Depth-first search ≈ rPostOrder
Reaching definitions

- **The problem:** What are the assignments (or definitions) of a variable $x$ that may reach a particular reference to $x$?

- **Why is this useful?**

Constant propagation:

\[
\begin{align*}
  &a = 1 \\
  &a = 2 \quad a = 2 \\
  &\quad b = 3 \\
  &\quad = a \\
  &\quad = b
\end{align*}
\]

Loop invariant code motion:

\[
\begin{align*}
  &L: \\
  &\quad a = a + 4 \\
  &\quad b = 20 \\
  &\quad c = b + a \\
  &\quad \text{if (\ldots) goto } L
\end{align*}
\]

Reaching definitions

- A **definition** of a variable $x$ is a statement that assigns, or may assign, a value to $x$.

- A definition $d$ **reaches** a program point $p$ if there exists a path from the point immediately following $d$ to $p$ such that $d$ is not killed along that path.

- $\text{REACH}(b)$ is the set of definitions reaching the entry of basic block $b$

- $\text{DEF}(b)$ is the set of *local definitions* in $b$ that reach the end of $b$

- $\text{KILL}(b)$ is the set of variables killed by $b$

- **Equations:**

\[
\text{REACH}(b) = \bigcup_{x \in \text{pred}(b)} (\text{DEF}(x) \cup (\text{REACH}(x) - \text{KILL}(x)))
\]

Best case for $\text{REACH}(b) = \emptyset$

Worse case for $\text{REACH}(b) = \{ \text{all definitions} \}$
Live variables

Definition:

• A definition \( d \) is live at program point \( p \) if the variable \( v \) defined by \( d \) may be used along some path in the program starting at \( p \) without being redefined between \( d \) and \( p \).

• Otherwise, the definition is dead

Why is this useful?

• global analysis to locate dead assignments.

\[
\begin{align*}
    a &= b \\
    b &= a \\
    a &= b \\
    b &= a \\
    &= b
\end{align*}
\]

Live variables

• Slightly different, since information at basic block is based on what happens later in the program.

• A backward data-flow problem.

• \( \text{LIVE}(b) \) is the set of definitions live on exit from block \( b \).

• \( \text{KILL}(b) \) is as before.

• \( \text{USE}(b) \) is the set of locally exposed uses

• \( \text{succ}(b) \) is the set of basic blocks that are immediate successors of \( b \) in the control flow graph.

• Equations:

\[
\text{LIVE}(b) = \bigcup_{x \in \text{succ}(b)} (\text{USE}(x) \cup (\text{LIVE}(x) - \text{KILL}(x)))
\]

Best case for \( \text{LIVE}(b) = \emptyset \)
Worse case for \( \text{LIVE}(b) = \{ \text{all definitions} \} \)
What do these have in common?

\[
\text{AVAIL}(b) = \bigcap_{x \in \text{pred}(b)} (\text{GEN}(x) \cup (\text{AVAIL}(x) - \text{KILL}(x)))
\]

\[
\text{REACH}(b) = \bigcup_{x \in \text{pred}(b)} (\text{DEF}(x) \cup (\text{REACH}(x) - \text{KILL}(x)))
\]

\[
\text{LIVE}(b) = \bigcup_{x \in \text{succ}(b)} (\text{USE}(x) \cup (\text{LIVE}(x) - \text{KILL}(x)))
\]

- **Confluence Operator or Meet Function**: union or intersection
- **Behavior for block**: GEN and KILL
- **A direction**: forward (confluence over predecessors) or backward (over successors)
- **Best case set value**: \( \top \)
- **Worst case set value**: \( \bot \)

**General equations**:

\[
\text{IN}(b) = \land_{p \in \text{pred}(b)} \text{OUT}(p)^\dagger
\]

\[
\text{OUT}(b) = \text{GEN}(b) \cup (\text{IN}(b) - \text{KILL}(b))
\]

\( \dagger \) Reverse graph for backward problem.

---

**Data-flow analysis frameworks**

**Use same framework for all data-flow problems**

- given local information GEN, KILL
- start with some initial values for sets IN, OUT
- iterate through nodes in the flow graph, recompute transfer functions until sets stabilize

**Framework has three components**

- Domain of values: \( L \)
- Operator for combining values: \( \land \)
- A set of transfer functions \( (L \to L) : \mathcal{F} \)

**Usefulness of unified framework**

- Defines a collection of properties that guarantee correctness, convergence;
- Can describe speed of convergence and precision of result for a family of analysis problems
- Can re-use code to solve new analysis problems
Data-flow lattices

Definitions

1. A lattice is a set $L$ and a meet operation $\land$ such that, $\forall a, b, c \in L$
   
   (a) $a \land a = a$  \hspace{1cm} [idempotent]
   
   (b) $a \land b = b \land a$  \hspace{1cm} [commutative]
   
   (c) $a \land (b \land c) = (a \land b) \land c$  \hspace{1cm} [associative]

2. $\land$ imposes a partial order on $L$, $\forall a, b \in L$
   
   (a) $a \geq b \iff a \land b = b$
   
   (b) $a > b \iff a \geq b$ and $a \neq b$

3. A lattice may have a bottom element $\bot$
   
   (a) $\forall a \in L, \bot \land a = \bot$
   
   (b) $\forall a \in L, a \geq \bot$

4. A lattice may have a top element $\top$
   
   (a) $\forall a \in L, \top \land a = a$
   
   (b) $\forall a \in L, \top \geq a$

Available expressions example:

let $D = \{ x \mid x \subseteq \{e_1, e_2, e_3\}\}, \land = \cap$

\[
\top = \]

\[
\bot = \]

Partial ordering $\{e_1, e_2\} \text{ vs. } \{e_3\}$

Single lattice vs. one for each variable

\[
\top = \]

\[
\bot = \]
Data-flow lattices

How does this relate to data-flow analysis?

- choose a semi-lattice $L$ to represent facts
- attach to each element of $L$ a *meaning*
  each $a \in L$ is a distinct set of known facts
- with each node $n$, associate a *transfer function*
  $f_n : L \rightarrow L$ to model behavior of $n$
- propagate facts around the graph

**Example — AVAIL**

- semi-lattice is $2^E$, where $E$ is the set of all expressions computed
  $\wedge$ is $\cap$, $\perp$ is $\emptyset$, $\top$ is $E$
- for a node $n$, $f_n$ has the form $f_n(x) = D_n \cup (x - N_n)$
  where $D_n = \text{GEN}_n$ and $N_n = \text{KILL}_n$
- the underlying graph is the flow graph $G = (N, E, n_0)$
  $n_0$ is the entry node

Iterative algorithm

What about loops?

- circular dependencies between blocks
- can initialize solutions, then solve repeatedly

**Example**

```plaintext
    c = a+b
```

L:

```plaintext
d = a+b
a = ...
if (...) goto L
```

**Termination**

- goal is for solutions to converge to a *fixed point*
- can stop once solutions stop changing
- is this guaranteed?
Monotonicity

- A framework \((D, \land, F)\) is monotone iff
  \[ x \leq y \implies f(x) \leq f(y) \]
  i.e., a “smaller or equal” input to the same function will always give
  a “smaller or equal” output

- Equivalently, monotone iff
  \[ f(x \land y) \leq f(x) \land f(y) \]
  i.e., if merge input, then apply \(f\), result is “smaller or equal” to
  applying \(f\) individually and merging result

- Intuitively, monotonicity means “smaller” input will not yield “larger”
  output.

- monotone frameworks are guaranteed to converge and terminate (if lattice
  elements can only drop in information a finite number of times)

Quality of solution

Possible solutions

- perfect solution = meet over \textit{real} paths taken during program execution
- meet-over-all-paths (MOP) = meet over \textit{potential} paths in control flow
  graph
- maximal-fixed-point (MFP) = solution from iterative framework

Properties

- in general, \(\text{MFP} \leq \text{MOP} \leq \text{Perfect Solution}\)
- in some sense, MOP is best feasible solution
- MFP is unique, regardless of order of propagation
- a framework is \textit{distributive} if \(f(x \land y) = f(x) \land f(y)\)
- for a distributive framework, \(\text{MFP} = \text{MOP}\)