Value numbering

Algorithm

1. Given operation \( \text{op} \ x \ x \ k \), compute values for \( x \).

2. Hash value from \( x \) into hash table.

3. Consistent hash key from \( \text{op} \) and value numbers.

4. If hash key exists in table, replace \( \text{op} \) with copy.

5. Otherwise, create value number \( v \), assign \( \text{op} \) to \( v \) and \( x \).

Invalid

Valid

Table: Value numbering example

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Yes</td>
<td>1. ( t )</td>
</tr>
<tr>
<td>6</td>
<td>Yes</td>
<td>2. ( t )</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>3. ( t )</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>4. ( t )</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>5. ( t )</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>6. ( t )</td>
</tr>
</tbody>
</table>

Two variables have the same value only if they
are provably identical

Distinct value within block

Associate unique value number with each

Table: Symbōls

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>5</td>
<td>( t )</td>
</tr>
<tr>
<td>const</td>
<td>9</td>
<td>( t )</td>
</tr>
<tr>
<td>const</td>
<td>15</td>
<td>( t )</td>
</tr>
</tbody>
</table>

Two variables have the same value only if they
are provably identical

Distinct value within block

Associate unique value number with each

Table: Constants

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>5</td>
<td>( t )</td>
</tr>
<tr>
<td>b</td>
<td>9</td>
<td>( t )</td>
</tr>
<tr>
<td>a</td>
<td>15</td>
<td>( t )</td>
</tr>
</tbody>
</table>
Data-flow analysis

Data-flow analysis

- *compile-time* reasoning about the *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

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

\[
\begin{align*}
a := 1 \\
\text{if} (b) \text{ then} \\
\quad c := a + b \\
\text{else} \\
\quad b := 1 \\
\quad c := a + b \\
\ldots
\end{align*}
\]
Available expressions

Note: Initializations must be conservative.

\[
(((x)_{\text{INIT}} \Rightarrow (x)_{\text{INIT}}) \cup \text{GEN}(x))_{\text{INIT}} \Rightarrow (q)_{\text{INIT}}
\]

Now, \(\text{AVAIL}\) can be defined as:

Program

Knowledge of all potential expressions in the
expressions from DAG construction.
\(\text{GEN}(q)\) can be calculated from the list of live
expression with reference to \(x\).
We can replace the
available with \(x\).
If at some definition point for \(p\) \(\rightarrow e\) is
Global common subexpression elimination

Note:

definition and \(p\).

for \(x\), \(x\) is not killed between its
prior definition, \(q\), and \(x\) is available at \(p\).

An expression \(e\) is available at \(p\) if one of its
computed at \(p\).
An expression \(e\) is defined at \(p\) if its value is

Available expressions
Available expressions example

A:

\[
\begin{align*}
  a &:= 1 \\
  \text{if (b)}
\end{align*}
\]

B:

\[
\begin{align*}
  c &:= a + b \\
  b &:= 1 \\
  c &:= a + b
\end{align*}
\]

C:

D:

\[
\begin{align*}
  \text{AVAII}(A) &= \\
  \text{AVAII}(B) &= \\
  \text{AVAII}(C) &= \\
  \text{AVAII}(D) &= \\
  &= \\
  &= \\
  &= \\
  &= \\
\end{align*}
\]

Iterative algorithm

Algorithm

\[
\begin{align*}
  \text{change} &= \text{true}; \\
  \text{while} \ (\text{change}) \\
  \quad \text{change} &= \text{false}; \\
  \quad \text{for each basic block in reverse PostOrder:} \\
  \quad &\quad \text{solve for } b \\
  \quad &\quad \text{if (old } \neq \text{ new) } \\
  \quad &\quad \quad \text{change} = \text{true}; \\
  \quad &\quad \text{end for} \\
  \quad \text{end while}
\end{align*}
\]

Speed of solution

- node may change only if some predecessor changes
- try to visit node after all its predecessors
- rPostOrder propagates information quickly
- programs usually converge after 3–4 passes
- use bitvectors for more efficiency
Step 1: PostOrder

\[ \text{PostOrder}(n) = \text{NumNodes} - \text{PostOrder}(n) \]

for each node in

Step 2: PostOrder

\[ \text{count} = \text{count} + 1; \]
\[ \text{PostOrder}(n) = \text{count}; \]
\[ \text{Visit}(s); \]
for each successor of \( n \) not yet visited
\[ \text{Visit}(n); \]
\[ \text{count} = 1; \]
\[ \text{main}(); \]

Step 1: PostOrder and Reverse PostOrder

Loop invariant code motion:

\[ B : = A \]
\[ A : = 2 \]
\[ A : = 5 \]

Constant propagation:

\[ \text{Why is this useful?} \]

- Why is this useful?
  - Particular reference to \( x \) in definitions (or variables) or that may reach a
  - The problem: \( \text{Visit}() \), \( x \) are the assignments (or

Reaching Definitions

Definition

\[ \text{Visit}(\text{root}) \]
\[ \text{count} = 1; \]
\[ \text{main}(); \]

Step 1: PostOrder and Reverse PostOrder

\[ \text{Visit}(\text{root}) \]
\[ \text{count} = 1; \]
\[ \text{main}(); \]
A definition is live at program point $p$ if the variable defined by the definition may be used along some path in the program starting at $p$ without being killed between $d$ and $p$. Otherwise, the definition is dead.

**Why is this useful?**

Global analysis to locate dead assignments.

**Equations:**

1. $\text{DECL}(d) \implies \text{REACH}(d) \land \text{DEF}(d) \implies \text{REACH}(d)$
2. $\text{KILL}(d) \implies \text{REACH}(d)$
3. $\text{REACH}(d) \implies \text{det}(d)$
4. $\text{REACH}(d) \iff \text{DET}(d)$
5. $\text{REACH}(d) \iff \text{KILL}(d)$
6. $\text{REACH}(d) \iff \text{DECL}(d)$

A definition $d$ is live at program point $p$ if the path from the definition to $p$ does not exist and the definition is not killed along the path. A definition is dead if it is not live.
LIVE variables

What do these have in common?

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

KILL\(b\) is as before.

USE\(x\) is the set of locally exposed uses.

Suc\(c\)\(b\) is the set of basic blocks that are immediately successors of b in the control flow.

Best case for \(\text{LIVE}\)\(b\): all definitions.

Worse case for \(\text{LIVE}\)\(b\): \(\emptyset\).

General equations:

\[
\begin{align*}
\text{IN}(b) \cap \text{PRE}(b) &= (q) \\
\text{OUT}(b) &\subseteq (q) \\
\text{REACH}(b) \cup \text{DEF}(b) &= (q) \\
\text{AVAIL}(b) \cap \text{KILL}(b) &= (q) \\
\end{align*}
\]

\[
\begin{align*}
\text{LIVE}(b) &= (q) \\
\text{US}\text{E}(x) \cap \text{LIVE}(x) &= (q) \\
\text{USE}(x) \cup \text{KILL}(x) &= (q) \\
\end{align*}
\]

\[
\begin{align*}
\text{AVAIL}(b) \cap \text{KILL}(b) &= (q) \\
\text{REACH}(b) \cup \text{DEF}(b) &= (q) \\
\text{IN}(b) \cap \text{PRE}(b) &= (q) \\
\end{align*}
\]

Confluence Operator or Meet Function:

\[
\begin{align*}
\text{union or intersection} \\
\end{align*}
\]

Behavior for block: GEN and KILL

A direction: forward (confluence over predecessors) or backward (over successors)

Reverse graph for backward data-flow problem.

Graph: Immediate successors of \(b\) in the control flow.

Program:

Block is based on what happens later in the

Similarly different, since information at basic

\[
\begin{align*}
\text{Best case for \text{LIVE}\(b\): all definitions.} \\
\text{Worse case for \text{LIVE}\(b\): } \emptyset. \\
\text{LIVE}\(b\) \cap \text{KILL}\(b\) \subseteq (q). \\
\text{IN}\(b\) \cap \text{PRE}\(b\) \subseteq (q). \\
\text{OUT}\(b\) \cap \text{DEF}\(b\) \subseteq (q). \\
\text{AVAIL}\(b\) \cap \text{KILL}\(b\) \subseteq (q). \\
\text{REACH}\(b\) \cup \text{DEF}\(b\) \subseteq (q). \\
\text{IN}\(b\) \cap \text{PRE}\(b\) \subseteq (q). \\
\end{align*}
\]

Confluence Operator or Meet Function:

\[
\begin{align*}
\text{union or intersection} \\
\end{align*}
\]

Behavior for block: GEN and KILL

A direction: forward (confluence over predecessors) or backward (over successors)

Reverse graph for backward data-flow problem.

Graph: Immediate successors of \(b\) in the control flow.

Program:

Block is based on what happens later in the

Similarly different, since information at basic
Use same framework for all data analysis problems

A lattice has three components:

1. A lattice may have a top element $\top$
   \[
   \top \geq a, \quad \forall a \in \mathcal{A}(T)
   \]
   \[
   \top = a \lor \top, \quad \forall a \in \mathcal{A}(T)
   \]

2. A lattice may have a bottom element $\bot$
   \[
   q \neq a \quad \text{and} \quad q \leq a \iff q < a
   \]
   \[
   q = q \lor q = q \lor a
   \]
   \[
   \bot \leq q \quad \forall q \in \mathcal{A}(T)
   \]

3. A lattice imposes a partial order on $\mathcal{A}(T)$
   \[
   \forall a, b \in \mathcal{A}(T)
   \]
   \[
   a \leq b \iff a \lor b = b
   \]
   \[
   a \leq b \iff a \lor b = b
   \]

Definitions:

- Associative
  \[
  a \lor (b \lor c) = (a \lor b) \lor c
  \]
- Commutative
  \[
  a \lor b = b \lor a
  \]
- Idempotent
  \[
  a \lor a = a
  \]

Given local information $GEN$, $INIT$

Use same framework for all data flow problems

Usefulness of unified framework

- $\mathcal{L}$:
  \[
  \mathcal{L} : \mathcal{A}(T) \rightarrow \mathcal{T}
  \]
  \[
  \lor \quad \text{Operator for combining values}
  \]
  \[
  \text{Domain of values } T
  \]
  \[
  \land \quad \text{Set of transfer functions } T
  \]

Data Flow Analysis Frameworks

Can re-use code to solve new analysis problems

Can describe speed of convergence and precision

Correctness, Convergence:

- Defines a collection of properties that guarantee convergence

Data Flow Lattices
Data-flow lattices

How do these relate to data-flow analysis?

Example - AVAIL

Properly facts around the graph

To model behavior of

Each node n has a distinct set of known facts

Attach to each element of T a meaning

Choose a semi-lattice L to represent facts

How does this relate to data-flow analysis?

Data-flow lattices

\( L = \top \)

\( \perp = \bot \)

Partial ordering

Single lattice vs. one for each variable

\{ \text{e}_1, \text{e}_2 \} \cup \{ \text{e}_3 \}

Avail expression example:

Let D be

\( U = \bigvee \{ \{ \text{e}_1, \text{e}_2, \text{e}_3 \} \times \{ x \} \} \)

Data-flow lattices
Iterative Algorithm

What about loops?

Example

C = A + B

termination

Goal is for solutions to converge to a fixed point

can stop once solutions stop changing

Monotonicity

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]

Framework (D, \{ \land \}) is monotone if

Monotone frameworks are guaranteed to converge and terminate (if lattice elements can

will not yield "larger" output.

Intuitively, monotonicity means "smaller input

implies smaller output". A framework will always give a "smaller or equal" result to the same input. This means the framework is monotone if

\[(h \land x) f \leq (h \land x) f \]
for a distributive framework, MFP = MOP

\((\forall x \in X) f(x) = (\forall x \in X) f'\)

a framework is distributive if

\[ \text{MFP is unique, regardless of order of} \]
\[ \text{in some sense, MOP is best feasible solution} \]
\[ \text{in general, MFP} \geq \text{MOP} \geq \text{Perfect Solution} \]

\[ f(x) = f(x') \]

for a distributive framework, MFP = MOP

\[ \text{Potential paths in control flow graph} \]
\[ \text{Potential paths in control flow graph} = \text{Meet over all paths (MOP)} \]
\[ \text{Meet over all paths (MOP)} = \text{Meet over} \]
\[ \text{Meet over all paths (MOP)} = \text{Meet over} \]

Possible solutions

Quality of Solution