Data Flow Analysis

Data flow analysis is used to collect information about the flow of data values across basic blocks.

- Dominator analysis collected global information regarding the program’s structure.
- For performing global code optimizations global information must be collected regarding values of program values.
  - Local optimizations involve statements from same basic block.
  - Global optimizations involve statements from different basic blocks → data flow analysis is performed to collect global information that drives global optimizations.
Local and Global Optimization

Applications of Data Flow Analysis

- Applicability of code optimizations
- Symbolic debugging of code
- Static error checking
- Type inference
- ......
1. Reaching Definitions

Definition d of variable v: a statement d that assigns a value to v.

Use of variable v: reference to value of v in an expression evaluation.

Definition d of variable v reaches a point p if there exists a path from immediately after d to p such that definition d is not killed along the path.

Definition d is killed along a path between two points if there exists an assignment to variable v along the path.

Example

d reaches u along path$_2$ & d does not reach u along path$_1$

Since there exists a path from d to u along which d is not killed (i.e., path$_2$), d reaches u.
Reaching Definitions Contd.

Unambiguous Definition: \( X = \ldots; \)
Ambiguous Definition: \( *p = \ldots; \) \( p \) may point to \( X \)

For computing reaching definitions, typically we only consider kills by unambiguous definitions.

\[
X = \ldots \\
*p = \ldots
\]

Does definition of \( X \) reach here? Yes

Computing Reaching Definitions

At each program point \( p \), we compute the set of definitions that reach point \( p \).

Reaching definitions are computed by solving a system of equations (data flow equations).

\[
\text{IN}[B] \quad \text{OUT}[B] \quad \text{GEN}[B] = \{d1\} \quad \text{KILL}[B] = \{d2, d3\}
\]

\[
d1: X = \ldots \\
d2: X = \ldots \\
d3: X = \ldots
\]
Data Flow Equations

\[ \text{IN}[B] = \bigcup_{p \in \text{pred}(B)} \text{OUT}(p) \]

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

**IN\[B\]**: Definitions that reach B’s entry.

**OUT\[B\]**: Definitions that reach B’s exit.

**GEN\[B\]**: Definitions within B that reach the end of B.

**KILL\[B\]**: Definitions that never reach the end of B due to redefinitions of variables in B.

Reaching Definitions Contd.

- **Forward** problem – information flows forward in the direction of edges.
- **May** problem – there is a path along which definition reaches a point but it does not always reach the point. Therefore in a May problem the meet operator is the **Union** operator.
Applications of Reaching Definitions

• Constant Propagation/folding

• Copy Propagation

2. Available Expressions

An expression is generated at a point if it is computed at that point.

An expression is killed by redefinitions of operands of the expression.

An expression A+B is available at a point if every path from the start node to the point evaluates A+B and after the last evaluation of A+B on each path there is no redefinition of either A or B (i.e., A+B is not killed).
Available Expressions

Available expressions problem computes: at each program point the set of expressions available at that point.

Data Flow Equations

\( \text{IN}[B] \): Expressions available at B’s entry.
\( \text{OUT}[B] \): Expressions available at B’s exit.

\[
\text{IN}[B] = \bigcap_{P \in \text{pred}(B)} \text{OUT}(P)
\]

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

\( \text{GEN}[B] \): Expressions computed within B that are available at the end of B.
\( \text{KILL}[B] \): Expressions whose operands are redefined in B.
Available Expressions Contd.

- **Forward** problem - information flows forward in the direction of edges.
- **Must** problem - expression is definitely available at a point along all paths.

Therefore in a Must problem the meet operator is the **Intersection** operator.

- Application:

```
A + B is available here
\[ X = A + B \]
\[ Y = A + B \]
\[ Z = A + B \]
\[ T = X = A + B \]
\[ T = Y = A + B \]
\[ Z = T \]
```

3. Live Variable Analysis

A path is **X-clear** if it contains no definition of \( X \).
A variable \( X \) is **live** at point \( p \) if there exists a X-clear path from \( p \) to a use of \( X \); otherwise \( X \) is **dead** at \( p \).

```
X = ...
\[ X \text{ is live} \]
\[ \ldots = X \]
\[ \text{Read}(X) \]
\[ \ldots = X \]
\[ \ldots = X \]
\[ X \text{ is dead} \]
```

**Live Variable Analysis**

Computes:
At each program point \( p \) identify the set of variables that are live at \( p \).
Data Flow Equations

\[ \text{IN}[B] = \bigcup_{s \in \text{succ}(B)} \text{IN}[s] \]
\[ \text{OUT}[B] = \text{GEN}[B] \cup (\text{OUT}[B] \setminus \text{KILL}[B]) \]

**GEN[B]:** Variables that are used in B prior to their definition in B.

**KILL[B]:** Variables definitely assigned value in B before any use of that variable in B.

Live Variables Contd.

- **Backward** problem - information flows backward in reverse of the direction of edges.
- **May** problem - there exists a path along which a use is encountered. Therefore in a May problem the meet operator is the **Union** operator.
Applications of Live Variables

- Register Allocation
- Dead Code Elimination
- Code Motion Out of Loops

4. Very Busy Expressions

A expression $A+B$ is very busy at point $p$ if for all paths starting at $p$ and ending at the end of the program, an evaluation of $A+B$ appears before any definition of $A$ or $B$.

Application: Code Size Reduction

Compute for each program point the set of very busy expressions at the point.
## Data Flow Equations

**IN[B]**: Expressions very busy at B’s entry.

**OUT[B]**: Expressions very busy at B’s exit.

\[
\begin{align*}
\text{OUT}[B] &= \bigcap_{S \in \text{Succ}(B)} \text{IN}[S] \\
\text{IN}[B] &= \text{GEN}[B] \cup (\text{OUT}[B] - \text{KILL}[B])
\end{align*}
\]

**GEN[B]**: Expression computed in B and variables used in the expression are not redefined in B prior to expression’s evaluation in B.

**KILL[B]**: Expressions that use variables that are redefined in B.

## Very Busy Expressions Contd.

- **Backward** problem - information flows backward in reverse of the direction of edges.

- **Must** problem - expressions must be computed along all paths.
  
  Therefore in a Must problem the meet operator is the **Intersection** operator.
Summary

<table>
<thead>
<tr>
<th>May/Union</th>
<th>Must/Intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reaching Definitions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live Variables</td>
</tr>
</tbody>
</table>

Conservative Analysis

Optimizations that we apply must be Safe => the data flow facts we compute should definitely be true (not simply possibly true).

Two main reasons that cause results of analysis to be conservative:
1. Control Flow
2. Pointers & Aliasing
Conservative Analysis

1. Control Flow – we assume that all paths are executable; however, some may be infeasible.

Conservative Analysis

2. Pointers & Aliasing – we may not know what a pointer points to.
   1. \( X = 5 \)
   2. \( \ast p = \ldots \) // \( p \) may or may not point to \( X \)
   3. \( \ldots = X \)

   Constant propagation: assume \( p \) does point to \( X \) (i.e., in statement 3, \( X \) cannot be replaced by 5).
   Dead Code Elimination: assume \( p \) does not point to \( X \) (i.e., statement 1 cannot be deleted).
Representation of Data Flow Sets

- **Bit vectors** - used to represent sets because we are computing binary information.
  - Does a definition reach a point? T or F
  - Is an expression available/very busy? T or F
  - Is a variable live? T or F
- For each expression, variable, definition we have one bit - intersection and union operations can be implemented using bitwise and & or operations.

Solving Data Flow Equations

**Iterative Approach**
- Initialize sets
- Iterate over the sets till they stabilize

**Example - Forward Problem (Available Expressions)**

\[
\begin{align*}
\text{in}(0) &: \emptyset, \quad \text{out}(0) : \text{gen}(0) \\
\text{for } i = 1 \text{ to } n \text{ do } \text{out}(i) &: \{\text{expr}_{i-1}\} - \text{null}(i) \\
\text{change} &: \text{false} \\
\text{while change} \text{ do} \\
\text{change} &: \text{false} \\
\text{in each block } &\neq 0 \text{ do} \\
\text{out}(i) &: \text{out}(i) \\
\text{for }\text{each} \\
\text{out}(i) &: \text{gen}(0) \cup (\text{in}(i) \times \text{null}(i)) \\
\text{if out}(i) \neq \text{null} \text{ then change} &: \text{true} \\
\text{end while} \\
\end{align*}
\]

- Start with largest eliminated & eliminate until the solution till it stabilizes.
Solving Data Flow Equations

Iterative Approach

Example - backtrace problem (five variables)

for \( x = 1 \) to \( N \) do
\( \text{IN}[x] = \text{GEN}[x] \)
endfor

\( \text{OUT}(x) = x \)

change = true

while change do
change = false

for each block \( B \) do
\( \text{OLDIN} = \text{IN}[B] \)
\( \text{OUT}[B] = \cup \text{IN}[S] \text{success}(S) \)
\( \text{IN}[B] = \text{GEN}(B) \cup (\text{OUT}[B] - \text{KILL}[B]) \)
if \( \text{OLDIN} \neq \text{IN}[B] \) then change = true
endfor
endwhile

\( \) - start with smallest solution and keep expanding until it stops shrinking.

Alternative Approach: Worklist Algorithm

Example - backtrace problem (five expressions)

for \( x = 1 \) to \( N \) do
\( \text{IN}[x] = \{ \text{Any expression} \} - \text{KILL}[B] \)
endfor

\( \text{OUT}(x) = x \)

worklist = all blocks

while worklist \( \neq \emptyset \) do
get \( B \) from worklist
\( \text{OLDIN} = \text{IN}[B] \)
\( \text{OUT}[B] = \cup \text{IN}[S] \text{success}(S) \)
\( \text{IN}[B] = \text{GEN}(B) \cup (\text{OUT}[B] - \text{KILL}[B]) \)
if \( \text{OLDIN} \neq \text{IN}[B] \) then
add \( \text{Pred}(B) \) to worklist
endwhile

\( \) - start with target solution and keep iterating until it stops shrinking.
Use-Def & Def-Use Chains

Directly link instructions that produce values with instructions that consume values.

<table>
<thead>
<tr>
<th>Use-def</th>
<th>Def-use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use chain for some variable use ( A ) is the list of pointers to all definitions of the variable that reach ( A )</td>
<td></td>
</tr>
<tr>
<td>Def chain for some variable definition ( A ) is the list of pointers to all uses of the variable that are reachable from ( A )</td>
<td></td>
</tr>
</tbody>
</table>

Block B: \( \text{ud}(A, i) \rightarrow \{ f \} \)

Block B: \( \text{du}(A, i) \rightarrow \text{in}(b) \)

Reaching definitions

Reachable self

\( \text{du}(A, i) \rightarrow \{ f \} \)

= \( \text{iu} \) \cup \text{out}^{+}(b) \)

Reachable self

(Slight correction)

Live variables

31