Available Expressions
An expression e is available at B if every path to B contains a computation of e from defs that are live at B;

Local Analysis:
Gen(B):
Kill(B):

Data Flow Equations:
In(B):
Out(B):
Initialize to
In(B) =
Out(B) =

What is common?

Flow graph: CFG
Local Analysis: Gen, Pres
Meet Function: \( \bigcup \)
Direction in flow graph: forward, backward
Data Flow Equations:
In(B) = \( \bigcup_{P \text{ pred of } B} Out(P) \)
Out(B) = Gen(B) \( \bigcup \) (In(B) \( \setminus \) Pres(B))
General case function:
Out(B) = \( \{ B \} \operatorname{In}(B) \)
General Data Flow Framework

Flow Graph $\mathcal{F}$: Ex CFG, Reverse CFG, call graph

Lattice $\mathcal{L} = (D, \sqcap, \sqcup)$: Top, Bottom $\top, \bot$

Induced partial order $X \preceq Y$

Set of Transfer functions $f : D \rightarrow D$

- include identity, constant functions
- closed under composition of functions, $\circ$

Function assigned to each node to summarize its effects

Example: Liveness Problem, 2 variables

Flow Graph $\mathcal{F}$: Reverse CFG

Lattice $\mathcal{L} = (D, \sqcap, \sqcup)$: Top, Bottom $\top, \bot$

$\top = \phi$

$\{x\}$ $\sqcap$ $\{y\}$

$\{x, y\} = \bot$

elements of $D$ form the lattice

Set of Transfer functions

All functions $D \rightarrow D$
Iterative Algorithm

**Initialization:** Reverse CFG if backward problem.
Initialize \( \text{Out}(B), \text{In}(B) \)

**Worklist Algorithm:**
- \( \text{Worklist} \) := Set of all nodes
- While (\( \text{Worklist} \neq 0 \))
  - Remove node \( N \) from worklist
  - \( \text{OldOut} := \text{Out}(N) \)
  - \( \text{In}(B) = \bigcap_{P \text{ pred of } B} \text{Out}(P) \)
  - \( \text{Out}(B) = f_{\phi}(\text{In}(B)) \)
  - if (\( \text{Out}(N) \neq \text{OldOut} \)) then add successors of \( N \) to \( \text{Worklist} \)

---

**How know iterative algorithm works?**

**Termination:**
- **Finite Lattice** Functions have finite no. of possible values
- **Monotone** Functions are non-increasing or decreasing

**Iterative Method guaranteed to terminate:**
Iterate till no change, all equations simultaneously satisfied

**Quality of solution?**
Quality of Iterative Solution

**Best Solution:** Holds for all real paths taken during program execution

**Meet-Over-all Paths (MOP):**

Iterative solution over all paths

\[
MOP(B) = \bigcap_{\text{all paths } p \text{ from Entry to } B} \bigcap_{i} f_{p} \ (i) \preceq \text{Best}
\]

where \( l \) is the initial lattice value at Entry

and \( f_{p} \) is \( f_{B_{n}} \circ \cdots \circ f_{B_{1}} \)

for \( p = B_{1} \ldots B_{n} \).

**MOP is undecidable (even if monotonic)**

When can MOP be achieved?

**Distributive property:** \( f(X \sqcap Y) = f(X) \sqcap f(Y) \)

Merge then apply \( f \) same as apply \( f \) then merge

**Distributive \( \rightarrow \) MOP solution via iterative algorithm**

What is not distributive?

**Constant propagation**

**Ex.**

\[
\begin{align*}
X &:= 2 \\
Y &:= 3 \\
\Rightarrow &X + Y
\end{align*}
\]

**Maximal Fixed Point (MFP) Solution:**

Achieved by iterative algorithm on all problems covered.

\( \text{MFP} \leq \text{MOP} \leq \text{Best} \)
Summary Iterative Algorithm

**Complexity:** $O(N^2)$ where $N =$ size of FG

**May take long to converge**
Can improve by good choice of node order, ...

**Simple to implement**

**Handles irreducible graphs**

**Doesn’t recognize program structure**
Loops, intervals

---

Interval-based Data Flow Analysis

**Local Propagation:**
For each interval, in order inner to outer,
collect local info for each node in the interval
use the local info for the interval nodes to collect
info for the node representing the interval

**Global Propagation:**
For each interval, in order outer to inner,
given the IN set for the header to the interval,
propagate global info to all nodes in the interval

**Result:** CFG with global data flow info
Interval-based Solution

**LOCAL:** inner to outer

![Diagram](local_diagram.png)

**GLOBAL:** outer to inner

![Diagram](global_diagram.png)

\[ f_{R1} = \lim_{n \to \infty} (f_{B2} \circ f_{B1})^n \]

**Summary Interval-based Algorithm**

**Complexity:** \( O(\alpha(E) \cdot (E)) \) if reducible, exponential if irreducible

**Backward Problems more difficult**
Reverse CFG can have multi-entry loops

**More complicated implementation**

**Uses more space**

**Irreducible subgraphs handled separately**

**Allows for incremental update**

**Often used in practice**
Reaching Defs Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Code</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L1: Read(N)</td>
<td></td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>Call FOO(N)</td>
<td></td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>I := 1</td>
<td>I3</td>
<td>I6</td>
</tr>
<tr>
<td>4</td>
<td>Repeat Until(I &gt; N)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A(I) := A(I) + 1</td>
<td>A5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>I := I + 1</td>
<td>I6</td>
<td>I3</td>
</tr>
<tr>
<td>7</td>
<td>Endrepeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Print(A(N))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Goto L1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Local: Gen(n), Pres(n) for each node n in l

Global: Out(h) = Gen(h) \(\cup\) (In(l) \(\cap\) Pres(h)) for header h

\[
\begin{align*}
\text{In}(n) &= \bigcup_{p \text{ pred of } n} \text{Out}(p) \\
\text{Out}(n) &= \text{Gen}(n) \bigcup_{h} (\text{In}(h) \setminus \text{Pres}(h))
\end{align*}
\]

Def-Use and Use-Def Chains

Application of Reaching Defs

Auxiliary data structures like CFG

Ex. Def-Use Chains

\[
\begin{align*}
X := & \\
\text{if } (...) \text{ then } X := & \\
\text{else } X := & \\
\end{align*}
\]

Optimizations may operate on Def-Use chains

Can bypass CFG entirely
**Static Single Assignment Form (SSA)**

*Each (static) assignment to a variable renamed*

*All of the uses reached by the assignment renamed*

*Results in a single def that reaches every use*

**Ex.**

\[
\begin{align*}
X &:= X \\
X &:= X_1 := X_2 := \\
\text{if(...) then } X &:= \text{ else } X := \\
\text{else } X &:= X_3 := \\
\text{else } X &:= X_4 := \\
\text{else } X &:= X_5 := (X_3, X_4) := \\
\end{align*}
\]

*More compact Def-Use representation*

**Advantages of SSA**

*More compact Def-Use representation*

**Ex.**

\[
\begin{align*}
X &:= X_1 := X_2 := \ldots := X_N := \\
\text{Def-use chains } O(N) \quad \text{with} \quad O(N) \quad \text{chains}
\end{align*}
\]
More Advantages of SSA

More powerful optimizations
Value Numbering
Program Equivalence
Constant Propagation

Faster optimizations
Constant Propagation
Code Motion

Increased Parallelism

Used in real compilers!
IBM Jikes, Sun, Compaq Swift Java, Tera MTA, Scale,...

SSA Loop Example

\[
\begin{align*}
(1) & \quad \text{Read}(N) \\
(2) & \quad I := 1 \\
(3) & \quad \text{If}(I > N) \text{goto L3} \\
(4) & \quad A(I) := A(I) + 1 \\
(5) & \quad I := I + 1 \\
(6) & \quad \text{goto L2} \\
(7) & \quad \text{L3: Print}(A(N)) \\
(1) & \quad \text{Read}(N) \\
(2) & \quad I := 1 \\
L2 : L3 := \Phi(I1,I2) \\
(3) & \quad \text{If}(I3 > N) \text{goto L3} \\
(4) & \quad A(I3) := A(I3) + 1 \\
(5) & \quad I2 := I2 + 1 \\
(6) & \quad \text{goto L2} \\
(7) & \quad \text{L3: Print}(A(N))
\end{align*}
\]

\( \Phi \) makes merge of values explicit

Qu: How construct SSA form?