Dataflow Analysis

Dragon book, Chapter 9, Section 9.2, 9.3, 9.4
Dataflow Analysis

• Dataflow analysis is a sub-area of static program analysis
  – Used in the compiler back end for optimizations of three-address code and for generation of target code
  – For software engineering: software understanding, restructuring, testing, verification

• Attaches to each CFG node some information that describes properties of the program at that point
  – Based on lattice theory

• Defines algorithms for inferring these properties
  – e.g., fixed-point computation
Map of what is coming next

• Six intraprocedural dataflow analyses
  – Reaching Definitions
  – Live Variables
  – Copy Propagation
  – Available Expressions
  – Very Busy Expressions
  – Constant Propagation
  – Points-to Analysis

• Foundations of dataflow analysis
  – Framework: lattices and transfer functions
  – Meet-over-all-paths
  – Fixed point algorithms and solutions
Analysis 1: Reaching Definitions

• A classical example of a dataflow analysis
  – We will consider *intraprocedural* analysis: only inside a single procedure, based on its CFG

• For a minute, assume CFG nodes are individual instructions, not basic blocks
  – Each node defines two *program points*: immediately before and immediately after

• Goal: identify all connections between variable definitions (“write”) and variable uses (“read”)
  – \( x = y + z \) has a definition of \( x \) and uses of \( y \) and \( z \)
Reaching Definitions

• A definition \( d \) reaches a program point \( p \) if there exists a CFG path that
  – starts at the program point immediately after \( d \)
  – ends at \( p \)
  – does not contain a definition of \( d \) (i.e., \( d \) is not “killed”)

• The CFG path may be \textit{infeasible} (could never occur)
  – Any compile-time analysis has to be \textit{conservative}, so we consider all paths in the CFG

• For a CFG node \( n \)
  – \( \text{IN}[n] \) is the set of definitions that reach the program point immediately before \( n \)
  – \( \text{OUT}[n] \) is the set of definitions that reach the program point immediately after \( n \)
  – Reaching definitions analysis: sets \( \text{IN}[n] \) and \( \text{OUT}[n] \) for each \( n \)
OUT[n1] = { }  
IN[n2] = { }  
OUT[n2] = { d1 }  
IN[n3] = { d1 }  
OUT[n3] = { d1, d2 }  
IN[n4] = { d1, d2 }  
OUT[n4] = { d1, d2, d3 }  
IN[n5] = { d1, d2, d3, d5, d6, d7 }  
OUT[n5] = { d2, d3, d4, d5, d6 }  
IN[n6] = { d2, d3, d4, d5, d6 }  
OUT[n6] = { d3, d4, d5, d6 }  
IN[n7] = { d3, d4, d5, d6 }  
OUT[n7] = { d3, d4, d5, d6 }  
IN[n8] = { d3, d4, d5, d6 }  
OUT[n8] = { d4, d5, d6 }  
IN[n9] = { d3, d5, d6, d7 }  
OUT[n9] = { d3, d5, d6, d7 }  
IN[n10] = { d3, d5, d6, d7 }  
OUT[n10] = { d3, d5, d6, d7 }  
IN[n11] = { d3, d5, d6, d7 }

Examples of relationships:

IN[n2] = OUT[n1]  
IN[n5] = OUT[n4] \cup OUT[n10]  
OUT[n7] = IN[n7]  
OUT[n9] = (IN[n9] – {d1,d4,d7}) \cup {d7}
Formulation as a System of Equations

• For each CFG node $n$

\[
\text{IN}[n] = \bigcup_{m \in \text{Predecessors}(n)} \text{OUT}[m] \quad \text{OUT}[\text{ENTRY}] = \emptyset
\]

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

– GEN[$n$] is a singleton set containing the definition $d$ at $n$
– KILL[$n$] is the set of all other definitions of the variable whose value is changed by $d$

• It can be proven that the “smallest” sets IN[$n$] and OUT[$n$] that satisfy this system are exactly the solution for the Reaching Definitions problem
  – To ponder: how do we know that this system has any solutions at all? how about a unique smallest one?
Iteratively Solving the System of Equations

\[ \text{OUT}[n] = \emptyset \] for each CFG node \( n \)

\( change = \text{true} \)

While (\( change \))

1. For each \( n \) other than \( \text{ENTRY} \)
   \( \text{OUT}_{\text{old}}[n] = \text{OUT}[n] \)

2. For each \( n \) other than \( \text{ENTRY} \)
   \( \text{IN}[n] = \text{union of OUT}[m] \) for all predecessors \( m \) of \( n \)

3. For each \( n \) other than \( \text{ENTRY} \)
   \( \text{OUT}[n] = ( \text{IN}[n] - \text{KILL}[n] ) \cup \text{GEN}[n] \)

4. \( change = \text{false} \)

5. For each \( n \) other than \( \text{ENTRY} \)
   If \( (\text{OUT}_{\text{old}}[n] \neq \text{OUT}[n]) \) \( change = \text{true} \)
Questions

• What are the guarantees that this algorithm terminates?
• Does it compute a **correct** solution for the system of equations?
• Does it compute **the smallest** solution for the system of equations?
  – Assuming that there is a unique smallest solution
• How do we even know that this solution is the desired solution for Reaching Definitions?
• We will revisit these questions later, when considering the general machinery of dataflow analysis frameworks
Better Algorithm: Round-Robin, in Order

\[ \text{OUT}[n] = \emptyset \text{ for each CFG node } n \]

\( \text{change} = \text{true} \)

While (\( \text{change} \))

\( \text{change} = \text{false} \)

For each \( n \) other than ENTRY, in rev. postorder

\[ \text{OUT}_{\text{old}}[n] = \text{OUT}[n] \]

\[ \text{IN}[n] = \text{union of OUT}[m] \text{ for all predecessors } m \text{ of } n \]

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

If (\( \text{OUT}_{\text{old}}[n] \neq \text{OUT}[n] \)) \( \text{change} = \text{true} \)
Alternative: Worklist Algorithm

\[ \text{IN}[n] = \emptyset \text{ for all } n \]

Put the successor of ENTRY on worklist

While (worklist is not empty)

1. Remove a CFG node \( m \) from the worklist
2. \( \text{OUT}[m] = (\text{IN}[m] - \text{KILL}[m]) \cup \text{GEN}[m] \)
3. For each successor \( n \) of \( m \)
   
   \( \text{old} = \text{IN}[n] \)
   
   \( \text{IN}[n] = \text{IN}[n] \cup \text{OUT}[m] \)
   
   If (\( \text{old} \neq \text{IN}[n] \)) add \( n \) to worklist

This is “chaotic” iteration

- The order of adding-to/removing-from the worklist is unspecified
  - e.g., could use stack, queue, set, etc.
- The order of processing of successor nodes is unspecified

Regardless of order, the resulting solution is always the same
A Simpler Formulation

• In practice, an algorithm will only compute \( \text{IN}[n] \)

\[
\text{IN}[n] = \bigcup_{m \in \text{Predecessors}(n)} (\text{IN}[m] - \text{KILL}[m]) \cup \text{GEN}[m]
\]

– Ignore predecessor \( m \) if it is ENTRY

• Worklist algorithm
  – \( \text{IN}[n] = \emptyset \) for all \( n \)
  – Put the successor of ENTRY on the worklist
  – While the worklist is not empty, remove \( m \) from the worklist; for each successor \( n \) of \( m \), do
    • \( \text{old} = \text{IN}[n] \)
    • \( \text{IN}[n] = \text{IN}[n] \cup (\text{IN}[m] - \text{KILL}[m]) \cup \text{GEN}[m] \)
    • If (\( \text{old} \neq \text{IN}[n] \)) add \( n \) to worklist
A Few Notes

• We sometimes write

\[ \text{IN}[n] = \bigcup_{m \in \text{Predecessors}(n)} (\text{IN}[m] \cap \text{PRES}[m]) \cup \text{GEN}[m] \]

• \text{PRES}[n]: the set of all definitions “preserved” (i.e., not killed) by \( n \)

• Efficient implementation: bitvectors
  – Sets are presented by bitvectors; set intersection is bitwise AND; set union is bitwise OR
  – \text{GEN}[n] and \text{PRES}[n] are computed once, at the very beginning of the dataflow analysis
  – \text{IN}[n] are computed iteratively, using a worklist
Reaching Definitions and Basic Blocks

• For space/time savings, we can solve the problem for basic blocks (i.e., CFG nodes are basic blocks)
  – Program points are before/after basic blocks
  – \( \text{IN}[n] \) is still the union of \( \text{OUT}[m] \) for predecessors \( m \)
  – \( \text{OUT}[n] \) is still \( ( \text{IN}[n] - \text{KILL}[n] ) \cup \text{GEN}[n] \)

• \( \text{KILL}[n] = \text{KILL}[s_1] \cup \text{KILL}[s_2] \cup \ldots \cup \text{KILL}[s_k] \)
  – \( s_1, s_2, \ldots, s_k \) are the statements in the basic blocks

• \( \text{GEN}[n] = \text{GEN}[s_k] \cup ( \text{GEN}[s_{k-1}] - \text{KILL}[s_k] ) \cup ( \text{GEN}[s_{k-2}] - \text{KILL}[s_{k-1}] - \text{KILL}[s_k] ) \cup \ldots \cup ( \text{GEN}[s_1] - \text{KILL}[s_2] - \text{KILL}[s_3] - \ldots - \text{KILL}[s_k] ) \)
  – \( \text{GEN}[n] \) contains any definition in the block that is downwards exposed (i.e., not killed by a subsequent definition in the block)
ENTRY

i = m - 1

j = n

a = u1

i = i + 1

KILL[n2] = { d1, d2, d3, d4, d5, d6, d7 }
GEN[n2] = { d1, d2, d3 }

KILL[n3] = { d1, d2, d4, d5, d7 }
GEN[n3] = { d4, d5 }

KILL[n4] = { d3, d6 }
GEN[n4] = { d6 }

KILL[n5] = { d1, d4, d7 }
GEN[n5] = { d7 }

IN[n2] = { }
OUT[n2] = { d1, d2, d3 }

IN[n3] = { d1, d2, d3, d5, d6, d7 }
OUT[n3] = { d3, d4, d5, d6 }

IN[n4] = { d3, d4, d5, d6 }
OUT[n4] = { d4, d5, d6 }

IN[n5] = { d3, d4, d5, d6 }
OUT[n5] = { d3, d5, d6, d7 }

EXIT
Uses of Reaching Definitions Analysis

- **Def-use (du) chains**
  - For a given definition (i.e., write) of a memory location, which statements read the value created by the def?
  - For basic blocks: all **upward-exposed uses** (use of variable does not have preceding def in the same basic block)

- **Use-def (ud) chains**
  - For a given use (i.e., read) of a memory location, which statements performed the write of this value?
  - The reverse of du-chains

- **Goal: potential write-read (flow) data dependences**
  - Compiler optimizations
  - Program understanding (e.g., slicing)
  - Dataflow-based testing: coverage criteria
  - Semantic checks: e.g., use of uninitialized variables
  - Could also find write-write (output) dependences
ENTRY

\[i = m-1\]

\[j = n\]

\[a = u1\]

\[i = i + 1\]

\[d1\]

\[d2\]

\[d3\]

\[a = u2\]

\[i = u3\]

\[\text{if}(..i..a)\]

\[\text{EXIT}\]

Upward exposed uses:

\[\text{USES}[n2] = \{ m@d1, n@d2, u1@d3 \}\]

\[\text{USES}[n3] = \{ i@d4, j@d5, a@c1 \}\]

\[\text{USES}[n4] = \{ u2@d6 \}\]

\[\text{USES}[n5] = \{ u3@d7, j@c2, a@c2 \}\]

Reaching definitions:

\[\text{IN}[n3] = \{ d1, d2, d3, d5, d6, d7 \}\]

\[\text{IN}[n4] = \{ d3, d4, d5, d6 \}\]

\[\text{IN}[n5] = \{ d3, d4, d5, d6 \}\]

Def-use chains across basic blocks:

\[\text{DU}[d1] = \text{upward exposed uses of variable } i \text{ in all basic blocks } n \text{ such that } d1 \in \text{IN}[n] = \{ i@d4 \}\]

\[\text{DU}[d2] = \{ j@d5 \}\]

\[\text{DU}[d3] = \{ a@c1, a@c2 \}\]

\[\text{DU}[d4] = \{ \}\]

\[\text{DU}[d5] = \{ j@d5, j@c2 \}\]

\[\text{DU}[d6] = \{ a@c1, a@c2 \}\]

\[\text{DU}[d7] = \{ i@d4 \}\]

Def-use chains inside basic blocks:

\[\text{DU}[d4] = \{ i@c1 \}\]

Use-def chains:

\[\text{UD}[m@d1]= \{ \}\]

\[\text{UD}[n@d2]= \{ \}\]

\[\text{UD}[u1@d3]= \{ \}\]

\[\text{UD}[i@d4]= \{ d1,d7 \}\]

\[\text{UD}[j@d5]= \{ d2,d5 \}\]

\[\text{UD}[i@c1]= \{ d4 \}\]

\[\text{UD}[a@c1]= \{ d3,d6 \}\]

\[\text{UD}[u2@d6]= \{ \}\]

\[\text{UD}[u3@d7]= \{ \}\]

\[\text{UD}[j@c2]= \{ d5 \}\]

\[\text{UD}[a@c2]= \{ d3,d6 \}\]
Analysis 2: Live Variables

• A variable $v$ is **live** at a program point $p$ if there exists a CFG path that
  - starts at $p$
  - ends at a statement that reads $v$
  - does **not** contain a definition of $v$

• Thus, the value that $v$ has at $p$ could be used later
  - “could” because the CFG path may be infeasible
  - If $v$ is not live at $p$, we say that $v$ is **dead** at $p$

• For a CFG node $n$
  - $\text{IN}[n]$ is the set of variables that are live at the program point immediately before $n$
  - $\text{OUT}[n]$ is the set of variables that are live at the program point immediately after $n$
ENTRY

\[
i = m-1
\]

\[
j = n
\]

\[
a = u1
\]

\[
i = i + 1
\]

\[
j = j - 1
\]

\[
\text{if (...)}
\]

\[
a = u2
\]

\[
i = u3
\]

\[
\text{if (...)}
\]

\[
\text{EXIT}
\]

OUT[\text{n1}] = \{ m, n, u1, u2, u3 \}

IN[n2] = \{ m, n, u1, u2, u3 \}

OUT[n2] = \{ n, u1, i, u2, u3 \}

IN[n3] = \{ n, u1, i, u2, u3 \}

OUT[n3] = \{ u1, i, j, u2, u3 \}

IN[n4] = \{ u1, i, j, u2, u3 \}

OUT[n4] = \{ i, j, u2, u3 \}

IN[n5] = \{ i, j, u2, u3 \}

OUT[n5] = \{ j, u2, u3 \}

IN[n6] = \{ j, u2, u3 \}

OUT[n6] = \{ u2, u3, j \}

IN[n7] = \{ u2, u3, j \}

OUT[n7] = \{ u2, u3, j \}

IN[n8] = \{ u2, u3, j \}

OUT[n8] = \{ u3, j, u2 \}

IN[n9] = \{ u3, j, u2 \}

OUT[n9] = \{ i, j, u2, u3 \}

IN[n10] = \{ i, j, u2, u3 \}

OUT[n10] = \{ i, j, u2, u3 \}

IN[n11] = \{ \}

Examples of relationships:

OUT[n1] = IN[n2]

OUT[n7] = IN[n8] \cup IN[n9]

IN[n10] = OUT[n10]

IN[n2] = (OUT[n2] - \{i\}) \cup \{m\}
Formulation as a System of Equations

- For each CFG node $n$

\[
\text{OUT}[n] = \bigcup_{m \in \text{Successors}(n)} \text{IN}[m] \quad \text{IN}[\text{EXIT}] = \emptyset
\]

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

- GEN$[n]$ is the set of all variables that are read by $n$
- KILL$[n]$ is a singleton set containing the variable that is written by $n$ (even if this variable is live immediately after $n$, it is not live immediately before $n$)

- The smallest sets IN$[n]$ and OUT$[n]$ that satisfy this system are exactly the solution for the Live Variables problem
Iteratively Solving the System of Equations

IN\[n\] = \emptyset for each CFG node \(n\)

\(change = true\)

While \((change)\)

1. For each \(n\) other than EXIT
   \(\text{IN}_{\text{old}}[n] = \text{IN}[n]\)

2. For each \(n\) other than EXIT
   \(\text{OUT}[n] = \text{union of} \ \text{IN}[m] \text{ for all successors} \ m \text{ of} \ \ n\)

3. For each \(n\) other than EXIT
   \(\text{IN}[n] = (\text{OUT}[n] - \text{KILL}[n]) \cup \text{GEN}[n]\)

4. \(\text{change} = false\)

5. For each \(n\) other than EXIT
   If \((\text{IN}_{\text{old}}[n] \neq \text{IN}[n])\) \(\text{change} = true\)

Better version: round-robin algorithm, in postorder
Worklist Algorithm

\[ \text{OUT}[n] = \emptyset \text{ for all } n \]

Put the predecessors of EXIT on worklist

While (worklist is not empty)

1. Remove a CFG node \( m \) from the worklist
2. \( \text{IN}[m] = (\text{OUT}[m] - \text{KILL}[m]) \cup \text{GEN}[m] \)
3. For each predecessor \( n \) of \( m \)
   \[ \text{old} = \text{OUT}[n] \]
   \[ \text{OUT}[n] = \text{OUT}[n] \cup \text{IN}[m] \]
   If (\( \text{old} \neq \text{OUT}[n] \)) add \( n \) to worklist

As with the worklist algorithm for Reaching Definitions, this is chaotic iteration. But, regardless of order, the resulting solution is always the same.
A Simpler Formulation

• In practice, an algorithm will only compute $\text{OUT}[n]$

$$\text{OUT}[n] = \bigcup_{m \in \text{Successors}(n)} (\text{OUT}[m] - \text{KILL}[m]) \cup \text{GEN}[m]$$

  – Ignore successor $m$ if it is EXIT

• Worklist algorithm
  – $\text{OUT}[n] = \emptyset$ for all $n$
  – Put the predecessors of EXIT on the worklist
  – While the worklist is not empty, remove $m$ from the worklist; for each predecessor $n$ of $m$, do
    • $old = \text{OUT}[n]$
    • $\text{OUT}[n] = \text{OUT}[n] \cup (\text{OUT}[m] - \text{KILL}[m]) \cup \text{GEN}[m]$
    • If ($old \neq \text{OUT}[n]$) add $n$ to worklist
A Few Notes

• We sometimes write

\[ \text{OUT}[n] = \bigcup_{m \in \text{Successors}(n)} (\text{OUT}[m] \cap \text{PRES}[m]) \cup \text{GEN}[m] \]

– \text{PRES}[n]: the set of all variables “preserved” (i.e., not written) by \( n \)
– Efficient implementation: bitvectors

• Comparison with Reaching Definitions
– Reaching Definitions is a forward dataflow problem and Live Variables is a backward dataflow problem
– Other than that, they are basically the same

• Uses of Live Variables
– Dead code elimination: e.g., when \( x \) is not live at \( x = y + z \)
– Register allocation (more on this in CSE 756)
Analysis 3: Copy Propagation

• Copy propagation: for \( x = y \), replace subsequent uses of \( x \) with \( y \), as long as \( x \) and \( y \) have not changed along the way
  – Creates opportunities for **dead code elimination**: e.g., after copy propagation we may find that \( x \) is not live

1) Dead code elimination: \( b=a \)

2) Strength reduction: \( e=a+a \) use left shift instead of addition
Formulation as a System of Equations

- For each CFG node \( n \) (assume nodes = instructions)
  
  \[
  \text{IN}[n] = \bigcap_{m \in \text{Predecessors}(n)} \text{OUT}[m] \\
  \text{OUT}[ENTRY] = \emptyset
  \]

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

- \( \text{IN}[n] \) is a set of copy instructions \( x=y \) such that neither \( x \) nor \( y \) is assigned along any path from \( x=y \) to \( n \)

- \( \text{GEN}[n] \) is
  - A singleton set containing the copy instruction, if \( n \) is a copy instruction
  - The empty set, otherwise

- \( \text{KILL}[n] \): if \( n \) assigns to \( x \), kill every \( y=x \) and \( x=y \)

- Note that we must use \textit{intersection} of \( \text{OUT}[m] \)
Worklist Algorithm

IN\([n]\) = \textit{the set of all copy instructions}, for all \(n\)

Put the successor of ENTRY on \textit{worklist}

While \((\text{worklist} \text{ is not empty})\)
  1. Remove a CFG node \(m\) from the worklist
  2. \(\text{OUT}[m] = (\text{IN}[m] – \text{KILL}[m]) \cup \text{GEN}[m]\)
  3. For each successor \(n\) of \(m\)
     \(\text{old} = \text{IN}[n]\)
     \(\text{IN}[n] = \text{IN}[n] \cap \text{OUT}[m]\)
     If \((\text{old} \neq \text{IN}[n])\) add \(n\) to \textit{worklist}

In Reaching Definitions, we initialized \(\text{IN}[n]\) to the empty set; here we cannot do this, because of \(\text{IN}[n] = \text{IN}[n] \cap \text{OUT}[m]\)

\(\bullet\) Here the “meet” operator of the lattice is \textit{set intersection}; the top element of the lattice is the set of all copy instructions

\(\bullet\) In Reaching Definitions, “meet” is \textit{set union}; “top” is the empty set
Classification

- **Forward vs backward problems**: intuitively, do we need to go forward along CFG paths, or backward?
  - Reaching Definitions: forward; Live Variables: backward; Copy Propagation: forward

- **May vs must problems**
  - Reaching Definitions: a definition may reach (union over predecessors – i.e., $\exists$ path ...)
  - Live Variables: a use may be reached (union over successors – i.e., $\exists$ path ...)
  - Copy Propagation: $x$ and $y$ must be preserved along all paths (intersection over predecessors – i.e., $\forall$ paths ...)
Analysis 4: Available Expressions

- Expression $x \text{ op } y$ is available at program point $p$
  1. Every path from ENTRY to $p$ evaluates $x \text{ op } y$
  2. After the last evaluation along the path, there are no subsequent assignments to $x$ or $y$

- Useful for common subexpression elimination

- **Must** and **forward** problem
  - “Every path” – must problem
  - “From ENTRY to $p$” – forward problem
Common Subexpression Elimination

Example courtesy of Prof. Barbara Ryder
Common Subexpression Elimination

\[ t_1 := a \times b \]
\[ q := t_1 \]
\[ t_1 := a \times b \]
\[ z := t_1 \]
\[ r := 2 \times z \]
\[ u := t_1 \]
\[ z := u / 2 \]
\[ w := a \times b \]

Cannot be eliminated because does not have \( a \times b \) available on all paths
Formulation as a System of Equations

• For each CFG node $n$

\[
\text{IN}[n] = \bigcap_{m \in \text{Predecessors}(n)} \text{OUT}[m] \quad \text{OUT}[\text{ENTRY}] = \emptyset
\]

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

- $\text{IN}[n]$ is a set of expressions $x \text{ op } y$ available at $n$
- $\text{GEN}[n]$ is
  - A singleton set containing the expression $x \text{ op } y$, if $n$ computes that expression
  - The empty set, otherwise
- $\text{KILL}[n]$: if $n$ assigns to $x$, kill every $x \text{ op } y$ and $y \text{ op } x$
- $\text{IN}[n]$ is initialized to the set of all expressions appearing on the right-hand size of any instruction
Analysis 5: Very Busy Expressions

- Expression $x \text{ op } y$ is **very busy** at $p$ if along every path from $p$ we come to a computation of $x \text{ op } y$ before any redefinition of $x$ or $y$
  - Useful for code motion: hoist $x \text{ op } y$ to program point $p$
  - **Backward must** problem

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

\[
\text{OUT}[n] = \bigcap_{m \in \text{Successors}(n)} \text{IN}[m]
\]

- Compare with Live Variables: backward **may** problem

\[
\text{OUT}[n] = \bigcup_{m \in \text{Successors}(n)} \text{IN}[m]
\]
Summary of Analyses 1-5

• Solution at a node is a **subset of a finite set** (thus, sometimes they are called “bitvector” problems)

• Functions are \( f(x) = (A \cap x) \cup B \) – “rapid” problems
  – Fast convergence w/ reverse postorder (forward analysis) or postorder (backward analysis): e.g.
    
    ```
    while (change)
    for each node n in reverse postorder
        IN[n] = ... IN[m]...
    
    \(d+2\) iterations; \( d \) is the max CFG loop nesting depth
    
    – If we use the worklist algorithm (i.e., chaotic iteration) non-determinism in worklist order and in order of successors
Analysis 6: Constant Propagation

• Can we guarantee that the value of a variable $v$ at a program point $p$ is always a known constant?

• Compile-time constants are quite useful
  – Constant folding: e.g., if we know that $v$ is always 3.14 immediately before $w = 2*v$; replace it $w = 6.28$
  • Often due to symbolic constants
  – Dead code elimination: e.g., if we know that $v$ is always false at if $(v) ...$
  – Program understanding, restructuring, verification, testing, etc.
Basic Ideas

• At each CFG node $n$, $\text{IN}[n]$ is a map $\text{Vars} \rightarrow \text{Values}$
  – Each variable $v$ is mapped to a value $x \in \text{Values}$
  – $\text{Values} = \text{all possible constant values} \cup \{nac, \text{undef}\}$

• Special “value” $nac$ (not-a-constant) means that the variable cannot be definitely proved to be a compile-time constant at this program point
  – E.g., the value comes from user input, file I/O, network
  – E.g., the value is 5 along one branch of an if statement, and 6 along another branch of the if statement
  – E.g., the value comes from some $nac$ variable

• Special “value” $\text{undef}$ (undefined): used temporarily during the analysis
  – Means “we have no information about $v$ yet”
Formulation as a System of Equations

• OUT[ENTRY] = a map which maps each v to undef

• For any other CFG node n
  – IN[n] = Merge(OUT[m]) for all predecessors m of n
  – OUT[n] = Update(IN[n])

• Merging two maps: if v is mapped to $c_1$ and $c_2$ respectively, in the merged map v is mapped to:
  – If $c_1 = undef$, the result is $c_2$
  – Else if $c_2 = undef$, the result is $c_1$
  – Else if $c_1 = nac$ or $c_2 = nac$, the result it nac
  – Else if $c_1 \neq c_2$, the result is nac
  – Else the result is $c_1$ (in this case we know that $c_1 = c_2$)
Formulation as a System of Equations

• **Updating** a map at an assignment $v = ...$
  – If the statement is not an assignment, $\text{OUT}[n] = \text{IN}[n]$

• The map does not change for any $w \neq v$

• If we have $v = c$, where $c$ is a constant: in $\text{OUT}[n]$, $v$ is now mapped to $c$

• If we have $v = p + q$ (or similar binary operators) and $\text{IN}[n]$ maps $p$ and $q$ to $c_1$ and $c_2$ respectively
  – If both $c_1$ and $c_2$ are constants: result is $c_1 + c_2$
  – Else if either $c_1$ or $c_2$ is $\text{nac}$: result is $\text{nac}$
  – Else: result is $\text{undef}$
ENTRY

\begin{align*}
a &= 1 \\
b &= 2 \\
c &= a + b \\
\text{if} \ (\ldots) \\
a &= 1 + c \\
b &= 4 + c \\
d &= a + b \\
a &= a + b \\
b &= a + c \\
\text{EXIT}
\end{align*}

\begin{align*}
\text{OUT}[n1] &= \{a \rightarrow \text{undef}, b \rightarrow \text{undef}, c \rightarrow \text{undef}, d \rightarrow \text{undef} \} \\
\text{OUT}[n2] &= \{a \rightarrow 1, b \rightarrow \text{undef}, c \rightarrow \text{undef}, d \rightarrow \text{undef} \} \\
\text{OUT}[n3] &= \{a \rightarrow 1, b \rightarrow 2, c \rightarrow \text{undef}, d \rightarrow \text{undef} \} \\
\text{OUT}[n4] &= \{a \rightarrow 1, b \rightarrow 2, c \rightarrow 3, d \rightarrow \text{undef} \} \\
\text{OUT}[n6] &= \{a \rightarrow 4, b \rightarrow 2, c \rightarrow 3, d \rightarrow \text{undef} \} \\
\text{OUT}[n7] &= \{a \rightarrow 4, b \rightarrow 7, c \rightarrow 3, d \rightarrow \text{undef} \} \\
\text{OUT}[n8] &= \{a \rightarrow 4, b \rightarrow 7, c \rightarrow 3, d \rightarrow 11 \} \\
\text{OUT}[n9] &= \{a \rightarrow 5, b \rightarrow 2, c \rightarrow 3, d \rightarrow \text{undef} \} \\
\text{OUT}[n10] &= \{a \rightarrow 5, b \rightarrow 6, c \rightarrow 3, d \rightarrow \text{undef} \} \\
\text{IN}[n11] &= \{a \rightarrow nac, b \rightarrow nac, c \rightarrow 3, d \rightarrow 11 \} \\
\text{OUT}[n11] &= \{a \rightarrow nac, b \rightarrow nac, c \rightarrow 3, d \rightarrow 11 \} \\
\text{OUT}[n12] &= \{a \rightarrow nac, b \rightarrow nac, c \rightarrow 3, d \rightarrow 11 \} \\
\text{Note: in reality, d could be uninitialized at n11 and n12 (see Section 9.4.6 for a good discussion on this issue)}
\end{align*}
Analysis 7: Points-To Analysis

• Question (oversimplified): can variable $x$ contain the address of variable $y$ at program point $p$?
• First abstraction: no arrays, no structs, no objects, no heap-allocated memory, no pointer arithmetic, no calls
• Instructions of interest
  - $x = \& y$
  - $x = y$
  - $x = *y$
  - $*x = y$
  - $x = \text{null}$
Basic Ideas

• At each CFG node \( n \), \( \text{IN}[n] \) is a set \( \subseteq \text{Vars} \times \text{Vars} \)
  – That is, a set of pairs of variables \((x, y)\)
  – Alternative formulation: map \( \text{Vars} \rightarrow \text{PowerSet(Vars)} \)
    • For each variable \( x \), its points-to set \( \text{Pt}(x) \)

• If for some path from ENTRY to \( n \) the value of \( x \) is the address of \( y \) (when \( n \) is reached), then \((x, y)\) must be an element of \( \text{IN}[n] \)
  – Often defined as “points-to graph”: an edge \( x \rightarrow y \)
    shows that \( x \) may point to \( y \)

• Similarly defined \( \text{OUT}[n] \)
Formulation as a System of Equations

• OUT[ENTRY] = empty set
• For any other CFG node \( n \)
  – \( \text{IN}[n] = \text{Merge} (\text{OUT}[m]) \) for all predecessors \( m \) of \( n \)
  – \( \text{OUT}[n] = \text{Update} (\text{IN}[n]) \)
• Merging two points-to graphs: just the union of their edge sets

1. if (...) goto (4)
2. \( x = \&a \) \quad \text{OUT}[2] = \{ (x,a) \}
3. goto (5)
4. \( x = \&b \) \quad \text{OUT}[4] = \{ (x,b) \}
5. \( z = x \) \quad \text{IN}[5] = \{ (x,a), (x,b) \}; \text{OUT}[5] = \{ (z,a), (z,b), (x,a), (x,b) \}
6. \( w = \&c \) \quad \text{OUT}[6] = \{ (z,a), (z,b), (x,a), (x,b), (w,c) \}
7. \( *z = w \) \quad \text{OUT}[7] = \{ (z,a), (z,b), (x,a), (x,b), (w,c), (a,c), (b,c) \}
8. \( v = *x \) \quad \text{OUT}[8] = \{ (z,a), (z,b), (x,a), (x,b), (w,c), (a,c), (b,c), (v,c) \}
Formulation as a System of Equations

- **Updating** at an assignment $v = ...$ or $*v = ...$
- $x = \textbf{null}$: $\text{OUT}[n] = \text{IN}[n] - \{x\} \times \text{Vars}$
- $x = \&y$: $\text{OUT}[n] = (\text{IN}[n] - \{x\} \times \text{Vars}) \cup \{ (x,y) \}$
- $x = y$: $\text{OUT}[n] = (\text{IN}[n] - \{x\} \times \text{Vars}) \cup \{ (x,z) \mid (y,z) \in \text{IN}[n] \}$
- $x = \ast y$: $\text{OUT}[n] = (\text{IN}[n] - \{x\} \times \text{Vars}) \cup \{ (x,z) \mid (y,w) \in \text{IN}[n] \land (w,z) \in \text{IN}[n] \}$
- $\ast x = y$: $\text{OUT}[n] = (\text{IN}[n] - \text{nothing}) \cup \{ (w,z) \mid (x,w) \in \text{IN}[n] \land (y,z) \in \text{IN}[n] \}$

- Why not kill $(w,...)$? In general, we cannot assert that $x$ definitely points to $w$, even if $(x,w) \in \text{IN}[n]$; more later ...
How About Real Programs?

- \( x = \texttt{malloc}(\ldots) \) or \( x = \texttt{new} \, X(\ldots) \): artificial name
  \[ \text{heap}_i : \text{OUT}[n] = (\text{IN}[n] - \{x\} \times \text{Vars}) \cup \{ (x,\text{heap}_i) \} \]

- \( a[x] = y \): treat array \( a \) as one uniform block of data
  \[ \text{OUT}[n] = \text{IN}[n] \cup \{ (a,z) \mid (y,z) \in \text{IN}[n] \} \]

- \( x = a[y] \): \( \text{OUT}[n] = (\text{IN}[n] - \{x\} \times \text{Vars}) \cup \{ (x,y) \mid (a,y) \in \text{IN}[n] \} \)

- Fields of structs/objects: labels on points-to edges

```c
struct S { int* f1; float* f2; };
struct S* x = malloc(sizeof(struct S));  (x,heap_1)
(*x).f1 = &a; (*x).f2 = &b;   (heap_1,f1,a) (heap_1,f2,b)
y = (*x).f1;                 (y, a)
```

- Many complications: e.g., pointer arithmetic
Approximations

• **Flow-insensitive analysis:** ignore the flow of control and compute one points-to graph for the entire program (rather than a separate points-to graph for each CFG node)

• **Field-insensitive:** do not distinguish between fields

\[ (*x).f1 = \&a; (*x).f2 = \&b; y = (*x).f1; \] treated as \[ *x = \&a; *x = \&b; y = *x; \]

\[ (heap_1,f1,a) (heap_1,f2,b), (y,a) \] becomes \[ (heap_1,a) (heap_1,b), (y,a), (y,b) \]

• **Base-object-insensitive:** treat \[ (*x).f1 \] as \[ f1 \]

Java: \[ x = \text{new } A; y = \text{new } A; x.f = \text{new } C; y.f = \text{new } D; z = y.f \] should lead to \[ (x,heap_1), (y,heap_2), (heap_1,f,heap_3), (heap_2,f,heap_4), (z,heap_4) \]

Instead, it is treated as \[ x = \text{new } A; y = \text{new } A; f = \text{new } C; f = \text{new } D; z = f \]

and leads to \[ (x,heap_1), (y,heap_2), (f,heap_3), (f,heap_4), (z,heap_3),(z,heap_4) \]
Flow-Insensitive Points-to Analysis

• A points-to graph could be $O(n^2)$ in size; a separate graph at each node is often too expensive

• “Fake” CFG with arbitrary sequences of statements

  while ...

  switch ....

    case 1: statement 1

    case 2: statement 2 ....

• Points-to graph at the merge point of the switch

• Simplified functions without “kill” (more efficient):

  $\text{OUT}[n] = (\text{IN}[n] - \{x\} \times \text{Vars}) \cup ...$ becomes

  $\text{OUT}[n] = \text{IN}[n] \cup ...$
Loss of Precision: FI, FS, and Beyond

1. \( x = &a \)  
   **FS:** OUT[1] = \{ (x,a) \}

2. \( y = &b \)  
   **FS:** OUT[2] = \{ (x,a), (y,b) \}

3. \( z = &c \)  
   **FS:** OUT[3] = \{ (x,a), (y,b), (z,c) \}

4. \( *x = y \)  
   **FS:** OUT[4] = \{ (x,a), (y,b), (z,c), (a,b) \}

5. \( *a = \ldots \)  
   dependence between these statements:

6. \( \ldots = c+1 \)  
   **FI:** yes; **FS:** no

7. \( *x = z \)  
   **FS:** OUT[7] = \{ (x,a), (y,b), (z,c), (a,b), (a,c) \}

8. \( *a = \ldots \)  
   dependence between these statements:

9. \( \ldots = b+2 \)  
   **FI and FS:** yes (wrong!)

**FI solution:** (x,a), (y,b), (z,c), (a,b), (a,c)

Can we improve **FS** to eliminate (a,b) from OUT[7]?
FS with Strong Updates

- **Updating** at an assignment \( v = \ldots \) or \( *v = \ldots \)
  - If the statement is not an assignment, \( \text{OUT}[n] = \text{IN}[n] \)

- \( x = \ldots \): \( \text{OUT}[n] = (\text{IN}[n] - \{x\} \times \text{Vars}) \cup \ldots \)

- \( *x = y \): \( \text{OUT}[n] = (\text{IN}[n] - \text{nothing}) \cup \ldots \)
  - Why not kill \((w,\ldots)\) for when \( x \) points to \( w \)? In general, we cannot assert that \( x \) definitely points to \( w \)

- But what if the points-to set of \( x \) is a **singleton set**?
  - E.g., in the previous example, \( \text{Pt}(x) = \{ a \} \): can we kill \((a,\ldots)\) at \(*x = y\)?
  - If we can, \( \text{OUT}[7] \) will become \( \{ (x,a), (y,b), (z,c), (a,c) \} \)
    and the precision is improved
    - False dependence between 8 and 9 disappears
FS with Strong Updates

• Proposal: at \( *x = y \), if \( \text{Pt}(x) \) is a singleton set \( \{ w \} \), perform a strong update on \( w \):
  
  \[ \text{OUT}[n] = (\text{IN}[n] - \{ w \} \times \text{Vars}) \cup \ldots \]

• Not so fast ... remember that \( w \) is just a static abstraction of a set of run-time memory locations; this set itself must be a singleton set

Example: recall field-insensitive analysis

\( x = \text{malloc}; \ (*x).f1 = \&a; \ (*x).f2 = \&b; \ y = (*x).f1; \) treated as \( x = \&\text{heap1}, \ *x = \&a; \ *x = \&b; \ y = *x; \)

• FI without strong updates: at \( *x=\&b, \) \( \text{IN} = \{ (x,\text{heap}_1), (\text{heap}_1,a) \}, \) \( \text{OUT} = \{ (x,\text{heap}_1), (\text{heap}_1,a), (\text{heap}_1,b) \} \) and later we get \( (y,a), (y,b) \)

• With strong updates: \( \text{OUT} = \{ (x,\text{heap}_1),(\text{heap}_1,b) \} \) but \( (y,a) \) is lost!
"Dangerous" Strong Update

Which points-to graph node may correspond to multiple memory locations (and should not be strongly updated)?

- Array: one name for the entire array
- Local variable of a recursive procedures
- Dynamically allocated memory (even with field sensitivity)

```java
curr = null
while (...) {
  1. prev = curr
      \[\text{IN}[1] = \{(prev,heap_1),(curr,heap_1),(y,heap_2),(heap_1,fld,heap_2)\}\]
  2. curr = new X
  3. y = new Y
  4. curr.fld = y
}
\[\text{IN}[5] = \{(prev,heap_1),(curr,heap_1),(y,heap_2),(heap_1,fld,heap_2)\}\]
  5. prev.fld = new Z
      \[\text{OUT}[5] = \{(prev,heap_1),(curr,heap_1),(y,heap_2),(heap_1,fld,heap_3)\}\]
  6. ... curr.fld.fld2 ...
  7. ... y.fld2 ...
```

Dependence between these statements? Yes

With strong updates: No, because heap3.fld2 \(\neq\) heap2.fld2
Foundations of Dataflow Analysis
Partial Order

• Given a set $S$, a relation $r$ between elements of $S$ is a set $r \subseteq S \times S$
  – Notation: if $(x,y) \in r$, write “$x \mathbin{r} y$”
  – Example: “less than” relation over integers

• A relation is a partial order if and only if
  – Reflexive: $x \mathbin{r} x$
  – Anti-symmetric: $x \mathbin{r} y$ and $y \mathbin{r} x$ implies $x = y$
  – Transitive: $x \mathbin{r} y$ and $y \mathbin{r} z$ implies $x \mathbin{r} z$
  – Example: “less than or equal to” over integers
  – By convention, the symbol used for a partial order is $\leq$ or something similar to it (e.g. $\sqsubseteq$)
Partially Ordered Set

- **Partially ordered set** \((S, \leq)\) is a set \(S\) with a defined partial order \(\leq\).

- Greatest element: \(x\) such that \(y \leq x\) for all \(y \in S\); often denoted by \(1\) or \(\top\) (top).

- Least element: \(x\) such that \(x \leq y\) for all \(y \in S\); often denoted by \(0\) or \(\bot\) (bottom).

- It is not necessary to have 1 or 0 in a partially ordered set.
  - e.g. \(S = \{ a, b, c, d \}\) and only \(a \leq b\) and \(c \leq d\).

- We can always add an artificial top or bottom to the set (if we need one).
Displaying Partially Ordered Sets

- Represented by an undirected graph
  - Nodes = elements of S
  - If $a \leq b$, a is shown below b in the picture
- If $a \leq b$, there is an edge $(a,b)$
  - But: transitive edges are typically not shown
- Example: $S = \{0,a,b,c,1\}$

```
0 \leq a \leq b \leq 1
0 \leq c \leq 1
```

Implicit
transitive
edges:

```
0 \leq b,
0 \leq 1, a \leq 1
```
Meet

• S – partially ordered set, $a \in S$, $b \in S$

• A meet of $a$ and $b$ is $c \in S$ such that
  - $c \leq a$ and $c \leq b$
  - For any $x$: $x \leq a$ and $x \leq b$ implies $x \leq c$
  - Also referred to as “the greatest lower bound of $a$ and $b”$
  - Typically denoted by $a \land b$

\[
\begin{align*}
a \land b &= a & a \land 0 &= 0 \\
a \land c &= 0 & a \land 1 &= a \\
b \land c &= 0 & b \land 1 &= b \\
b \land 0 &= 0 & \ldots
\end{align*}
\]
Join

- A **join** of $a$ and $b$ is $c \in S$ such that
  - $a \leq c$ and $b \leq c$
  - For any $x$: $a \leq x$ and $b \leq x$ implies $c \leq x$
  - Also referred to as “the least upper bound of $a$ and $b”
  - Typically denoted by $a \lor b$

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\end{array}
\quad
\begin{array}{c}
\text{1} \\
\text{0} \\
\text{a} \\
\text{b} \\
\text{c} \\
\end{array}
\]

\[
\begin{align*}
  a \lor b &= b & a \lor 0 &= a \\
  a \lor c &= 1 & a \lor 1 &= 1 \\
  b \lor c &= 1 & b \lor 1 &= 1 \\
  b \lor 0 &= b & \ldots
\end{align*}
\]
Lattices

• Any pair \((a,b)\) has either zero or one meets
  – Why can’t there be two meets?
  – Similarly for joins

\[
\begin{array}{c}
  a & \Lambda & b \\
  c & \Lambda & d
\end{array}
\]

\(a \Lambda b\) does not exist

“\(x \leq a\) and \(x \leq b\) implies \(x \leq \text{meet}\)” : NO!

• If every pair \((a,b)\) has is a meet and a join, the set is a lattice with operators \(\Lambda\) and \(\vee\)
  – If only a meet operator is defined: a meet semilattice

• Finite lattice: the underlying set is finite

• Finite-height lattice: any chain \(x < y < z < \ldots\) is finite
Cross-Product Lattice

- Given a lattice \(( L, \leq, \Lambda, V )\)
- Let \( L^n = L \times L \times \ldots \times L \) (elements are n-tuples)
- Partial order: \((a_1, \ldots, a_n) \leq (b_1, \ldots, b_n)\) iff \(a_i \leq b_i\) for all \(i\)
- Meet: \((a_1, \ldots, a_n) \Lambda (b_1, \ldots, b_n) = (a_1 \Lambda b_1, \ldots, a_n \Lambda b_n)\)
  - Same for join
- Cross-product lattice: \(( L^n, \leq, \Lambda, V )\)
- If \( L \) has a bottom element \(0\), \( L^n \) has a bottom element \((0, \ldots, 0)\)
- If \( L \) has a top element \(1\), \( L^n \) has a top element \((1, \ldots, 1)\)
- If \( L \) has finite height, so does \( L^n \)
So What?

• All of this is basic discrete math. What does it have to do with compile-time code analysis and code optimizations?
• For many analysis problems, program properties can be conveniently encoded as lattice elements.
• If $a \leq b$, in some sense the property encoded by $a$ is weaker (or stronger) than the one encoded by $b$.
  – Exactly what “weaker”/“stronger” means depends on the problem.
• We usually care only about “going in one direction” (down) in the lattice, so typically it is enough to have a meet semilattice.
The Most Basic Lattice

- Many dataflow analyses use a lattice \( L \) that is the power set \( \mathcal{P}(X) \) of some set \( X \)
  - \( \mathcal{P}(X) \) is the set of all subsets of \( X \)
  - A lattice element is a subset of \( X \)
  - Partial order \( \leq \) is the \( \supseteq \) relation
  - Meet is set union \( \cup \); join is set intersection \( \cap \)
  - \( 0 = X; 1 = \emptyset \)
Reaching Definitions and Live Variables

• Let D be the set of all definitions in the CFG

• Reaching definitions: the lattice L is $\mathcal{P}(D)$
  – The solution for every CFG node is a lattice element
    • $\text{IN}[n] \in \mathcal{P}(D)$ is the set of definitions reaching $n$
  – The complete solution is a map $\text{Nodes} \rightarrow L$
    • Actually, an element of the cross-product lattice $L|\text{Nodes}|$; basically, an n-tuple

• Let V be the set of all variables that are read anywhere in the CFG

• Live variables: the lattice L is $\mathcal{P}(V)$
  – The solution for every CFG node is a lattice element
    • $\text{OUT}[n] \in \mathcal{P}(V)$ is the set of variables live at n
  – The complete solution is a map $\text{Nodes} \rightarrow L$
The Role of Meet

• The partial order encodes some notion of strength for properties
  – if $x \leq y$, then $x$ is “less precise” than $y$

• Reaching Definitions: $x \leq y$ iff $x \supseteq y$
  – $x$ tells us that more things are possible, so $x$ is less precise than $y$
  – Extreme case: if $x = 0 = D$, this tells us that any definition may reach

• $x \wedge y$ is less precise than $x$ and $y$
  – greatest lower bound is the most precise lattice element that “describes” both $x$ and $y$
  – E.g., the union of two sets of reaching definitions is the smallest (most precise) way to describe both
    • Any superset of the union has redundancy in it
The Role of Meet (cont’d)

• Recall the Constant Propagation problem
  – At each CFG node \( n \), \( \text{IN}[n] \) is a map \( \text{Vars} \rightarrow \text{Values} \)
  – \( \text{Values} = \text{all possible constant values} \cup \{ \text{nac, undef} \} \)
  – \( \text{Values} \) is an infinite lattice with finite height
    • \( \text{nac} \leq \text{any constant value} \leq \text{undef} \)
    • two different constant values are not comparable

• Meet operation in \( \text{Values} \):
  – If \( c_1 = \text{undef} \), the result is \( c_2 \)
  – Else if \( c_2 = \text{undef} \), the result is \( c_1 \)
  – Else if \( c_1 = \text{nac} \) or \( c_2 = \text{nac} \), the result it \( \text{nac} \)
  – Else if \( c_1 \neq c_2 \), the result is \( \text{nac} \)
  – Else the result is \( c_1 \) (in this case we know that \( c_1 = c_2 \))

• Problem lattice \( \mathbf{L} \): cross-product \( \text{Values}^{\mid \text{Vars} \mid} \)
Transfer Functions

• A dataflow analysis defines a meet semilattice $L$ that encodes some program properties

• It also has to define the effects of program statements on these properties
  – A transfer function $f_n: L \rightarrow L$ is associated with each CFG node $n$
  – For forward problems: if the properties before the execution of $n$ were encoded by $x \in L$, the properties after the execution of $n$ are encoded by $f_n(x)$

• Reaching Definitions
  – $f_n(x) = (x \cap \text{PRES}[n]) \cup \text{GEN}[n]$
  – Expressed with meet and join: $f(x) = (x \lor a) \land b$
Function Space and Dataflow Framework

• Given: meet semilattice \((L, \leq, \Lambda, 1)\) with finite height
  – This is what we typically want as the part of the definition of the dataflow analysis

• A monotone functions space for \(L\) is a set \(F\) of functions \(f : L \rightarrow L\) such that
  – Each \(f\) is monotone: \(x \leq y\) implies \(f(x) \leq f(y)\)
  • This is equivalent to \(f(x \Lambda y) \leq f(x) \Lambda f(y)\)
  – \(F\) contains the identity function
  – \(F\) is closed under composition and meet: \(f \circ g\) and \(f \Lambda g\) are in \(F\)
    \[\text{Note: } (f \circ g)(x) = f(g(x))\text{ and } (f \Lambda g)(x) = f(x) \Lambda g(x)\]

• Dataflow framework: \((L, F)\)
  – Forward or backward; we will consider only forward
  – Framework instance \((G, M)\): \(G=(N, E)\) is a CFG; \(M: N \rightarrow F\)
    associates a transfer function \(f \in F\) with each node \(n \in N\)
Intraprocedural Dataflow Analysis

• Given: an intraprocedural CFG, a lattice L, and transfer functions
  – Plus a lattice element \( \eta \in L \) that describes the properties that hold at the entry node of the CFG

• The effects of one particular CFG path \( p=(n_0,n_1,\ldots,n_k) \) are

\[
  f_{n_k} (f_{n_{k-1}} (\ldots f_1 (f_0 (\eta))\ldots))
\]

  – i.e., \( f_p(\eta) \), where \( f_p \) is the composition of the transfer functions for nodes in the path
  – \( n_0 \) is the entry node of the CFG
Intraprocedural Dataflow Analysis

• Analysis goal: for each CFG node \( n \), compute a meet-over-all-paths solution

\[
\text{MOP}(n) = \bigwedge_{p \in \text{Paths}(n_0, n)} f_p(\eta)
\]

– \( \text{Paths}(n_0, n) \) the set of all paths from the entry node to \( n \) (the paths do not include \( n \))

• This solution “summarizes” all properties that could hold immediately before \( n \)
  – Many execution paths: “meet” ensures that we get the greatest lower bound of their effects

  • E.g., the \textbf{smallest} set of reachable definitions
The MOP Solution

• The MOP solution encodes everything that could potentially happen at run time
  – e.g., for Reaching Definitions: if there exists a run-time execution in which variable x is assigned at \( m \) and read at \( n \), set MOP(\( n \)) is guaranteed to contain the definition of x at \( m \)

• Problems for computing MOP(\( n \)):
  – Potentially infinite # paths due to loops
  – Even if there is a finite number of paths, there are too many of them: too expensive to compute MOP(\( n \)) by considering each path separately

• Finding the MOP solution is **undecidable** for general monotone dataflow frameworks
  – Or even just for the constant propagation problem
Approximating the MOP Solution

• A compromise: compute an approximation of the MOP solution

• A correct approximation: $S(n) \leq MOP(n)$
  – Recall that $\leq$ means “less precise”
  – e.g., for Reaching Definitions $IN[n] \supseteq MOP(n)$
  – “safe solution” = “correct solution”

• A precise approximation: $S(n)$ should be as close to $MOP(n)$ as possible
  – In the best case, $S(n)=MOP(n)$
Standard Approximation Algorithm

• Idea: define a system of equations and then solve it with fixed-point computation

\[ S(n) = \bigwedge_{m \in \text{Pred}(n)} f_m(S(m)) \]

• This system has the form \( S = F(S) \)
  – \( S: \text{Nodes} \to L \) is map from CFG nodes to lattice elements (\( S \) is in the cross-product lattice \( L^{\mid\text{Nodes}\mid} \))
  – \( F: (\text{Nodes} \to L) \to (\text{Nodes} \to L) \) is a function that computes the new solution from the old one, based on the node-level transfer functions \( f_n \)
Computing a Fixed Point

- Discrete math: if $f$ is a function, a **fixed point** of $f$ is a value $x$ such that $x = f(x)$
  - We want to compute a fixed point of $F$
  - Standard algorithm (fixed-point computation)

\[
S := [1,1,...,1] \\
\text{change} := \text{true} \\
\text{while} \ (\text{change}) \\
\quad \text{old}_S := S; \\
\quad S := F(S) \\
\quad \text{if} \ (S \neq \text{old}_S) \ \text{change} := \text{true} \\
\quad \text{else} \quad \text{change} := \text{false}
\]
Does This Really Work?

• Does not necessarily terminate
• Common case: finite-height lattice + monotone function space (as described earlier)
• In this case, the algorithm provably terminates with the greatest (maximum) fixed point MFP
  – Note: be careful with the difference between maximal (no one is > x) and maximum (x > everyone)
• MFP is a safe approximation of the MOP solution: \( MFP(n) \leq MOP(n) \)
  – For some categories of problems, the computed solution is the same as the MOP solution
    • e.g., for Reaching Definitions, but not for Constant Propagation
Outline of Proofs

- Termination with a fixed point
- Monotonicity: $1^n \geq F(1^n) \geq F^2(1^n) \geq F^3(1^n) \geq \ldots$
- Finite height for $L$ implies finite height for $L^n$, which gives us termination with $F^m(1^n) = F^{m+1}(1^n)$
  - $F^m(1^n)$ is a fixed point of $F$, and a solution to the system
- Is it the greatest (maximum) fixed point?
  - For any other fixed point $S$: $1^n \geq S$, $F(1^n) \geq F(S) = S$, …
    - By induction on $j$, $F^j(1^n) \geq S$
- Why is $\text{MOP} \geq \text{MFP}$?
  - For each CFG path $p=(n_0, n_1, \ldots, n_k)$, $f_p(\eta) \geq \text{MFP}$ for any successor of $n_k$
    - Proof by induction on the length of paths
Distributive Frameworks

- Each \( f \) is monotone: \( x \leq y \) implies \( f(x) \leq f(y) \)
  - This is equivalent to \( f(x \land y) \leq f(x) \land f(y) \)
- Distributive: \( f(x \land y) = f(x) \land f(y) \)
  - Each distributive function is also monotone
  - Examples: Reaching Defs, Live Variables, Available Expressions, Very Busy Expressions, Copy Propagation

- In this case, \( MFP = MOP \)
  - Proof outline: Since we already know that \( MOP \geq MFP \), enough to show that \( MFP \geq MOP \)
  - Show by induction on \( j \) that \( F^j(1^n) \geq MOP \)
  - Enough to show that \( F(MOP) = MOP \): that is, \( MOP(n) = \) meet of \( f_m(MOP(m)) \) over all predecessors \( m \) of \( n \)
  - By definition, \( MOP(m) \) is a meet over all paths leading to \( m \); \( f_m(\text{meet of paths}) = \text{meet}(f_m(\text{path})) \)
An Approximation: Flow-Insensitive Analysis

• Some problems are too complex/expensive to compute a solution specific to each CFG node
  – Typical example: pointer analysis (more later)
• Approximation: “pretend” that statements can execute in any order
  – Not only in the order defined by CFG paths
• Completely ignore all CFG edges – just consider the transfer functions at nodes
  – For technical reasons, make the functions “non-kill”: \( f(x) \leq x \) [e.g. as if KILL set was empty for Reaching Defs]
• Single solution (lattice element) for the entire CFG
• Naïve algo: start from 1 and apply the transfer functions in arbitrary order; get to a fixed point