Analysis of Software Artifacts

Dataflow Analysis: Examples and Correctness

Jonathan Aldrich

Outline

- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm
- Example Dataflow Analyses
  - Constant Propagation
  - Reaching Definitions
  - Live Variable Analysis
- Dataflow Analysis Correctness
  - Termination
  - Soundness

Worklist Dataflow Analysis Algorithm

worklist = new Set();
for all node indexes i do
    input[i] = $\perp_A$;
input[*entry*] = ι_A;
worklist.add(*all nodes*);

while (!worklist.isEmpty()) do
    i = worklist.pop();
    after = f_A(input[i], node(i));
    for all k $\in$ succ(i) do
        newinput = input[k] $\sqcup$ after
        if (!(newinput $\sqsubseteq$ input[k]))
            input[k] = newinput;
            worklist.add(k);

Worklist Dataflow Analysis Algorithm

worklist = new Set();
for all node indexes i do
    input[i] = $\bot_A$;
input[*entry*] = ι_A;
worklist.add(*all nodes*);

Ok to just add entry node if flow functions cannot return $\bot_A$ (examples will assume this)

while (!worklist.isEmpty()) do
    i = worklist.pop();
    after = f_A(input[i], node(i));
    for all k$\in$succ(i) do
        newinput = input[k] $\sqcup$ after
        if (!(newinput $\sqsubseteq$ input[k]))
            input[k] = newinput;
            worklist.add(k);

Worklist Dataflow Analysis Algorithm

worklist = new Set();
for all node indexes i do
    input[i] = $\perp_A$;
input[*entry*] = ι_A;
worklist.add(*all nodes*);

Ok to just add entry node if flow functions cannot return $\perp_A$ (examples will assume this)

while (!worklist.isEmpty()) do
    i = worklist.pop();
    after = f_A(input[i], node(i));
    for all k $\in$ succ(i) do
        newinput = input[k] $\sqcup$ after
        if (!(newinput $\sqsubseteq$ input[k]))
            input[k] = newinput;
            worklist.add(k);

Pop removes the most recently added element from the set (performance optimization)

Example of Worklist

$[a := 0]_1$

$[b := 0]_2$

while $[a < 2]_3$ do

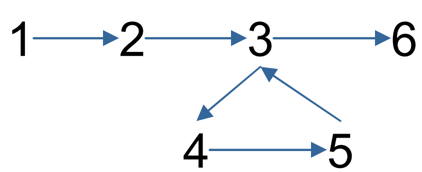
    $[b := a]_4$;

    $[a := a + 1]_5$;

$[a := 0]_6$

Position	Worklist	a	b
0	1	MZ	MZ

Control Flow Graph



Example of Worklist

$[a := 0]_1$

$[b := 0]_2$

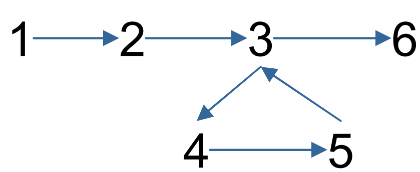
while $[a < 2]_3$ do

  $[b := a]_4$;

  $[a := a + 1]_5$;

$[a := 0]_6$

Control Flow Graph



Position	Worklist	a	b
0	1	MZ	MZ
1	2	Z	MZ
2	3	Z	Z
3	4,6	Z	Z
4	5,6	Z	Z
5	3,6	MZ	Z
3	4,6	MZ	Z
4	5,6	MZ	MZ
5	3,6	MZ	MZ
3	4,6	MZ	MZ
4	6	MZ	MZ
6		Z	MZ

Quick Quiz

Show how the worklist algorithm given in class operates on the program given, by filling in the table below.

```
1: x := 0
2: y := 1
3: if (z == 0)
4:        x := x + y
5: else y := y – 1
6: w := y
```

Position	Worklist	x	y	w
0				

Worklist Algorithm Performance

```
worklist = new Set();
for all node indexes i do
    input[i] = ⊥A;
input[entry] = ιA;
worklist.add(all nodes);

while (!worklist.isEmpty()) do
    i = worklist.pop();
    after = fA(input[i], node(i));
    for all k∈succ(i) do
        newinput = input[k] ⊔ after
        if (!(newinput ⊑ input[k]))
            input[k] = newinput;
            worklist.add(k);
```

- How many times might a node get added to the worklist?
  - The node's input must increase each time
  - The number of increases is bound by the height *h* of the lattice
- How many times do statements execute?
  - $h*n$ in total: we may run it *h* times for each node *n*
  - but we must propagate along all successor edges; these statements execute $h*e$ times
- Assume statement cost is *c*
  - Then performance is $O(h*e*c)$
  - Often *h, e,* and *c* are bounded by n. So we get $O(n^3)$
  - Good enough to run on a function, but not on the whole program

Outline

- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm
- Example Dataflow Analyses
  - Constant Propagation
  - Reaching Definitions
  - Live Variable Analysis
- Dataflow Analysis Correctness
  - Termination
  - Soundness

Constant Propagation

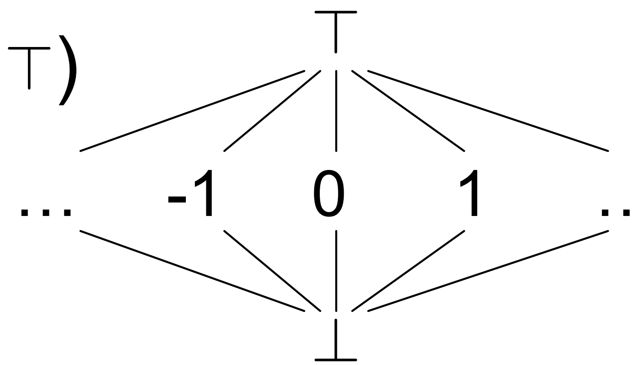
- Goal: determine which variables hold a constant value:

```
x := 3;
y := x+7;
if b
    then z := x+2
    else z := y-5;
w := z-2
```

- What is w?
  - Useful for optimization, error checking
  - Zero analysis is a special case

Constant Propagation Definition

- Constant lattice (L_C, $\sqsubseteq_C$, $\sqcup_C$, $\bot$, $\top$)
  - L_C = **Integer** $\lceil$ { $\bot$, $\top$ }
  - $\forall n \in$ **Integer** : $\bot \sqsubseteq_C n$ && $n \sqsubseteq_C \top$



- Constant propagation lattice
  - Tuple lattice formed from above lattice
  - See notes on zero analysis for details
- Abstraction function:
  - $\alpha_C(n) = n$
  - $\alpha_{CP}(\eta)$ = { x $\mapsto \alpha_C(\eta(\text{x}))$ | x $\in$ **Var** }
- Initial data:
  - ι_{CP} = { x $\mapsto \top$ | x $\in$ **Var** }

Constant Propagation Definition

- $f_{CP}(\sigma, [x := y]) = [x \mapsto \sigma(y)]\ \sigma$
- $f_{CP}(\sigma, [x := n]) = [x \mapsto n]\sigma$
- $f_{CP}(\sigma, [x := y\ op\ z]) = [x \mapsto (\sigma(y)\ op_{\top}\ \sigma(z))]\ \sigma$
  - $n\ op_{\top}\ m = n\ op\ m$
  - $n\ op_{\top}\ \top = \top$
  - $\top\ op_{\top}\ m = \top$
  - *Note: we could define for $\perp$ too, but we won't actually ever see $\perp$ during analysis*
- $f_{CP}(\sigma,$ /* any other */$) = \sigma$

Constant Propagation Example

$[x := 3]_1;$
$[y := x+7]_2;$
if $[b]_3$
  then $[z := x+2]_4$
  else $[z := y-5]_5;$
$[w := z-2]_6$

Position	Worklist	x	y	z	w

Constant Propagation Example

$[x := 3]_1$;
$[y := x+7]_2$;
if $[b]_3$
  then $[z := x+2]_4$
  else $[z := y-5]_5$;
$[w := z-2]_6$

Position	Worklist	x	y	z	w
0	1	$\top$	$\top$	$\top$	$\top$
1	2	3	$\top$	$\top$	$\top$
2	3	3	10	$\top$	$\top$
3	4,5	3	10	$\top$	$\top$
4	6,5	3	10	5	$\top$
6	5	3	10	5	3
5	6	3	10	5	$\top$
6		3	10	5	3

Constant Propagation Example

$[x := 3]_1;$
$[y := x+7]_2;$
$\text{if } [b]_3$
$\quad \text{then } [z := x+\mathbf{1}]_4$
$\quad \text{else } [z := y-5]_5;$
$[w := z-2]_6$

Position	Worklist	x	y	z	w
0	1	$\top$	$\top$	$\top$	$\top$
1	2	3	$\top$	$\top$	$\top$
2	3	3	10	$\top$	$\top$
3	4,5	3	10	$\top$	$\top$

Constant Propagation Example

$[x := 3]_1;$
$[y := x+7]_2;$
$\text{if } [b]_3$
$\quad \text{then } [z := x+\mathbf{1}]_4$
$\quad \text{else } [z := y-5]_5;$
$[w := z-2]_6$

Position	**Worklist**	**x**	**y**	**z**	**w**
0	1	$\top$	$\top$	$\top$	$\top$
1	2	3	$\top$	$\top$	$\top$
2	3	3	10	$\top$	$\top$
3	4,5	3	10	$\top$	$\top$
4	6,5	3	10	4	$\top$
6	5	3	10	4	2
5	6	3	10	5	$\top$
6		3	10	$\top$	$\top$

Loss of Precision

if $[x = 0]_1$

then $[y := 1]_2$;

else $[y := x]_3$;

$[z := 10/y]_4$

Position	Worklist	x	y	z
0	1	MZ	MZ	MZ
1	2,3	MZ	MZ	MZ
2	4,3	MZ	**NZ**	MZ
4	3	MZ	NZ	NZ
3	4	MZ	**MZ**	MZ
4		MZ	**MZ**	NZ

Branch Sensitivity for Zero Analysis

- Existing flow functions
  - $f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)]\ \sigma$
  - $f_{ZA}(\sigma, [x := n])$ = if n==0
    then $[x \mapsto \mathrm{Z}]\sigma$
    else $[x \mapsto \mathrm{NZ}]\sigma$
  - $f_{ZA}(\sigma, [x := y\ \mathit{op}\ z]) = [x \mapsto \mathrm{MZ}]\ \sigma$
  - $f_{ZA}(\sigma,$ /* *any other* */$) = \sigma$

Branch Sensitivity for Zero Analysis

- Existing flow functions
  - $f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)]\ \sigma$
  - $f_{ZA}(\sigma, [x := n]) = \text{if } n{=}{=}0$
    then $[x \mapsto \text{Z}]\sigma$
    else $[x \mapsto \text{NZ}]\sigma$
  - $f_{ZA}(\sigma, [x := y \text{ } op \text{ } z]) = [x \mapsto \text{MZ}]\ \sigma$
  - $f_{ZA}(\sigma, \text{/* } \textit{any other} \text{ */}) = \sigma$
- Propagate different info on branches
  - $f_{ZA}^{T}(\sigma, [x = 0]) = [x \mapsto \text{Z}]\ \sigma$
  - $f_{ZA}^{F}(\sigma, [x = 0]) = [x \mapsto \text{NZ}]\ \sigma$
  - Slightly more general:
    - $f_{ZA}^{T}(\sigma, [x = y]) = [x \mapsto \sigma(y)]\ \sigma$
    - $f_{ZA}^{F}(\sigma, [x = y]) = [x \mapsto \neg\sigma(y)]\ \sigma$
      - Assume $\neg$Z=NZ; $\neg$NZ=Z; $\neg$MZ=MZ

Precision Regained

Worklist simplified to the statement level

if $[x = 0]_1$
    then $[y := 1]_2$;
    else $[y := x]_3$;
$[z := 10/y]_4$

Position	Worklist	x	y	z
0	1	MZ	MZ	MZ
1^T	2,3	**Z**	MZ	MZ
1^F	2,3	**NZ**	MZ	MZ
2 (use 1^T)	4,3	Z	**NZ**	MZ
4	3	Z	NZ	NZ
3 (use 1^F)	4	NZ	**NZ**	MZ
4		**MZ**	**NZ**	NZ

Outline

- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm
- Example Dataflow Analyses
  - Constant Propagation
  - Reaching Definitions
  - Live Variable Analysis
- Dataflow Analysis Correctness
  - Termination
  - Soundness

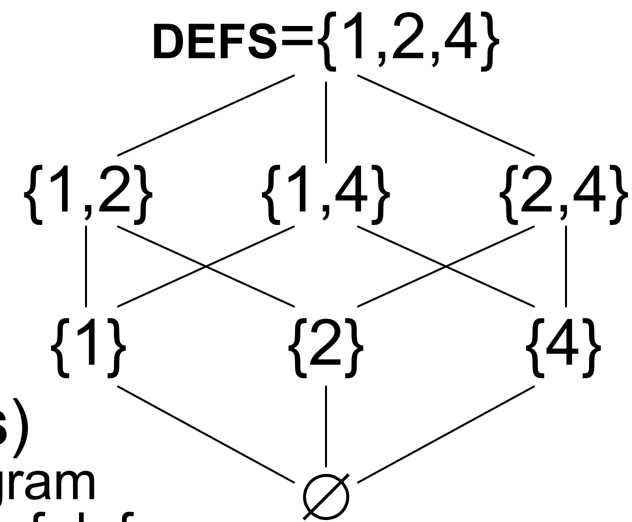
Reaching Definitions Analysis

- Goal: determine which is the most recent assignment to a variable that precedes its use:

$[y := x]_1;$
$[z := 1]_2;$
while $[y>1]_3$ do
    $[z := z * y]_4;$
    $[y := y - 1]_5;$
$[y := 0]_6;$

- Example: definitions 1 and 5 reach the use of y at 4
- Applications
  - Simpler version of constant propagation, zero analysis, etc.
    - Just look at the reaching definitions for constants
  - If definitions reaching use include “undefined” sentinal, then we may be using an undefined variable

Reaching Definitions



- Set Lattice ($\mathbb{P}^{\textbf{DEFS}}$, $\sqsubseteq_{RD}$, $\sqcup_{RD}$, $\varnothing$, **DEFS**)
  - **DEFS** is the set of definitions in the program
  - Each element of the lattice is a subset of defs
    - $\mathbb{P}^{\textbf{DEFS}}$ is the powerset of **DEFS**, i.e. the set of all subsets of **DEFS**
  - Approximation
    - A definition d may reach program point P if d is in the lattice at P
    - We call this a *may analysis*
  - $x \sqsubseteq_{RD} y$ iff $x \subseteq y$
  - $x \sqcup_{RD} y = x \cup y$
    - This is a direct consequence of the definition of $\sqsubseteq_{RD}$
  - Most precise element $\bot=\varnothing$ (no reaching definitions)
  - Least precise element $\top$=**DEFS** (all definitions reach)

Reaching Definitions

- Initially assume dummy assignments
  - Represents passed values for parameters
  - Represents uninitialized for non-parameters
  - $\iota_{RD} = \{ x_0 \mid x \in \mathbf{Var} \}$
- Flow functions
  - $f_{RD}(\sigma, [x := \ldots]_k)$

    $= \sigma - \{ x_m \mid x_m \in \mathbf{DEFS}(x)\} \cup \{ x_k \}$
    - Kills (removes from set) all other definitions of x
    - Generates (adds to set) the current definition x_k
    - Kill/Gen pattern true in many analyses with set lattices
  - $f_{RD}(\sigma,$ */* any other */*$) = \sigma$

Reaching Definitions Example

$[y := x]_1$;

$[z := 1]_2$;

while $[y>1]_3$ do

    $[z := z * y]_4$;

    $[y := y - 1]_5$;

$[y := 0]_6$;

Position	Worklist	Lattice Element

Reaching Definitions Example

$[y := x]_1$;

$[z := 1]_2$;

while $[y>1]_3$ do

    $[z := z * y]_4$;

    $[y := y - 1]_5$;

$[y := 0]_6$;

Position	Worklist	Lattice Element
0	1	$\{x_0, y_0, z_0\}$
1	2	$\{x_0, y_1, z_0\}$
2	3	$\{x_0, y_1, z_2\}$
3	4,6	$\{x_0, y_1, z_2\}$
4	5,6	$\{x_0, y_1, z_4\}$
5	3,6	$\{x_0, y_5, z_4\}$
3	4,6	$\{x_0, y_1, y_5, z_2, z_4\}$
4	5,6	$\{x_0, y_1, y_5, z_4\}$
5	6	$\{x_0, y_5, z_4\}$
6		$\{x_0, y_6, z_2, z_4\}$

Outline

- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm
- Example Dataflow Analyses
  - Constant Propagation
  - Reaching Definitions
  - Live Variable Analysis
- Dataflow Analysis Correctness
  - Termination
  - Soundness

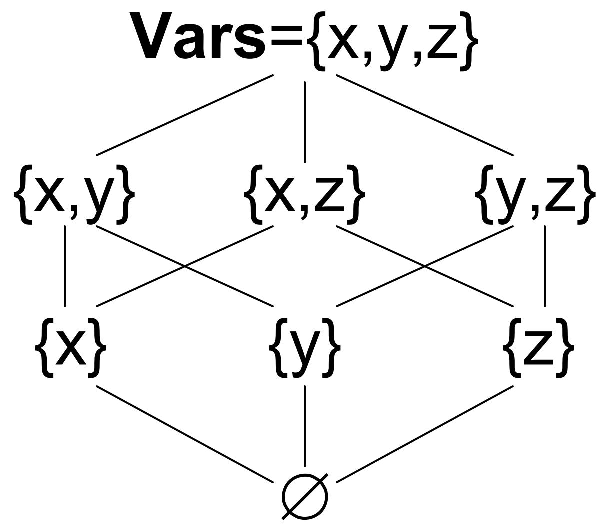
Live Variables Analysis

- Goal: determine which variables may be used again before they are redefined (i.e. are live) at the current program point:

$[y := x]_1$;
$[z := 1]_2$;
while $[y>1]_3$ do
    $[z := z * y]_4$;
    $[y := y - 1]_5$;
$[y := 0]_6$;

- Example: after statement 1, y is live, but x and z are not
- Optimization applications
  - If a variable is not live after it is defined, can remove the definition statement (e.g. 6 in the example)

Live Variables Definition



- Set Lattice ($\mathbb{P}^{\textbf{Vars}}$, $\sqsubseteq_{LV}$, $\sqcup_{LV}$, $\emptyset$, **Vars**)
  - **Vars** is the set of variables in the program
  - Each element of the lattice is a subset of **Vars**
    - $\mathbb{P}^{\textbf{Vars}}$ is the powerset of **Vars**, i.e. the set of all subsets of **Vars**
  - $x \sqsubseteq_{LV} y$ iff $x \subseteq y$
  - $x \sqcup_{LV} y = x \cup y$
  - Most precise element $\bot = \emptyset$ (no live variables)
  - Least precise element $\top$=**DEFS** (all variables live)

Live Variables Definition

- Live Variables is a *backwards* analysis
  - To figure out if a variable is live, you have to look at the future execution of the program
- Will x be used before it is redefined?
  - When x is defined, assume it is not live
  - When x is used, assume it is live
  - Propagate lattice elements as usual, except backwards
- Initially assume return value is live
  - $\iota_{LV} = \{ x \}$ where x is the variable returned from the function

Flow Function Practice

- Write flow functions for Live Variable analysis:
  - $f_{\mathrm{LV}}(\sigma, [x := e]_{\mathrm{k}}) =$
  - $f_{\mathrm{LV}}(\sigma,$ /* *any other* */$) =$

Flow Function Practice

- Write flow functions for Live Variable analysis:
  - $f_{\mathrm{LV}}(\sigma, [x := e]_k) = (\sigma - \{ x \}) \cup \mathrm{vars}(e)$
    - Kills (removes from set) the variable x
    - Generates (adds to set) the variables in e
    - Note: must kill first then generate (what if $e = x$?)
  - $f_{\mathrm{LV}}(\sigma, [e]_k) = \sigma \cup \mathrm{vars}(e)$
  - $f_{\mathrm{LV}}(\sigma, /^{*}\ \textit{any other}\ ^{*}/) = \sigma$

Worklist Practice

Show how the worklist algorithm given in class operates on the program given, by filling in the table below.

[y := x]$_1$;

[z := 1]$_2$;

while [y>1]$_3$ do

    [z := z * y]$_4$;

    [y := y – 1]$_5$;

[y := 0]$_6$;

return z;

Position	Worklist	Lattice Value

Live Variables Example

$[y := x]_1$;

$[z := 1]_2$;

while $[y>1]_3$ do

    $[z := z * y]_4$;

    $[y := y - 1]_5$;

$[y := 0]_6$;

return z;

Position	Worklist	Lattice Element
exit	6	{z}
6	3	{z}
3	5,2	{y,z}
5	4,2	{y,z}
4	3,2	{y,z}
3	2	{y,z}
2	1	{y}
1		{x}

Outline

- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm
- Example Dataflow Analyses
  - Constant Propagation
  - Reaching Definitions
  - Live Variable Analysis
- Dataflow Analysis Correctness
  - Termination
  - Soundness

What does Correctness Mean?

- Intuition
  - Analysis will eventually terminate at a fixed point
  - At a fixed point, analysis results are a *sound abstraction* of *program execution*
  - *program execution* must be formally defined
  - *abstraction function* relates program execution to data flow lattice elements
  - *sound* means truth $\sqsubseteq$ analysis results
    - also called *conservative* or *safe*

Termination

- Intuition
  - Dataflow information for a statement gets less precise every time we visit the statement
  - Information can only get less precise as many times as the lattice is high
  - When information stops changing, we stop

- Key property: Monotonic flow functions
  - f is *monotone* iff $\sigma \sqsubseteq \sigma'$ implies $f(\sigma) \sqsubseteq f(\sigma')$

Nonterminating Analysis

(bad) idea: Track set of values for each variable

$[x := 0]_1$

while $[x < y]_2$ do

   $[x := x + 1]_3$;

$[x := 0]_4$;

Iter	Position	x	y
1	0	$\mathbb{Z}$	$\mathbb{Z}$
2	1	{0}	$\mathbb{Z}$
3	2	{0}	$\mathbb{Z}$
4	3	{1}	$\mathbb{Z}$
5	2	{0,1}	$\mathbb{Z}$
6	3	{1,2}	$\mathbb{Z}$
7	2	{0,1,2}	$\mathbb{Z}$
8	3	{1,2,3}	$\mathbb{Z}$
9	2	{0,1,2,3}	$\mathbb{Z}$
10	3	{1,2,3,4}	$\mathbb{Z}$
…			

Moral: make your lattices finite height!

Dataflow Analysis Termination

- Theorem: If the flow function of a dataflow analysis is monotone, and the height of the lattice is finite, then the analysis will terminate

Dataflow Analysis Termination

- Theorem: If the flow function of a dataflow analysis is monotone, and the height of the lattice is finite, then the analysis will terminate
- Lemma: Each time a node is added to the worklist, a dataflow value has increased (and no dataflow value has decreased)
  - Proof outline: by induction
    - Base case: The dataflow value for every statement is $\perp$. This is the lowest point in the lattice. Thus the first time the value changes, it will increase.
    - Inductive case: Assume the last application of the dataflow function mapped σ to $f(\sigma)$. By assumption $\sigma \sqsubseteq \sigma'$. By monotonicity $f(\sigma) \sqsubseteq f(\sigma')$. Thus the output value increased.
  - Will not affect others because only the flow value for the current statement is set.

Dataflow Analysis Termination

- Theorem: If the flow function of a dataflow analysis is monotone, and the height of the lattice is finite, then the analysis will terminate
- Lemma: Each time a node is added to the worklist, a dataflow value has increased (and no dataflow value has decreased)
  - Proof outline: by induction
    - Base case: The dataflow value for every statement is $\perp$. This is the lowest point in the lattice. Thus the first time the value changes, it will increase.
    - Inductive case: Assume the last application of the dataflow function mapped σ to $f(\sigma)$. By assumption $\sigma \sqsubseteq \sigma'$. By monotonicity $f(\sigma) \sqsubseteq f(\sigma')$. Thus the output value increased.
  - Will not affect others because only the flow value for the current statement is set.
- Proof outline for theorem:
  - Each time a node is added to the worklist, the dataflow value was raised in the lattice for one statement.
  - If there are *n* statements in the program and the height of the lattice is *h*, this can happen at most *n*h* times.
  - An inspection of the worklist algorithm shows that a finite number of steps occurs between applications of flow functions, and that when the values stop changing the algorithm terminates.

Outline

- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm
- Example Dataflow Analyses
  - Constant Propagation
  - Reaching Definitions
  - Live Variable Analysis
- Dataflow Analysis Correctness
  - Termination
  - Soundness

Dataflow Analysis Soundness

- Intuition
  - The result of dataflow analysis is a conservative approximation of all possible run time states

- Definition
  - A dataflow analysis is sound if, for all programs P, for all inputs I, for all times T in P's execution on input I,
  - If P is at statement S at time T, with memory η, and the analysis returned abstract state σ for S,
  - then $\alpha(\eta) \sqsubseteq \sigma$

Local Soundness

$$
\begin{tikzcd}
\alpha_{DF}(\eta_i) \sqsubseteq \sigma_i \arrow[rr, "f_{DF}(\sigma_i,\, S)"] & & \sigma_o \sqsupseteq \alpha_{DF}(\eta_o) \\
\eta_i \arrow[u, dashed, "\alpha_{DF}"'] \arrow[rr, "S \rightarrow"] & & \eta_o \arrow[u, dashed, "\alpha_{DF}"']
\end{tikzcd}
$$

- Local correctness condition for dataflow analysis
  - If applying a transfer function to statement S and input information σ_i yields output information σ_o,
  - Then for all program states η_i such that $\alpha(\eta_i) \sqsubseteq \sigma_i$ and executing S in state η_i yields state η_o,
  - We must have $\alpha(\eta_o) \sqsubseteq \sigma_o$

Intuitively, translating from concrete to abstract and applying the flow function will safely approximate ($\sqsupseteq$) taking a step in the trace and translating from concrete to abstract

Finding Errors with Local Soundness

- Consider the **incorrect** flow function:
  $f_{ZA}(\sigma, [x := y\ op\ z]) =$
  if $\sigma[y]=Z \mid\mid \sigma[z]=Z$
  then $[x \mapsto Z]\sigma$
  else $[x \mapsto MZ]\sigma$

$$\begin{tikzcd}
\alpha_{DF}(\eta_i) \sqsubseteq \sigma_i \arrow[r, "f_{DF}(\sigma_i, S)"] & \sigma_o \sqsupseteq \alpha_{DF}(\eta_o) \\
\eta_i \arrow[u, dashed, "\alpha_{DF}"] \arrow[r, "S \rightarrow"] & \eta_o \arrow[u, dashed, "\alpha_{DF}"']
\end{tikzcd}$$

- Challenge: find an example where local soundness fails

Finding Errors with Local Soundness

- Consider the **incorrect** flow function:

  $f_{ZA}(\sigma, [x := y \text{ op } z]) =$

  if $\sigma[y]$=Z || $\sigma[z]$=Z
  then $[x \mapsto Z]\sigma$
  else $[x \mapsto MZ]\sigma$

$$\begin{tikzcd}
\alpha_{DF}(\eta_i) \sqsubseteq \sigma_i \arrow[r, "f_{DF}(\sigma_i, S)"] & \sigma_o \sqsupseteq \alpha_{DF}(\eta_o) \\
\eta_i \arrow[u, dashed, "\alpha_{DF}"] \arrow[r, "S \rightarrow"] & \eta_o \arrow[u, dashed, "\alpha_{DF}"']
\end{tikzcd}$$

- Local Soundness failure:
  - Let σ_i = [], S = "x := 3+0"
  - Consider η_i = []. As required, $\alpha_{DF}(\eta_i) = [] \sqsubseteq \sigma_i$
  - Now $\sigma_o = f_{DF}(\sigma_i, S) = [x \mapsto Z]$
  - And $\eta_o = S(\eta_i) = [x \mapsto 3]$
  - So $\alpha_{DF}(\eta_o) = \alpha_{DF}([x \mapsto 3]) = [x \mapsto NZ]$
  - BUT $\alpha_{DF}(\eta_o) \not\sqsubseteq \sigma_o$ because Z $\not\sqsubseteq$ NZ, so local soundness is violated

Proving Correctness

- Consider a Zero Analysis flow function
  - $f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)]\ \sigma$

- Monotonicity
  - Assume $\sigma' \sqsubseteq \sigma$.
  - $f_{ZA}(\sigma, [x := y])$ changes only the value for x
  - Therefore for all variables $z \neq x$ we have $f_{ZA}(\sigma', [x := y])(z) \sqsubseteq f_{ZA}(\sigma, [x := y])(z)$
  - Since $f_{ZA}(\sigma, [x := y])(x) = \sigma(y)$ and $\sigma'(y) \sqsubseteq \sigma(y)$ we have $f_{ZA}(\sigma', [x := y])(x) \sqsubseteq f_{ZA}(\sigma, [x := y])(x)$
  - Thus $f_{ZA}(\sigma', [x := y]) \sqsubseteq f_{ZA}(\sigma, [x := y])$

Proving Correctness

- Consider a Zero Analysis flow function
  - $f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)]\ \sigma$

- Local Soundness
  - Assume $\alpha(\eta_i) \sqsubseteq \sigma_i$.
  - By Java's semantics $\eta_o = [x \mapsto \eta_i(y)]\ \eta_i$
  - By the flow function, $\sigma_o = [x \mapsto \sigma_i(y)]\ \sigma_i$
  - Since both maps changed only in their x value, for all variables $z \neq x$ we have $\alpha_{DF}(\eta_o)(z) \sqsubseteq \sigma_o(z)$
  - Since $\alpha(\eta_i)(y) \sqsubseteq \sigma_i(y)$, $\alpha(\eta_o)(x) = \alpha(\eta_i)(y)$, and $\sigma_o(y) = \sigma_i(y)$, we also know that $\alpha(\eta_o)(x) \sqsubseteq \sigma_o(x)$
  - Thus $\alpha(\eta_o) \sqsubseteq \sigma_o$

Global Soundness

- Intuition
  - We begin with initial dataflow facts ι that safely approximate ($\sqsupseteq$) all initial stores η_0
  - By local soundness, each transfer function when given safe input information yields safe output information
  - By induction, any fixed point of the analysis is sound

Why care about Soundness?

- Analysis Producers
  - Writing analyses is hard
    - People make mistakes all the time
    - Need to know how to ***think*** about correctness
    - When the analysis gets tricky, it's useful to prove it correct formally

- Analysis Consumers
  - Sound analysis provides guarantees
    - Optimizations won't break the program
    - Finds all defects of a certain sort
  - Decision making
    - Knowledge of soundness techniques lets you ask the right questions about a tool you are considering
    - Soundness affects where you invest QA resources
      - Focus testing efforts on areas where you don't have a sound analysis!