

Program Representations and Bug Finders

17-654/17-754

Analysis of Software Artifacts

Jonathan Aldrich



Analysis of Software Artifacts -
Spring 2008

Representing Programs



- To analyze software automatically, we must be able to represent it precisely
- Some representations
 - Source code
 - **Abstract syntax trees**
 - Control flow graph
 - Bytecode
 - Assembly code
 - Binary code

Analysis of Software Artifacts -
Spring 2008

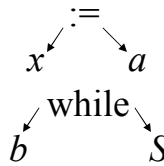
2

Abstract Syntax Trees

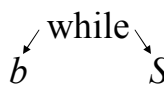


- A tree representation of source code
- Based on the language grammar
- One type of node for each production

• $S ::= x := a$



• $S ::= \text{while } b \text{ do } S$



Parsing: Source to AST

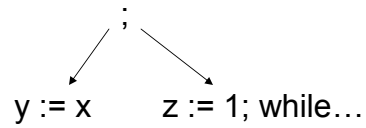


- Parsing process (top down)
 1. Determine the top-level production to use
 2. Create an AST element for that production
 3. Determine what text corresponds to each child of the AST element
 4. Recursively parse each child
- Algorithms have been studied in detail
 - For this course you only need the intuition
 - Details covered in compiler courses

Parsing Example



```
y := x;  
z := 1;  
while y > 1 do  
  z := z * y;  
  y := y - 1
```

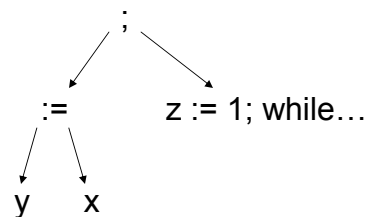


- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$

Parsing Example



```
y := x;  
z := 1;  
while y > 1 do  
  z := z * y;  
  y := y - 1
```

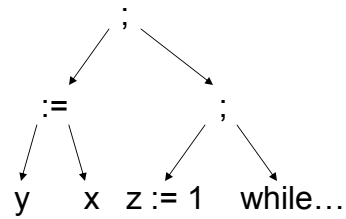


- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$

Parsing Example



```
y := x;  
z := 1;  
while y>1 do  
  z := z * y;  
  y := y - 1
```

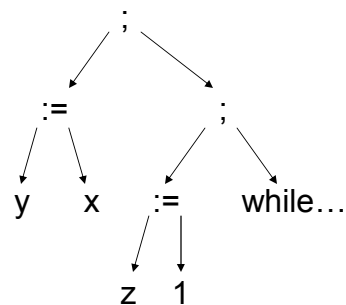


- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$

Parsing Example



```
y := x;  
z := 1;  
while y>1 do  
  z := z * y;  
  y := y - 1
```

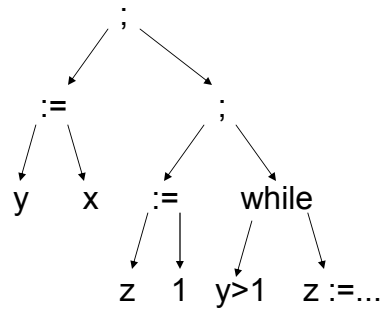


- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$

Parsing Example



```
y := x;  
z := 1;  
while y>1 do  
  z := z * y;  
  y := y - 1
```

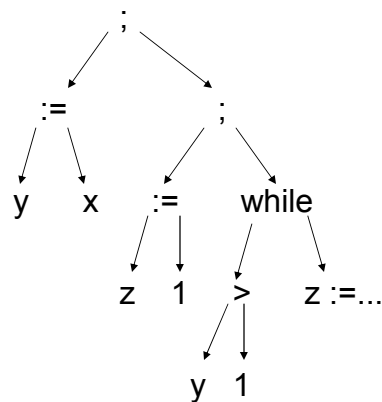


- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$

Parsing Example



```
y := x;  
z := 1;  
while y>1 do  
  z := z * y;  
  y := y - 1
```



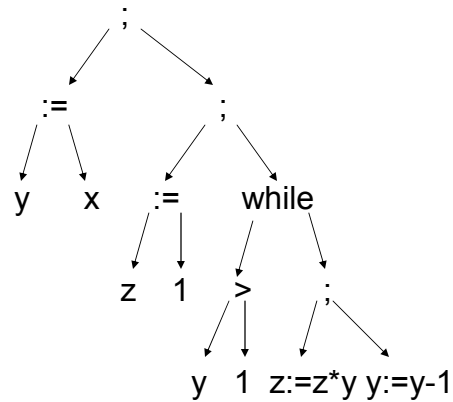
- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$

Parsing Example



```

y := x;
z := 1;
while y > 1 do
  z := z * y;
  y := y - 1
  
```



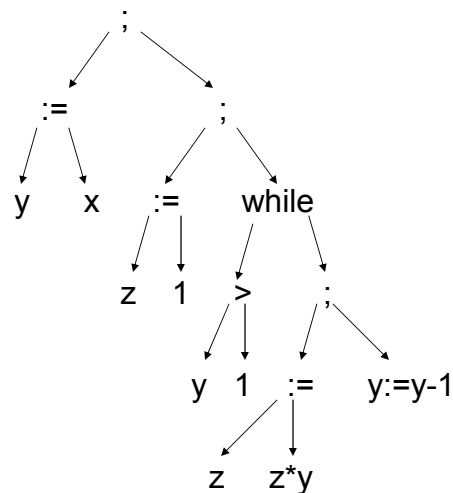
- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$

Parsing Example



```

y := x;
z := 1;
while y > 1 do
  z := z * y;
  y := y - 1
  
```



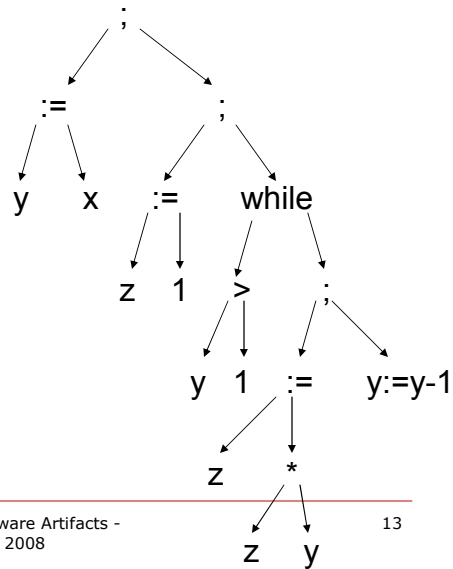
- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$

Parsing Example



```
y := x;  
z := 1;  
while y > 1 do  
  z := z * y;  
  y := y - 1
```

- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$



Analysis of Software Artifacts -
Spring 2008

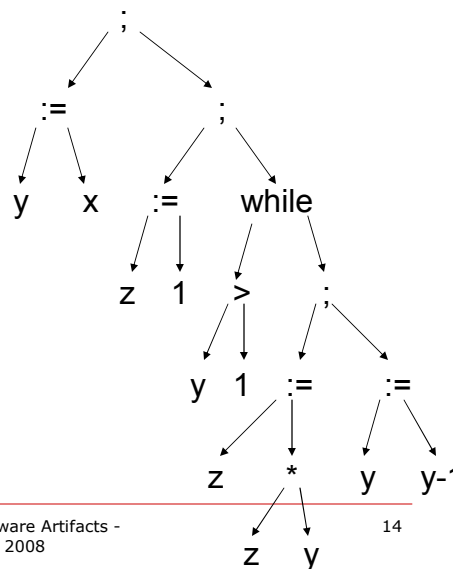
13

Parsing Example



```
y := x;  
z := 1;  
while y > 1 do  
  z := z * y;  
  y := y - 1
```

- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$



Analysis of Software Artifacts -
Spring 2008

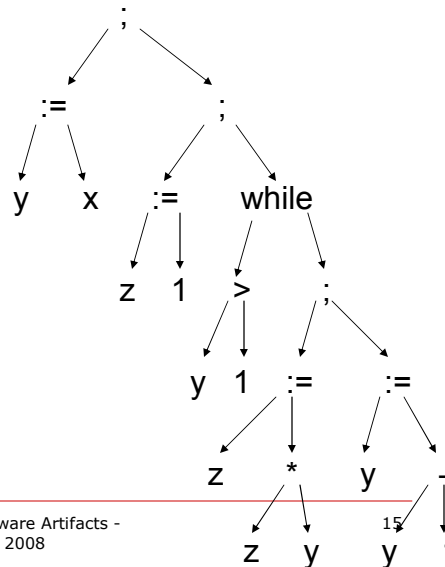
14

Parsing Example



```

y := x;
z := 1;
while y > 1 do
  z := z * y;
  y := y - 1
    
```



- Top-level production?
 - $S_1; S_2$
- What are the parts?
 - $y := x$
 - $z := 1; \text{while } \dots$

Analysis of Software Artifacts -
Spring 2008

WHILE ASTs in Java



- Java data structures mirror grammar

- $S ::= x := a$
 - | skip
 - | $S_1; S_2$
 - | if b then S_1 else S_2
 - | while b do S

```

class AST { ... }
class Stmt extends AST { ... }
class Assign extends Stmt {
  Var var;
  AExpr expr;
}
class Skip extends Stmt { }
class Seq extends Stmt {
  Stmt left;
  Stmt right;
}
class If extends Stmt {
  BExpr cond;
  Stmt thenStmt;
  Stmt elseStmt;
}
class While extends Stmt {
  BExpr cond;
  Stmt body;
}
    
```

Analysis of Software Artifacts -
Spring 2008

16

Matching AST against Bug Patterns



- **AST Walker Analysis**
 - Walk the AST, looking for nodes of a particular type
 - Check the immediate neighborhood of the node for a bug pattern
 - Warn if the node matches the pattern
- **Common architecture based on Visitors**
 - class Visitor has a visitX method for each type of AST node X
 - Default Visitor code just descends the AST, visiting each node
 - To find a bug in AST element of type X, override visitX
- **Semantic grep**
 - Like grep, looking for simple patterns
 - Unlike grep, consider not just names, but semantic structure of AST
 - Makes the analysis more precise

Example: Shifting by more than 31 bits



```
class BadShiftAnalysis extends Visitor
  visitShiftExpression(ShiftExpression e) {
    if (type of e's left operand is int)
      if (e's right operand is a constant)
        if (value of constant < 0 or > 31)
          warn("Shifting by less than 0 or more
than 31 is meaningless")

    super.visitShiftExpression(e);
  }
```

Example: String concatenation in a loop



```
class StringConcatLoopAnalysis extends Visitor
    private int loopLevel = 0;

    visitStringConcat(StringConcat e) {
        if (loopLevel > 0)
            warn("Performance issue: String concatenation in loop (use
StringBuffer instead)")
            super.visitStringConcat(e);    // visits AST children
        }

    visitWhile(While e) {
        loopLevel++;
        super.visitWhile(e);                // visits AST children
        loopLevel--;
    }
    // similar for other looping constructs
```

Example Tool: FindBugs



- Origin: research project at U. Maryland
 - Now freely available as open source
 - Standalone tool, plugins for Eclipse, etc.
- Checks over 250 “bug patterns”
 - Over 100 correctness bugs
 - Many style issues as well
 - Includes the two examples just shown
- Focus on simple, local checks
 - Similar to the patterns we’ve seen
 - But checks bytecode, not AST
 - Harder to write, but more efficient and doesn’t require source
- <http://findbugs.sourceforge.net/>

Example FindBugs Bug Patterns



- Correct equals()
- Use of ==
- Closing streams
- Illegal casts
- Null pointer dereference
- Infinite loops
- Encapsulation problems
- Inconsistent synchronization
- Inefficient String use
- Dead store to variable

FindBugs Experiences



- Useful for learning idioms of Java
 - Rules about libraries and interfaces
 - e.g. equals()
- Customization is important
 - Many warnings may be irrelevant, others may be important – depends on domain
 - e.g. embedded system vs. web application
- Useful for pointing out things to examine
 - Not all are real defects
 - Turn off false positive warnings for future analyses on codebase

More Dataflow Analyses: Reaching Definitions and Live Variables

17-654/17-754

Analysis of Software Artifacts

Jonathan Aldrich



Analysis of Software Artifacts -
Spring 2008

Reaching Definitions Analysis



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

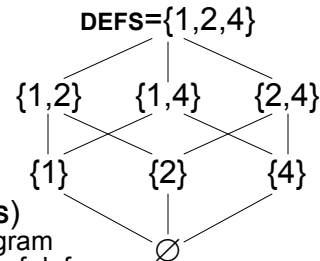
```
[y := x]1;  
[z := 1]2;  
while [y > 1]3 do  
  [z := z * y]4;  
  [y := y - 1]5;  
[y := 0]6;
```

- 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” sentinel, then we may be using an undefined variable

Analysis of Software Artifacts -
Spring 2008

24

Reaching Definitions



- Set Lattice (\mathbb{P}^{DEFS} , \sqsubseteq_{RD} , \sqcup_{RD} , \emptyset , DEFS)
 - DEFS is the set of definitions in the program
 - Each element of the lattice is a subset of defs
 - \mathbb{P}^{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*
 - $x \sqsubseteq_{\text{RD}} y$ iff $x \subseteq y$
 - $x \sqcup_{\text{RD}} y = x \cup y$
 - This is a direct consequence of the definition of \sqsubseteq_{RD}
 - Most precise element $\perp = \emptyset$ (no reaching definitions)
 - Least precise element $\top = \text{DEFS}$ (all definitions reach)

Reaching Definitions



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

$$= \sigma - \{x_m \mid x_m \in \text{DEFS}(x)\} \cup \{x_k\}$$
 - 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
 - $f_{\text{RD}}(\sigma, /* \text{any other} */) = \sigma$

Reaching Definitions Example



	Position	Worklist	Lattice Element
<code>[y := x]₁;</code>	0	1	{x ₀ , y ₀ , z ₀ }
<code>[z := 1]₂;</code>	1	2	{x ₀ , y ₁ , z ₀ }
<code>while [y > 1]₃ do</code>	2	3	{x ₀ , y ₁ , z ₂ }
<code>[z := z * y]₄;</code>	3	4,6	{x ₀ , y ₁ , z ₂ }
<code>[z := z * y]₄;</code>	4	5,6	{x ₀ , y ₁ , z ₄ }
<code>[y := y - 1]₅;</code>	5	3,6	{x ₀ , y ₅ , z ₄ }
<code>[y := y - 1]₅;</code>	3	4,6	{x ₀ , y ₁ , y ₅ , z ₂ , z ₄ }
<code>[y := 0]₆;</code>	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 }

Live Variables Analysis

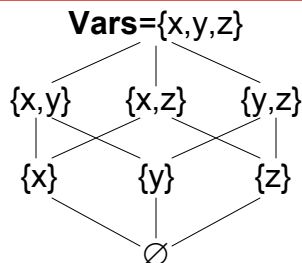


- Goal: determine which variables may be used again (i.e. are live) at the current program point:

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

- $f_{LV}(\sigma, [x := e]_k) =$

- $f_{LV}(\sigma, /* \textit{any other} */) =$

Flow Function Practice



- Write flow functions for Live Variable analysis:

- $f_{LV}(\sigma, [x := e]_k) = (\sigma - \{x\}) \cup \{\text{vars}(e)\}$

- Kills (removes from set) the variable x
- Generates (adds to set) the variables in e
- Note: must kill first then generate (what if $e = x$?)

- $f_{LV}(\sigma, /* \textit{any other} */) = \sigma$

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;
```

Position	Worklist	Lattice Value
exit		

Live Variables Example



[y := x] ₁ ;	Position	Worklist	Lattice Element
[z := 1] ₂ ;	exit	6	{z}
while [y > 1] ₃ do	6	3	{z}
[z := z * y] ₄ ;	3	5,2	{y,z}
[y := y - 1] ₅ ;	5	4,2	{y,z}
[y := 0] ₆ ;	4	3,2	{y,z}
	3	2	{y,z}
	2	1	{y}
	1		{x}

Dataflow Analysis Correctness

17-654/17-754

Analysis of Software Artifacts

Jonathan Aldrich



Analysis of Software Artifacts -
Spring 2008

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*

Analysis of Software Artifacts -
Spring 2008

36

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



	Iter	Position	x	y
(bad) idea: Track set of values for each variable	1	0	Z	Z
	2	1	{0}	Z
	3	2	{0}	Z
	4	3	{1}	Z
	5	2	{0,1}	Z
$[x := 0]_1$	6	3	{1,2}	Z
while $[x < y]_2$ do	7	2	{0,1,2}	Z
$[x := x + 1]_3$;	8	3	{1,2,3}	Z
	9	2	{0,1,2,3}	Z
$[x := 0]_4$;	10	3	{1,2,3,4}	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
- Lemma: Each application of the flow function will increase some dataflow value (and not affect others)
 - 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 application of a flow function raises the dataflow value 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 $n \cdot h$ 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.

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 $\alpha(\eta) \sqsubseteq \sigma$

Proving Soundness



- Formally define analysis
 - Including abstraction function
 - We already know how
- Formalize *trace semantics*
- Prove *local soundness* for flow functions
- Apply *global soundness theorem*

Execution Traces



- Sequence of $\langle pp, mem \rangle$ pairs
 - pp is a program point
 - Just after statement pp
 - mem is the state of variables in memory

$[y := x]_1;$

$[z := 1]_2;$

while $[y > 1]_3$ do

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

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

$[y := 0]_6;$

<u>pp</u>	<u>x</u>	<u>y</u>	<u>z</u>
0	2	0	0
1	2	2	0
2	2	2	1
3	2	2	1
4	2	2	2
5	2	1	2
3	2	1	2
6	2	0	2

Execution Traces



- Sequence of $\langle pp, mem \rangle$ pairs
 - pp is a program point
 - Just after statement pp
 - mem is the state of variables in memory

pp	x	y	z
0	1	0	0
1	1	1	0
2	1	1	1
3	1	1	1
6	1	0	1

```
[y := x]1;  
[z := 1]2;  
while [y > 1]3 do  
  [z := z * y]4;  
  [y := y - 1]5;  
[y := 0]6;
```

Execution Traces



- Sequence of $\langle pp, mem \rangle$ pairs
 - pp is a program point
 - Just after statement pp
 - mem is the state of variables in memory

pp	x	y	z
0	3	0	0
1	3	3	0
2	3	3	1
3	3	3	1
4	3	3	3
5	3	2	3
3	3	2	3
4	3	2	6
5	3	1	6
3	3	1	6
6	3	0	6

```
[y := x]1;  
[z := 1]2;  
while [y > 1]3 do  
  [z := z * y]4;  
  [y := y - 1]5;  
[y := 0]6;
```

Execution Traces



- Sequence of $\langle pp, mem \rangle$ pairs
 - pp is a program point
 - Just after statement pp
 - mem is the state of variables in memory

pp x y z

Repeat for all possible initial values of x, y, z !

```
[y := x]1;  
[z := 1]2;  
while [y > 1]3 do  
  [z := z * y]4;  
  [y := y - 1]5;  
[y := 0]6;
```

WHILE Traces, Formally



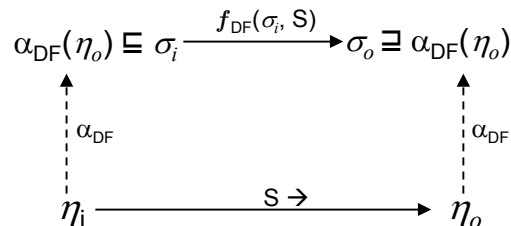
- A trace for program S_1 and initial state η_0 is either:
 - A finite sequence $(\eta_0, S_1), \dots, (\eta_n, \text{skip})$
where $(\eta_i, S_i) \rightarrow (\eta_{i+1}, S_{i+1})$ for $0 \leq i < n$
 - An infinite sequence $(\eta_0, S_1), \dots, (\eta_i, S_i), \dots$
where $(\eta_i, S_i) \rightarrow (\eta_{i+1}, S_{i+1})$ for $i \geq 0$
- Slight notational simplification
 - We will abbreviate $(\eta_0, S_0), \dots, (\eta_n, S_n)$
as $(\eta_0, \text{first}(S_0)), \dots, (\eta_n, \text{first}(S_n))$
 - *first* is the label of the first statement in S
 - Uses program counter labels instead of complete programs

What does Correctness Mean?



- Intuition
 - At a fixed point, analysis results are a *sound abstraction of program execution*
- Soundness condition
 - When data flow analysis reaches a fixed point F , then for all traces T and all times t in each trace, $\alpha(T(t)) \sqsubseteq \sigma_{pp(T(t))}$ where $\sigma_{pp(T(t))}$ is the analysis results at $pp(T(t))$
- Constant propagation
 - For trace on last slide with $t=10$
 - $\alpha_{CP}(T(10)) = \{x \mapsto 3, y \mapsto 0, z \mapsto 6\}$
 - $\sigma_{pp(T(t))} = \sigma_6 = \{x \mapsto \top, y \mapsto 0, z \mapsto \top\}$
 - $\{x \mapsto 3, y \mapsto 0, z \mapsto 6\} \sqsubseteq_{CP} \{x \mapsto \top, y \mapsto 0, z \mapsto \top\}$
 - Because $3 \sqsubseteq_C \top$ and $0 \sqsubseteq_C 0$ and $6 \sqsubseteq_C \top$ in the CP lattice
 - To prove soundness, repeat for all times in all traces

Local Soundness



- 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:
 $f_{ZA}(\sigma, [x := y \text{ op } z]) =$
if $\sigma[y]=Z \parallel \sigma[z]=Z$
then $[x \mapsto Z]\sigma$ else $[x \mapsto MZ]\sigma$
- Local Soundness fails!
 - Consider $\eta_i = []$, $S_i = [x := 3+0]_k$
 - $\sigma_i = \alpha_{DF}(\eta_i) = \alpha_{DF}([]) = []$
 - $\sigma_{i+1} = f_{DF}(\sigma_i, first(S_i)) = [x \mapsto Z]$
 - $\alpha_{DF}(\eta_{i+1}) = \alpha_{DF}([x \mapsto 3]) = [x \mapsto NZ]$
 - $\alpha_{DF}(\eta_{i+1}) \not\sqsubseteq \sigma_{i+1}$ because $Z \not\sqsubseteq NZ$

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

Global Soundness



- Theorem (Global Soundness)
 - Assume that $\forall T \in \text{traces}(S) \alpha_{DF}(\eta_0) \sqsubseteq \iota$ and that analysis DF is monotone and locally sound with respect to α_{DF}
 - Then for any fixed point DF_{fix} of DF on program S , $\forall T \in \text{traces}(S) \forall t \in \text{times}(T)$ we have $\alpha_{DF}(\eta_t) \sqsubseteq DF_{fix}(pp(T(t)))$
- Proof outline: For each trace T we do induction on t
 - Induction hypothesis: $\alpha_{DF}(\eta_t) \sqsubseteq DF_{fix}(pp(T(t)))$
 - Base case: $t=0$
 - By assumption $\alpha_{DF}(\eta_0) \sqsubseteq \iota = DF_{fix}(pp(\eta_0))$
 - Inductive case: time t and statement S_t
 - *Simplifying assumption: straight-line control flow*
 - By induction hypothesis we have $\alpha_{DF}(\eta_{t-1}) \sqsubseteq DF_{fix}(pp(T(t-1)))$
 - By monotonicity of DF we have:
 $f_{DF}(\alpha_{DF}(\eta_{t-1}), S_t) \sqsubseteq f_{DF}(DF_{fix}(pp(T(t-1))), S_t)$
 - By local soundness we have $\alpha_{DF}(\eta_t) \sqsubseteq f_{DF}(\alpha_{DF}(\eta_{t-1}), S_t)$
 - By transitivity we get $\alpha_{DF}(\eta_t) \sqsubseteq f_{DF}(DF_{fix}(pp(T(t-1))), S_t)$
 - But $f_{DF}(DF_{fix}(pp(T(t-1))), S_t) = DF_{fix}(pp(T(t)))$ because it's a fixed point
 - So we have $\alpha_{DF}(\eta_t) \sqsubseteq DF_{fix}(pp(T(t)))$

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!