

Dataflow Analysis, Continued

17-654/17-765
Analysis of Software Artifacts
Jonathan Aldrich



Analysis of Software Artifacts -
Spring 2006

Loss of Precision

Worklist simplified to the statement level



	Position	Worklist	x	y	z
if [x = 0] ₁	0	1	MZ	MZ	MZ
then [y := 1] ₂ ;	1	2,3	MZ	MZ	MZ
else [y := x] ₃ ;	2	4,3	MZ	NZ	MZ
	4	3	MZ	NZ	NZ
[z := 10/y] ₄	3	4	MZ	MZ	MZ
	4		MZ	MZ	NZ

Analysis of Software Artifacts -
Spring 2006

2

Zero Analysis Flow Functions



- Existing flow functions
 - $f_{ZA}(\sigma, [x]_k) = [t_k \mapsto \sigma(x)] \sigma$
 - $f_{ZA}(\sigma, [n]_k) = \text{if } n==0 \text{ then } [t_k \mapsto Z] \sigma$
 else $[t_k \mapsto NZ] \sigma$
 - $f_{ZA}(\sigma, [x := [\dots]_n]_k) = [x \mapsto \sigma(t_n)] \sigma$
 - $f_{ZA}(\sigma, [[\dots]_n \text{ op } [\dots]_m]_k) = [t_k \mapsto MZ] \sigma$
 - $f_{ZA}(\sigma, /* \text{ any other } */) = \sigma$
- Propagate different info on branches
 - $f_{ZA}^T(\sigma, [[x]_n = [0]_m]_k) = [x \mapsto Z] \sigma$
 - $f_{ZA}^F(\sigma, [[x]_n = [0]_m]_k) = [x \mapsto NZ] \sigma$
 - Slightly better:
 - $f_{ZA}^T(\sigma, [[x]_n = [\dots]_m]_k) = [x \mapsto \sigma(t_m)] \sigma$
 - $f_{ZA}^F(\sigma, [[x]_n = [\dots]_m]_k) = [x \mapsto \neg \sigma(t_m)] \sigma$
 - Assume $\neg Z = NZ$; $\neg NZ = Z$; $\neg MZ = MZ$

Precision Regained

Worklist simplified to the statement level



	Position	Worklist	x	y	z
if $[x = 0]_1$	0	1	MZ	MZ	MZ
then $[y := 1]_2$;	1^T	2,3	Z	MZ	MZ
else $[y := x]_3$;	1^F	2,3	NZ	MZ	MZ
[z := 10/y] ₄	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

Reaching Definitions Analysis

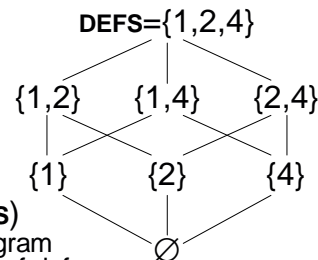


- Goal: determine which is the most recent assignment to a variable that precedes its use:

```
[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

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
 - $\iota_{RD} = \{x_0 \mid x \in \mathbf{Var}\}$
- Flow functions
 - $f_{RD}(\sigma, [x := [\dots]_n]_k)$
 $= \sigma - \{x_m \mid x_m \in \mathbf{DEFS}\} \cup \{x_k\}$
 - 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
 - $f_{RD}(\sigma, /* \text{any other} */) = \sigma$

Reaching Definitions Example

Worklist simplified to the statement level



	Position	Worklist	Lattice Element
$[y := x]_1;$	0	1	$\{x_0, y_0, z_0\}$
$[z := 1]_2;$	1	2	$\{x_0, y_1, z_0\}$
while $[y > 1]_3$ do	2	3	$\{x_0, y_1, z_1\}$
$[z := z * y]_4;$	3	4,6	$\{x_0, y_1, z_1\}$
$[z := z * y]_4;$	4	5,6	$\{x_0, y_1, z_4\}$
$[y := y - 1]_5;$	5	3,6	$\{x_0, y_5, z_4\}$
$[y := y - 1]_5;$	3	4,6	$\{x_0, y_1, y_5, z_1, z_4\}$
$[z := z * y]_4;$	4	5,6	$\{x_0, y_1, y_5, z_4\}$
$[y := 0]_6;$	5	6	$\{x_0, y_5, z_4\}$
	6		$\{x_0, y_6, z_1, z_4\}$

Live Variables Analysis

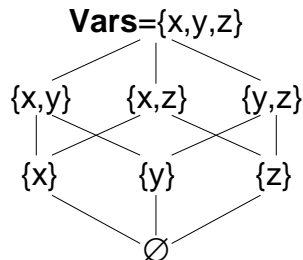


- Goal: determine which variables may be used again (i.e. are live) at the current program point:

```
[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})
 - **Vars** is the set of variables in the program
 - Each element of the lattice is a subset of **Vars**
 - $\mathbb{P}^{\mathbf{Vars}}$ 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 $\perp = \emptyset$ (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 functions
 - $f_{LV}(\sigma, [x]_k) = \sigma \cup \{x\}$
 - Generates (adds to set) the variable x
 - $f_{LV}(\sigma, [x := [\dots]_n]_k) = \sigma - \{x\}$
 - Kills (removes from set) the variable x
 - $f_{LV}(\sigma, /* \text{any other} */) = \sigma$

Live Variables Example

Worklist simplified to the statement level



	Position	Worklist	Lattice Element
$[y := x]_1;$	exit	6	$\{z\}$
$[z := 1]_2;$	6	3	$\{z\}$
while $[y > 1]_3$ do	3	5,2	$\{y, z\}$
$[z := z * y]_4;$	5 _{asn}	5 _{exp} , 2	$\{z\}$
$[y := y - 1]_5;$	5 _{exp}	4 _{asn} , 2	$\{y, z\}$
	4 _{asn}	4 _{exp} , 2	$\{y\}$
	4 _{exp}	3, 2	$\{y, z\}$
$[y := 0]_6;$	3	2	$\{y, z\}$
	2	1 _{asn}	$\{y\}$
	1 _{asn}	1 _{exp}	\emptyset
	1 _{exp}		$\{x\}$

Dataflow Analysis Correctness

17-654/17-765
Analysis of Software Artifacts
Jonathan Aldrich



Analysis of Software Artifacts -
Spring 2006

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 2006

14

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!

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 $f(\sigma) \sqsubseteq f(\sigma')$

Monotonicity for Zero Analysis



- f is *monotone* iff $\sigma \sqsubseteq \sigma'$ implies $f(\sigma) \sqsubseteq f(\sigma')$
- Case $f_{ZA}(\sigma, [x]_k) = [t_k \mapsto \sigma(x)] \sigma$
 - Assume $\sigma \sqsubseteq_{ZA} \sigma'$
 - Then $\sigma(x) \sqsubseteq_Z \sigma'(x)$
 - So $[t_k \mapsto \sigma(x)] \sigma \sqsubseteq_{ZA} [t_k \mapsto \sigma'(x)] \sigma'$
- Case $f_{ZA}(\sigma, [n]_k) = \text{if } n == 0$
 then $[t_k \mapsto Z] \sigma$ else $[t_k \mapsto NZ] \sigma$
 - Assume $\sigma \sqsubseteq_{ZA} \sigma'$
 - Subcase $n == 0$: $[t_k \mapsto Z] \sigma \sqsubseteq_{ZA} [t_k \mapsto Z] \sigma'$
 - Subcase $n != 0$: $[t_k \mapsto NZ] \sigma \sqsubseteq_{ZA} [t_k \mapsto NZ] \sigma'$

Monotonicity for Zero Analysis



- f is *monotone* iff $\sigma \sqsubseteq \sigma'$ implies $f(\sigma) \sqsubseteq f(\sigma')$
- Case $f_{ZA}(\sigma, [x := [\dots]_n]_k) = [x \mapsto \sigma(t_n)] \sigma$
 - Assume $\sigma \sqsubseteq_{ZA} \sigma'$
 - Then $\sigma(t_n) \sqsubseteq_Z \sigma'(t_n)$
 - So $[x \mapsto \sigma(t_n)] \sigma \sqsubseteq_{ZA} [x \mapsto \sigma'(t_n)] \sigma'$
- Case $f_{ZA}(\sigma, [[\dots]_n \text{ op } [\dots]_m]_k) = [t_k \mapsto MZ] \sigma$
 - Assume $\sigma \sqsubseteq_{ZA} \sigma'$
 - Therefore $[t_k \mapsto MZ] \sigma \sqsubseteq_{ZA} [t_k \mapsto MZ] \sigma'$
- Case $f_{ZA}(\sigma, /* \text{ any other } */) = \sigma$
 - Follows directly from assumption that $\sigma \sqsubseteq_{ZA} \sigma'$

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.

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

	<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
$[y := x]_1;$	5	2	1	2
$[z := 1]_2;$	3	2	1	2
while $[y > 1]_3$ do	6	2	0	2
$[z := z * y]_4;$				
$[y := y - 1]_5;$				
$[y := 0]_6;$				

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

	<u>pp</u>	<u>x</u>	<u>y</u>	<u>z</u>
	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

	<u>pp</u>	<u>x</u>	<u>y</u>	<u>z</u>
	0	3	0	0
	1	3	3	0
	2	3	3	1
	3	3	3	1
	4	3	3	3
$[y := x]_1;$	5	3	2	3
$[z := 1]_2;$	3	3	2	3
while $[y > 1]_3$ do	4	3	2	6
$[z := z * y]_4;$	5	3	1	6
$[y := y - 1]_5;$	3	3	1	6
$[y := 0]_6;$	6	3	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

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

Analysis of Software Artifacts -

25

What does Correctness Mean?

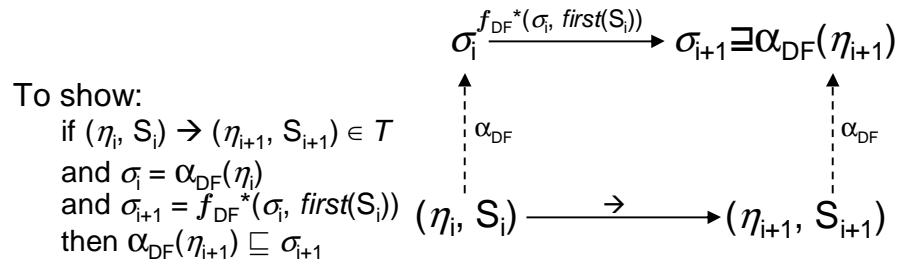


- Intuition
 - At a fixed point, analysis results are a *sound abstraction* of *program execution*
- Soundness condition
 - When data flow analysis reaches a fixed point F , then for all traces T and all times t in each trace, $\alpha(T(t)) \sqsubseteq \sigma_{pp(T(t))}$ where $\sigma_{pp(T(t))}$ is 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

Analysis of Software Artifacts -
Spring 2006

26

Local Soundness



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