

Interprocedural Analysis

17-654/17-754

Analysis of Software Artifacts

Jonathan Aldrich



Interprocedural Analysis Strategies



- Make default assumptions
- Assume and check annotations
- Build Interprocedural CFG
- Compute Summaries
 - All contexts one by one
 - Each context on demand
 - All context at once

Making Default Assumptions



- Assumptions
 - Starting dataflow value for all parameters
 - Dataflow value for result
 - Starting and ending information for globals if you're tracking them (*most analyses don't*)
- Verification
 - Initial info: starting value for parameters
 - Verify result \sqsubseteq assumption_{result}
 - Ending value for result obeys assumption
 - Verify arg \sqsubseteq assumption_{arg}
 - Actual arguments obey assumptions of formal parameter
 - Similar extension to globals if you're tracking them

Making Default Assumptions



- Example: Zero Analysis
 - Default: \top (MZ) for arguments and results
 - Benefit: actual arguments and actual result always obey assumption
 - Cost: very conservative for arguments
 - Will report false positive errors
 - Globals: easiest solution is not to track them
 - Could also track but assume \top at boundaries

Example: Default Assumptions



<pre>int divByX(int x) { [result := 10/x]₁; }</pre>	<ul style="list-style-type: none">Analyze divByX												
	<table><tr><td>p</td><td>x</td><td>result</td></tr><tr><td>0</td><td>MZ</td><td>MZ</td></tr><tr><td>1</td><td>MZ</td><td>NZ</td></tr></table>	p	x	result	0	MZ	MZ	1	MZ	NZ			
p	x	result											
0	MZ	MZ											
1	MZ	NZ											
<pre>void caller() { [x := 5]₁; [y := divByX(x)]₂; }</pre>	<ul style="list-style-type: none">Warning: div by zero at 1Verify $\sigma[\text{result}] \sqsubseteq \text{MZ}$												
	<ul style="list-style-type: none">Analyze caller												
	<table><tr><td>p</td><td>x</td><td>y</td></tr><tr><td>0</td><td>MZ</td><td>MZ</td></tr><tr><td>1</td><td>NZ</td><td>MZ</td></tr><tr><td>2</td><td>NZ</td><td>MZ</td></tr></table>	p	x	y	0	MZ	MZ	1	NZ	MZ	2	NZ	MZ
p	x	y											
0	MZ	MZ											
1	NZ	MZ											
2	NZ	MZ											
	<ul style="list-style-type: none">Verify $\sigma[x] \sqsubseteq \text{MZ}$Note that div by zero can't happen!												

Optimistic Assumption: NZ



<pre>int divByX(int x) { [result := 10/x]₁; }</pre>	<ul style="list-style-type: none">Analyze divByX												
	<table><tr><td>p</td><td>x</td><td>result</td></tr><tr><td>0</td><td>NZ</td><td>MZ</td></tr><tr><td>1</td><td>NZ</td><td>NZ</td></tr></table>	p	x	result	0	NZ	MZ	1	NZ	NZ			
p	x	result											
0	NZ	MZ											
1	NZ	NZ											
<pre>void caller() { [x := 5]₁; [y := divByX(x)]₂; }</pre>	<ul style="list-style-type: none">No warningVerify $\sigma[\text{result}] \sqsubseteq \text{NZ}$												
	<ul style="list-style-type: none">Analyze caller												
	<table><tr><td>p</td><td>x</td><td>y</td></tr><tr><td>0</td><td>MZ</td><td>MZ</td></tr><tr><td>1</td><td>NZ</td><td>MZ</td></tr><tr><td>2</td><td>NZ</td><td>NZ</td></tr></table>	p	x	y	0	MZ	MZ	1	NZ	MZ	2	NZ	NZ
p	x	y											
0	MZ	MZ											
1	NZ	MZ											
2	NZ	NZ											
	<ul style="list-style-type: none">Verify $\sigma[x] \sqsubseteq \text{NZ}$												

Optimistic Assumption: NZ



```
int double(int x) {
  [result := 2*x]1;
}

void caller() {
  [x := 0]1;
  [y := double(x)]2;
}
```

- Analyze double
 - p x result
 - 0 NZ MZ
 - 1 NZ NZ
- No warning
- Verify $\sigma[\text{result}] \sqsubseteq \text{NZ}$
- Analyze caller
 - p x y
 - 0 MZ MZ
 - 1 Z MZ
 - 2 Z NZ
- Verify $\sigma[x] \sqsubseteq \text{NZ}$ fails!
- False positive—this code is OK

Assume and Check Annotations



- Annotations
 - Starting dataflow value for all parameters
 - Dataflow value for result
- Verification
 - Initial info: starting value for parameters
 - Verify $\text{result} \sqsubseteq \text{annotation}_{\text{result}}$
 - Ending value for result obeys annotation
 - Verify $\text{arg} \sqsubseteq \text{annotation}_{\text{arg}}$
 - Actual arguments obey annotations on formal parameter

Assumption Example



```

@NZ int divByX(@NZ int x) {
  [result := 10/x]1;
}

void caller() {
  [x := 5]1;
  [y := divByX(x)]2;
}

```

- Analyze divByX

p	x	result
0	NZ	MZ
1	NZ	NZ

 - Verify $\sigma[\text{result}] \in \text{NZ}$
- Analyze caller

p	x	y
0	MZ	MZ
1	NZ	MZ
2	NZ	NZ

 - Verify $\sigma[x] \in \text{NZ}$

Assumption Example



```

@MZ int double(@MZ int x) {
  [result := 2*x]1;
}

void caller() {
  [x := 5]1;
  [y := double(x)]2;
  [z := 10/y]3;
}

```

- Analyze divByX

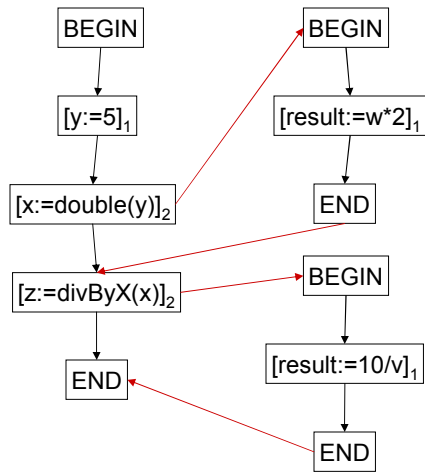
p	x	result
0	MZ	MZ
1	MZ	MZ

 - Verify $\sigma[\text{result}] \in \text{MZ}$
- Analyze caller

p	x	y	z
0	MZ	MZ	MZ
1	NZ	MZ	MZ
2	NZ	MZ	MZ
3	NZ	MZ	MZ

 - Verify $\sigma[x] \in \text{MZ}$
 - Warning: possible div by zero
 - False positive!

Interprocedural CFG Intuition

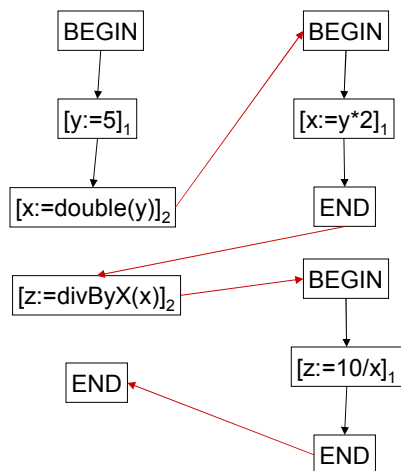


29 February 2008

Analysis of Software Artifacts
© 2008 Jonathan Aldrich

11

Interprocedural CFG Intuition



29 February 2008

Analysis of Software Artifacts
© 2008 Jonathan Aldrich

12

Interprocedural CFG Example



<code>int double(int x) {</code>	p	x	y	z
<code>[y := 2*x]₆;</code>	0	MZ	MZ	MZ
<code>}</code>	1	NZ	MZ	MZ
	6	NZ	NZ	MZ
<code>void caller() {</code>	2	NZ	NZ	MZ
<code>[x := 5]₁;</code>	3	NZ	NZ	NZ
<code>[y := double(x)]₂;</code>	4	Z	NZ	NZ
<code>[z := 10/y]₃;</code>	5	MZ	MZ	MZ
<code>[x := 0]₄;</code>	5	MZ	MZ	MZ
<code>[y := double(x)]₅;</code>				
<code>}</code>				
	2	MZ	MZ	MZ
	3	MZ	MZ	NZ

- No div by zero
- Must revisit node 2 because result of double changed
- Divide by zero warning
- False positive!

Context Sensitive Summaries



- Intuition
 - Interprocedural CFG loses too much precision when a function is called with different argument dataflow lattice elements
 - Simple annotations have same issue
 - (but same Summary technique works there)
- Summaries
 - Maps from input dataflow information to output dataflow information
 - *Context sensitive*: different results for different calls
 - When function is called, apply the map!

Generating Context Sensitive Summaries



- **Brute force**
 - Analyze the function once for each possible input lattice element
 - Problem: way too many lattice elements—would take too long
- **On demand**
 - Analyze the function once for each actual input lattice element it is called with
 - Much better—but can still be impractical for large programs with precise lattices
- **Abstract summaries**
 - Symbolically represent function's effect on input lattice element
 - Example: PREFIX's technique
 - The state of the art in interprocedural analysis

On Demand Summaries



```
/* Summary
 * Case x:NZ -> result:NZ
 */
int double(int x) {
  [result := 2*x]1;
}
void caller() {
  [x := 5]1;
  [y := double(x)]2;
  [z := 10/y]3;
  [x := 0]4;
  [y := double(x)]5;
}
```

p	x	y	z
0	MZ	MZ	MZ
1	NZ	MZ	MZ
2	NZ	NZ	MZ

Compute summary of double for x:NZ

p	x	result
0	NZ	MZ
1	NZ	NZ

On Demand Summaries



```

/* Summary
 * Case x:NZ -> result:NZ
 * Case x:Z -> result:Z
 */
int double(int x) {
  [result := 2*x]₁;
}

void caller() {
  [x := 5]₁;
  [y := double(x)]₂;
  [z := 10/y]₃;
  [x := 0]₄;
  [y := double(x)]₅;
}

```

	p	x	y	z
	0	MZ	MZ	MZ
	1	NZ	MZ	MZ
	2	NZ	NZ	MZ
	3	NZ	NZ	NZ
	4	Z	NZ	NZ
	5	Z	Z	NZ

		Compute summary of double for x:Z		
	p	x	result	
	0	Z	MZ	
	1	Z	Z	

29 February 2008

Analysis of Software Artifacts
© 2008 Jonathan Aldrich

17

Context Sensitive Annotations



```

@Case("x:NZ -> result:NZ")
@Case("x:Z -> result:Z")
int double(int x) {
  [result := 2*x]₁;
}

void caller() {
  [x := 5]₁;
  [y := double(x)]₂;
  [z := 10/y]₃;
  [x := 0]₄;
  [y := double(x)]₅;
}

```

	Verify annotation @Case("x:Z -> result:Z")		
	p	x	result
	0	Z	MZ
	1	Z	Z

	Verify annotation @Case("x:NZ -> result:NZ")		
	p	x	result
	0	NZ	MZ
	1	NZ	NZ

	Verify client			
	p	x	y	z
	0	MZ	MZ	MZ
	1	NZ	MZ	MZ
	2	NZ	NZ	MZ // case x:NZ
	3	NZ	NZ	NZ
	4	Z	NZ	NZ
	5	Z	Z	NZ // case x:Z

29 February 2008

Analysis of Software Artifacts
© 2008 Jonathan Aldrich

18

Abstract Summaries

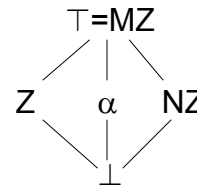


/* Summary	Compute summary of double for x:α			
* Case x:α -> result:α	p	x	result	
*/	0	α	MZ	
int double(int x) {	1	α	α	
[result := 2*x];				
}				
	Analyze client			
	p	x	y	z
void caller() {	0	MZ	MZ	MZ
[x := 5];	1	NZ	MZ	MZ
[y := double(x)];	2	NZ	NZ	MZ // α=NZ
[z := 10/y];	3	NZ	NZ	NZ
[x := 0];	4	Z	NZ	NZ
[y := double(x)];	5	Z	Z	NZ // α=Z
}				

Abstract Summaries for Zero Analysis



- New ZA lattice has α
- Flow functions
 - $f_{ZA}(\sigma, [x]_k) = [t_k \mapsto \sigma(x)] \sigma$
 - $f_{ZA}(\sigma, [n]_k) = \text{if } n=0$
 then $[t_k \mapsto Z] \sigma$
 else $[t_k \mapsto NZ] \sigma$
 - $f_{ZA}(\sigma, [x := [\dots]_n]_k) = [x \mapsto \sigma(t_n)] \sigma$
 - $f_{ZA}(\sigma, [[\dots]_n \text{ op } [\dots]_m]_k) =$
 if $\text{op} = *$ and $\sigma[t_m] = NZ$
 then $[t_k \mapsto \sigma(t_n)] \sigma$ // this case used in example
 if ...
 else $[t_k \mapsto MZ] \sigma$
 - $f_{ZA}(\sigma, /* \text{ any other } */) = \sigma$
- Many other ways to generate summaries



Comparison



- Assumptions
 - Simple, efficient
 - Imprecise
- Annotations
 - Require effort
 - More precise than assumptions
 - More efficient than IP analysis
 - Can use “summary annotations” to get context sensitivity
- Both work on partial programs
- Interprocedural CFG
 - Simple for programmer
 - As precise as simple annotations
 - Still imprecise, can be very costly
 - $O(n^3)$ in size of program
- Summaries
 - Excellent precision
 - Costly if not abstract
- Both require whole program