Lecture 9: Interprocedural Analysis

17-355/17-665/17-819: Program Analysis

Rohan Padhye

February 15, 2022

* Course materials developed with Jonathan Aldrich and Claire Le Goues

Extend WHILE with functions

Extend WHILE3ADDR with functions

$$F \quad ::= \quad \text{fun } f(x) \; \{ \; \overline{n : I} \; \}$$
$$I \quad ::= \quad \ldots \;|\; \text{return } x \;|\; y := f(x)$$

Extend WHILE3ADDR with functions

$$F ::= \text{fun } f(x)\ \{\ \overline{n : I}\ \}$$
$$I ::= \ldots \mid \text{return } x \mid y := f(x)$$

$$
\begin{aligned}
1 &: \quad \text{fun } double(x) : int \\
2 &: \qquad y := 2 * x \\
3 &: \qquad \text{return } y \\
4 &: \quad \text{fun } main() : void \\
5 &: \qquad z := 0 \\
6 &: \qquad w := double(z)
\end{aligned}
$$

Extend WHILE3ADDR with functions

$$
\begin{array}{ll}
1: & \text{fun } \mathit{divByX}(x) : \mathit{int} \\
2: & \quad y := 10/x \\
3: & \quad \text{return } y \\
4: & \text{fun } \mathit{main}() : \mathit{void} \\
5: & \quad z := 5 \\
6: & \quad w := \mathit{divByX}(z)
\end{array}
$$

$$
\begin{array}{ll}
1: & \text{fun } \mathit{double}(x) : \mathit{int} \\
2: & \quad y := 2 * x \\
3: & \quad \text{return } y \\
4: & \text{fun } \mathit{main}() : \mathit{void} \\
5: & \quad z := 0 \\
6: & \quad w := \mathit{double}(z)
\end{array}
$$

Data-Flow Analysis

HOW DO WE ANALYZE THESE PROGRAMS?

Approach #1: Analyze functions independently

- Pretend function *f()* cannot see the source of function *g()*
- Simulates separate compilation and dynamic linking (e.g. C, Java)
- Create CFG for each function body and run **intraprocedural** analysis
- **Q**: What should be is σ_0 and $f_Z[\![x := g(y)]\!]$ and $f_Z[\![\text{return } x]\!]$ for zero analysis?

$$\sigma_0 =$$

$$f[\![x := g(y)]\!](\sigma) =$$

$$f[\![\textbf{return } x]\!](\sigma) =$$

Can we show that division on line 2 is safe?

$$
\begin{array}{ll}
1: & \text{fun } \mathit{divByX}(x) : \mathit{int} \\
2: & \quad y := 10/x \\
3: & \quad \text{return } y \\
4: & \text{fun } \mathit{main}() : \mathit{void} \\
5: & \quad z := 5 \\
6: & \quad w := \mathit{divByX}(z)
\end{array}
$$

Approach #2: User-defined Annotations

@NonZero -> @NonZero

$$
\begin{array}{l}
\text{fun } \mathit{divByX}(x) : \mathit{int} \\
\quad y := 10/x \\
\quad \text{return } y \\
\text{fun } \mathit{main}() : \mathit{void} \\
\quad z := 5 \\
\quad w := \mathit{divByX}(z)
\end{array}
$$

$$
\begin{array}{lll}
f[\![x := g(y)]\!](\sigma) & = \sigma[x \mapsto \mathit{annot}[\![g]\!].r] & (\text{error if } \sigma(y) \not\sqsubseteq \mathit{annot}[\![g]\!].a) \\
f[\![\text{return } x]\!](\sigma) & = \sigma & (\text{error if } \sigma(x) \not\sqsubseteq \mathit{annot}[\![g]\!].r)
\end{array}
$$

Approach #2: User-defined Annotations

@NonZero -> @NonZero

$$\text{fun } \mathit{divByX}(x) : \mathit{int}$$
$$\quad y := 10/x$$
$$\quad \text{return } y$$
$$\text{fun } \mathit{main}() : \mathit{void}$$
$$\quad z := 5$$
$$\quad w := \mathit{divByX}(z)$$

@NonZero -> @NonZero

$$\text{fun } \mathit{double}(x) : \mathit{int}$$
$$\quad y := 2 * x$$
$$\quad \text{return } y$$
$$\text{fun } \mathit{main}() : \mathit{void}$$
$$\quad z := 0$$
$$\quad w := \mathit{double}(z)$$

Error!

$$f[\![x := g(y)]\!](\sigma) = \sigma[x \mapsto \mathit{annot}[\![g]\!].r] \quad (\text{error if } \sigma(y) \not\equiv \mathit{annot}[\![g]\!].a)$$
$$f[\![\text{return } x]\!](\sigma) = \sigma \quad (\text{error if } \sigma(x) \not\equiv \mathit{annot}[\![g]\!].r)$$

Approach #2: User-defined Annotations

@NonZero -> @NonZero

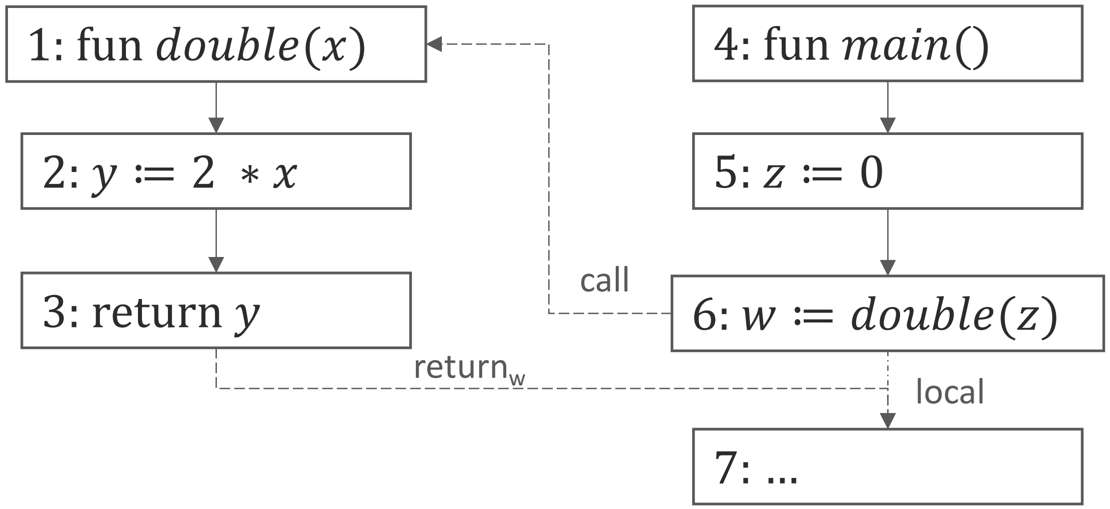
$$
\begin{array}{l}
\text{fun } \mathit{divByX}(x) : \mathit{int} \\
\quad y := 10/x \\
\quad \text{return } y \\
\text{fun } \mathit{main}() : \mathit{void} \\
\quad z := 5 \\
\quad w := \mathit{divByX}(z)
\end{array}
$$

@Any -> @NonZero

$$
\begin{array}{l}
\text{fun } \mathit{double}(x) : \mathit{int} \\
\quad y := 2 * x \\
\quad \text{return } y \quad \textbf{Error!} \\
\text{fun } \mathit{main}() : \mathit{void} \\
\quad z := 0 \\
\quad w := \mathit{double}(z)
\end{array}
$$

$$
\begin{array}{lll}
f[\![x := g(y)]\!](\sigma) & = \sigma[x \mapsto \mathit{annot}[\![g]\!].r] & (\text{error if } \sigma(y) \not\sqsubseteq \mathit{annot}[\![g]\!].a) \\
f[\![\text{return } x]\!](\sigma) & = \sigma & (\text{error if } \sigma(x) \not\sqsubseteq \mathit{annot}[\![g]\!].r)
\end{array}
$$

Approach #3: Interprocedural CFG



$$f_Z[\![x := g(y)]\!]_{local}(\sigma) = \sigma \setminus (\{x\} \cup \; Globals)$$

$$f_Z[\![x := g(y)]\!]_{call}(\sigma) = \{v \mapsto \sigma(v) | \, v \in Globals\} \cup \{formal(g) \mapsto \sigma(y)\}$$

$$f_Z[\![\text{return } y]\!]_{return_x}(\sigma) = \{v \mapsto \sigma(v) | \, v \in Globals\} \cup \{x \mapsto \sigma(y)\}$$

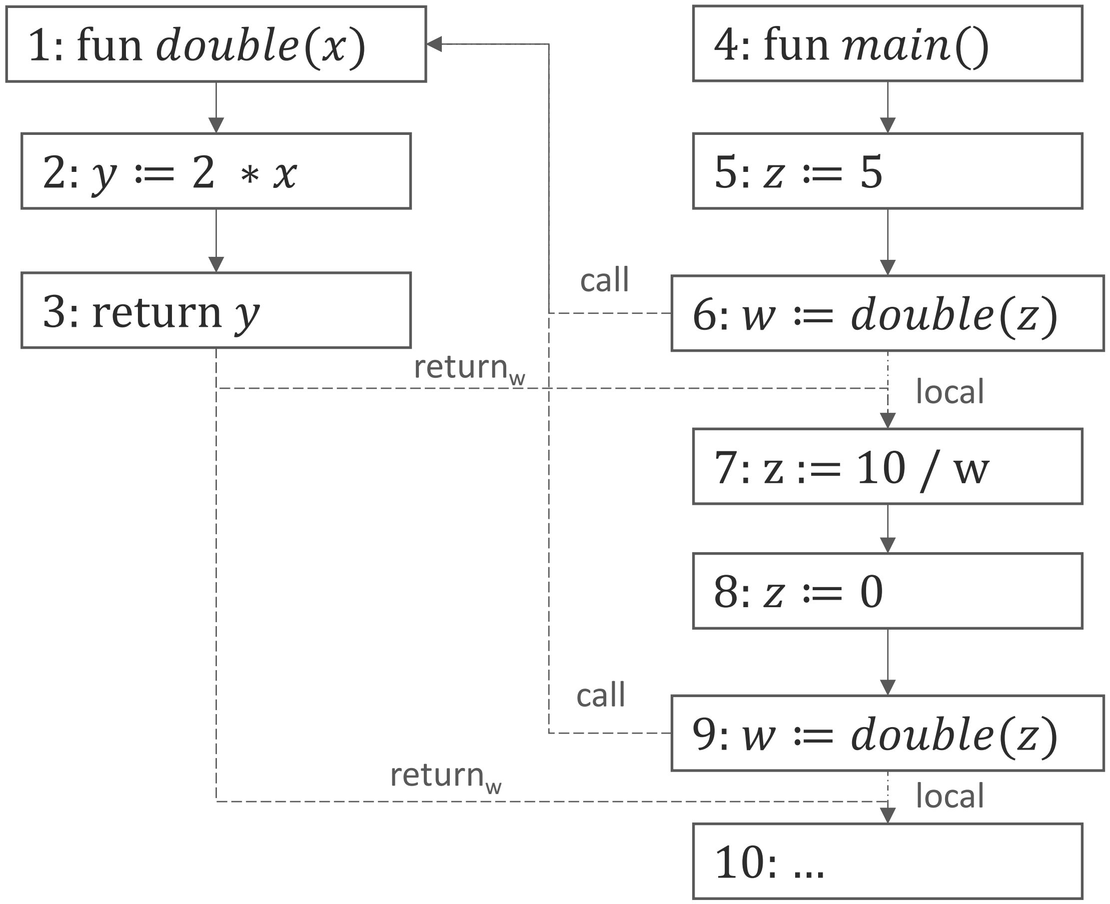
Approach #3: Interprocedural CFG

Exercise: What would be the result of zero analysis for this program at line 7 and at the end (after line 9)?

```
1 :  fun double(x) : int
2 :      y := 2 * x
3 :      return y

4 :  fun main()
5 :      z := 5
6 :      w := double(z)
7 :      z := 10/w
8 :      z := 0
9 :      w := double(z)
```

Approach #3: Interprocedural CFG



$$
\begin{array}{ll}
1: & \text{fun } double(x) : int \\
2: & \quad y := 2 * x \\
3: & \quad \text{return } y \\
4: & \text{fun } main() \\
5: & \quad z := 5 \\
6: & \quad w := double(z) \\
7: & \quad z := 10/w \\
8: & \quad z := 0 \\
9: & \quad w := double(z)
\end{array}
$$

$$f_Z[\![x := g(y)]\!]_{local}(\sigma) = \sigma \setminus (\{x\} \cup Globals)$$

$$f_Z[\![x := g(y)]\!]_{call}(\sigma) = \{v \mapsto \sigma(v) \mid v \in Globals\} \cup \{formal(g) \mapsto \sigma(y)\}$$

$$f_Z[\![\text{return } y]\!]_{return_x}(\sigma) = \{v \mapsto \sigma(v) \mid v \in Globals\} \cup \{x \mapsto \sigma(y)\}$$

Problems with Interprocedural CFG

- Merges (joins) information across call sites to same function
- Loses precision
- Models infeasible paths (call from one site and return to another)
- Can we “remember” where to return data-flow values?

Enter:

CONTEXT-SENSITIVE ANALYSIS

Context-Sensitive Analysis Example

```
fun double(x) : int
    y := 2 * x
    return y

fun main()
    z := 5
    w := double(z)
    z := 10/w
    z := 0
    w := double(z)
```

Key idea: Separate analyses for functions called in different ”contexts”.

(“context” = some statically definable condition)

Context-Sensitive Analysis Example

```
1 : fun double(x) : int
2 :     y := 2 * x
3 :     return y

4 : fun main()
5 :     z := 5
6 :     w := double(z)
7 :     z := 10/w
8 :     z := 0
9 :     w := double(z)
```

Context	σ_{in}	σ_{out}
Line 6	{x->N}	{x->N, y->N}
Line 9	{x->Z}	{x->Z, y->Z}

Context-Sensitive Analysis Example

$$
\begin{array}{l}
\text{fun } double(x) : int \\
\quad y := 2 * x \\
\quad \text{return } y \\
\\
\text{fun } main() \\
\quad z := 5 \\
\quad w := double(z) \\
\quad z := 10/w \\
\quad z := 0 \\
\quad w := double(z)
\end{array}
$$

Context	σ_{in}	σ_{out}
<main, T>	T	{w->Z, Z->Z}
<double, N>	{x->N}	{x->N, y->N}
<double, Z>	{x->Z}	{x->Z, y->Z}

type $Context$
  val $fn : Function$
  val $input : \sigma$

type $Summary$
  val $input : \sigma$
  val $output : \sigma$

val $results : Map[Context, Summary]$

Context	σ_{in}	σ_{out}
<main, T>	T	{w->Z, Z->Z}
<double, N>	{x->N}	{x->N, y->N}
<double, Z>	{x->Z}	{x->Z, y->Z}

Works for non-recursive contexts!

function GETCTX($f, callingCtx, n, \sigma_{in}$)
  return $Context(f, \sigma_{in})$
end function

function ANALYZE(ctx, σ_{in})
  $\sigma'_{out} \leftarrow$ INTRAPROCEDURAL(ctx, σ_{in})
  $results[ctx] \leftarrow Summary(\sigma_{in}, \sigma'_{out})$
  return σ'_{out}
end function

function FLOW($[\![n\colon x := f(y)]\!], ctx, \sigma_n$)
  $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$
  $calleeCtx \leftarrow$ GETCTX(f, ctx, n, σ_{in})
  $\sigma_{out} \leftarrow$ RESULTSFOR($calleeCtx, \sigma_{in}$)
  return $\sigma_n[x \mapsto \sigma_{out}[result]]$
end function

function RESULTSFOR(ctx, σ_{in})
  if $ctx \in \mathrm{dom}(results)$ **then**
    if $\sigma_{in} \sqsubseteq results[ctx].input$ **then**
      return $results[ctx].output$
    else
      return ANALYZE($ctx, results[ctx].input \sqcup \sigma_{in}$)
    end if
  else
    return ANALYZE(ctx, σ_{in})
  end if
end function