# Lecture 9: Interprocedural Analysis

17-355/17-655/17-819: Program Analysis

Rohan Padhye and Jonathan Aldrich

March 4, 2021

* Course materials developed with Claire Le Goues

# Extend WHILE with functions

# Extend WHILE3ADDR with functions

$$\begin{array}{lcl} F & ::= & \mathsf{fun}\ f(x)\ \{\ \overline{n : I}\ \} \\ I & ::= & \ldots \mid \mathsf{return}\ x \mid y := f(x) \end{array}$$

# Extend WHILE3ADDR with functions

$$
\begin{array}{lll}
F & ::= & \text{fun } f(x)\ \{\ \overline{n : I}\ \} \\
I & ::= & \ldots \mid \textbf{return } x \mid y := f(x)
\end{array}
$$

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

# Extend WHILE3ADDR with functions

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

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

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 $\sigma_0$ and $f_Z[\![x := g(y)]\!]$ and $f_Z[\![\text{return } x]\!]$ for zero analysis?

$$\sigma_0 =$$

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

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

# Can we show that division on line 2 is safe?

```
1 :  fun divByX(x) : int
2 :      y := 10/x
3 :      return y

4 :  fun main() : void
5 :      z := 5
6 :      w := divByX(z)
```

# 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**

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

**@NonZero -> @NonZero**

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

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

# Approach #2: User-defined Annotations

@NonZero -> @NonZero

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

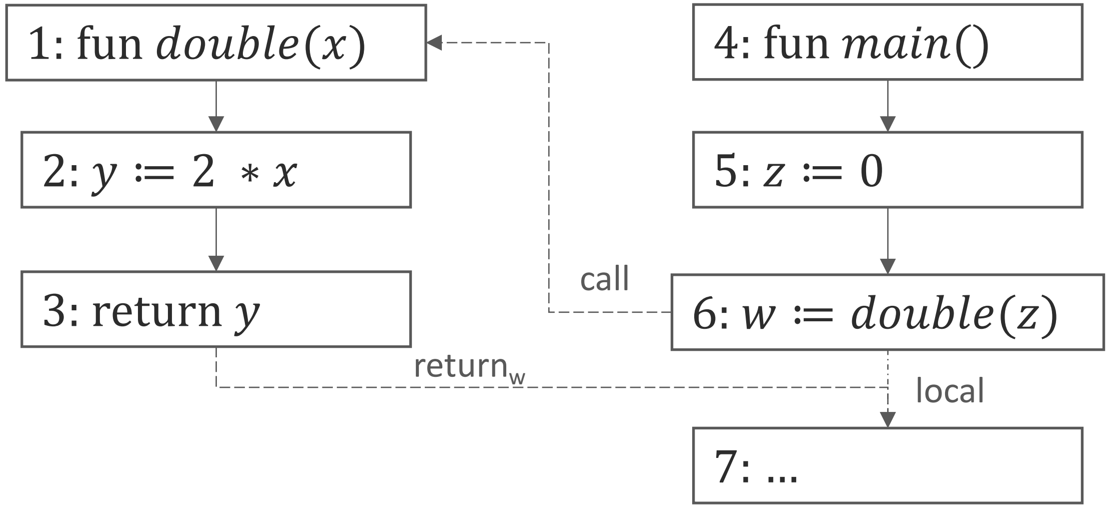
@Any -> @NonZero

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

Error!

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

# Approach #3: Interprocedural CFG




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

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

$$f_Z[\![x := g(y)]\!]_{call}(\sigma) = \{formal(g) \to \sigma(y)\}$$

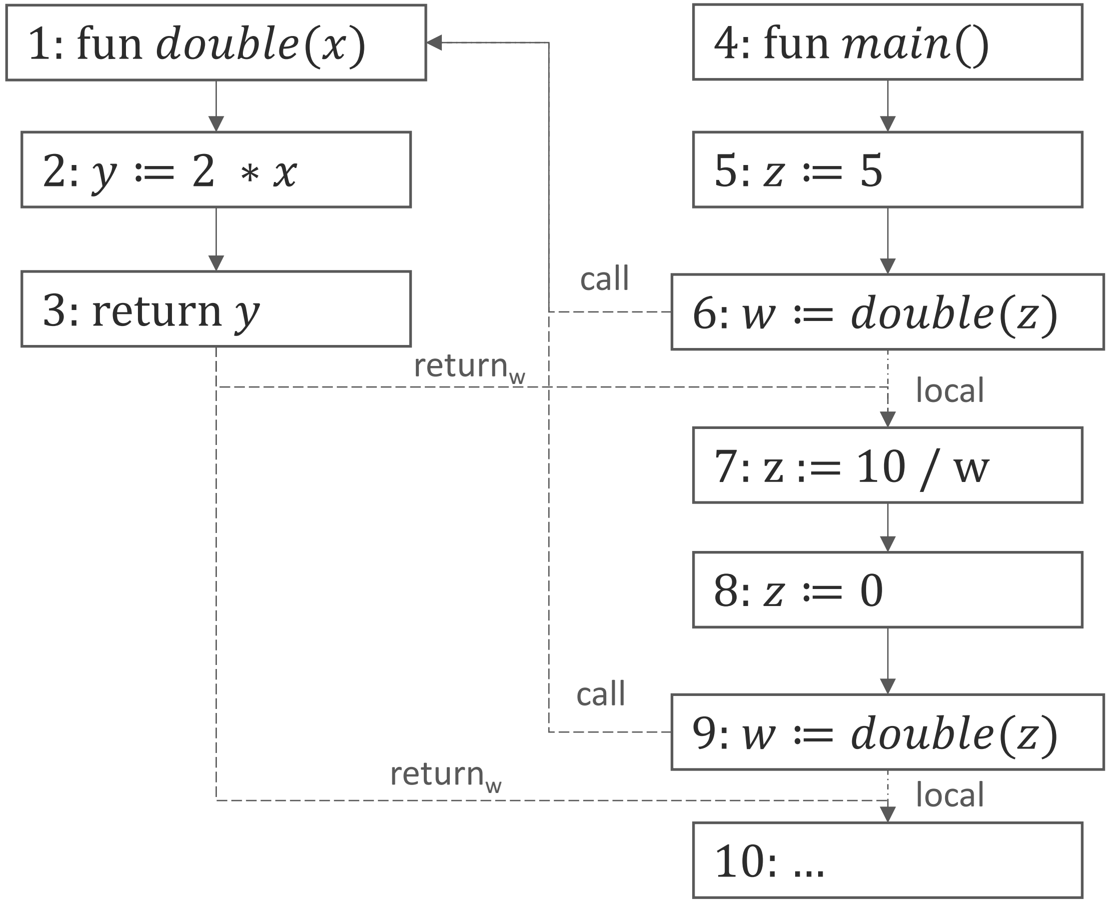
$$f_Z[\![\text{return } x]\!]_{ret}(\sigma) = \{z \to \sigma(z) \mid z \in Globals\} \cup \{ret \to \sigma(x)\}$$

# Approach #3: Interprocedural CFG

**Exercise**: What would be the result of zero analysis for this program on line 7 and at the end?

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

# Approach #3: Interprocedural CFG




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

# 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 | $\sigma_{in}$ | $\sigma_{out}$ |
|---|---|---|
| Line 6 | {x->N} | {x->N, y->N} |
| Line 9 | {x->Z} | {x->Z, y->Z} |

# Context-Sensitive Analysis Example

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

| Context | $\sigma_{in}$ | $\sigma_{out}$ |
|---|---|---|
| <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** $output : \sigma$

| Context | $\sigma_{in}$ | $\sigma_{out}$ |
|---|---|---|
| <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, σ_in)
    return Context(f, σ_in)
end function
```

**val** $results : Map[Context, Summary]$

```
function Analyze(ctx, σ_in)
    σ'_out ← Intraprocedural(ctx, σ_in)
    results[ctx] ← Summary(σ_in, σ'_out)
    return σ'_out
end function
```

```
function Flow(⟦n: x := f(y)⟧, ctx, σ_n)
    σ_in ← [formal(f) ↦ σ_n(y)]
    calleeCtx ← GetCtx(f, ctx, n, σ_in)
    σ_out ← ResultsFor(calleeCtx, σ_in)
    return σ_n[x ↦ σ_out[result]]
end function
```

```
function ResultsFor(ctx, σ_in)
    if ctx ∈ dom(results) then
        if σ_in ⊑ results[ctx].input then
            return results[ctx].output
        else
            return Analyze(ctx, results[ctx].input ⊔ σ_in)
        end if
    else
        return Analyze(ctx, σ_in)
    end if
end function
```