Lecture 9: Context-Sensitive Analysis
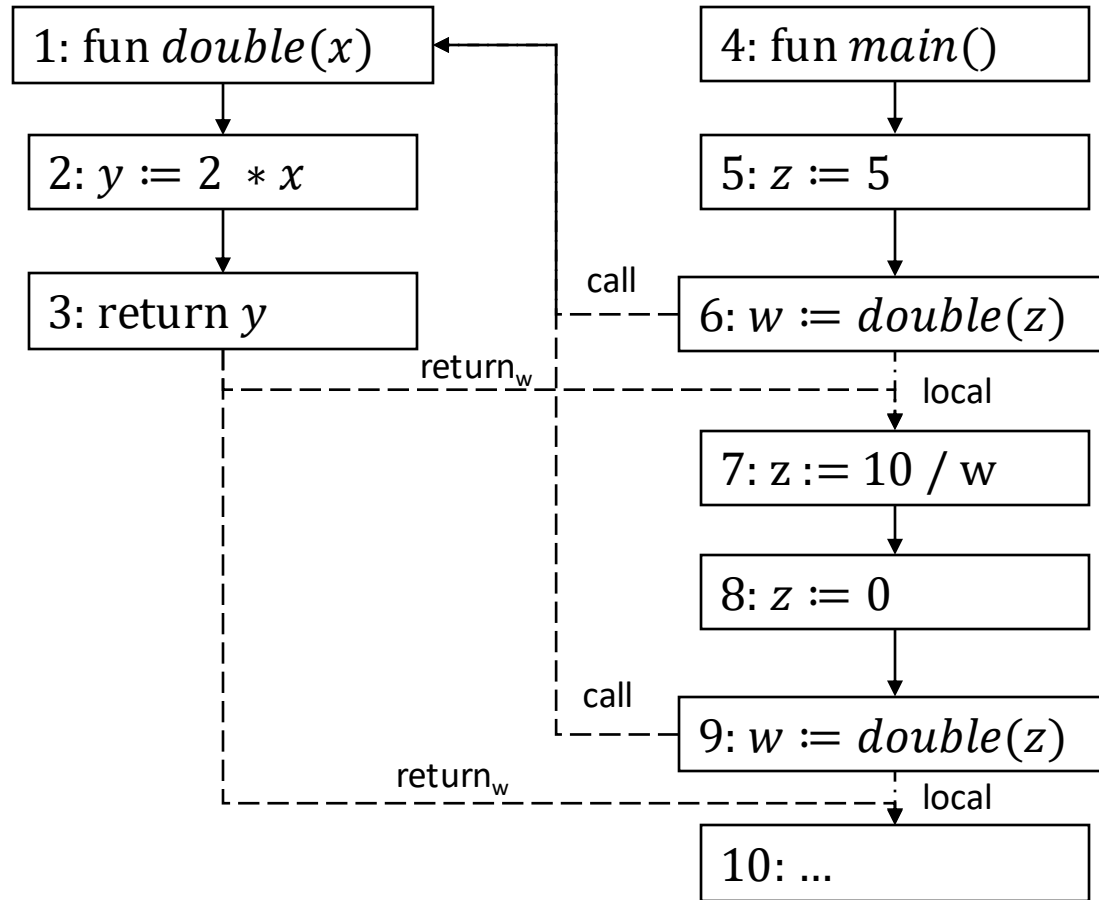
17-355/17-665/17-819: Program Analysis

Rohan Padhye

September 25, 2025

Recap: Interprocedural CFG

1: fun $double(x)$

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: $w := double(z)$

10: ...

call

return$_w$

local

call

return$_w$

local

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

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

S3D Software and Societal Systems Department

Carnegie Mellon University

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?

Context-Sensitive Analysis Example

$$1: \quad \textbf{fun } double(x) : int$$
$$2: \qquad y := 2 * x$$
$$3: \qquad \textbf{return } y$$

$$4: \quad \textbf{fun } main()$$
$$5: \qquad z := 5$$
$$6: \qquad w := double(z)$$
$$7: \qquad z := 10/w$$
$$8: \qquad z := 0$$
$$9: \qquad w := double(z)$$

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

("context" = some statically definable condition)

Context-Sensitive Analysis Example

$$1: \quad \text{fun } double(x) : int$$
$$2: \quad\quad y := 2 * x$$
$$3: \quad\quad \text{return } y$$

$$4: \quad \text{fun } main()$$
$$5: \quad\quad z := 5$$
$$6: \quad\quad w := double(z)$$
$$7: \quad\quad z := 10/w$$
$$8: \quad\quad z := 0$$
$$9: \quad\quad w := double(z)$$

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

```
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}
<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 $\textsc{GetCtx}(f, callingCtx, n, \sigma_{in})$
   return $Context(f, \sigma_{in})$
end function

function $\textsc{Analyze}(ctx, \sigma_{in})$
   $\sigma'_{out} \leftarrow \textsc{Intraprocedural}(ctx, \sigma_{in})$
   $results[ctx] \leftarrow Summary(\sigma_{in}, \sigma'_{out})$
   return σ'_{out}
end function

function $\textsc{ResultsFor}(ctx, \sigma_{in})$
   if $ctx \in \mathrm{dom}(results)$ **then**

       return $results[ctx].output$

   else
       return $\textsc{Analyze}(ctx, \sigma_{in})$
   end if
end function

function $\textsc{Flow}(\llbracket n: x := f(y) \rrbracket, ctx, \sigma_n)$
   $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$
   $calleeCtx \leftarrow \textsc{GetCtx}(f, ctx, n, \sigma_{in})$
   $\sigma_{out} \leftarrow \textsc{ResultsFor}(calleeCtx, \sigma_{in})$
   return $\sigma_n[x \mapsto \sigma_{out}[result]]$
end function

type $Context$
   val $fn : Function$
   val $string : List[Int]$

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

val $results : Map[Context, Summary]$

Context	σ_{in}	σ_{out}
<main, []>	T	{w->Z, Z->Z}
<double, [6]>	{x->N}	{x->N, y->N}
<double, [9]>	{x->Z}	{x->Z, y->Z}

Works for non-recursive contexts!

function $\textsc{GetCtx}(f, callingCtx, n, \sigma_{in})$
   $newStr \leftarrow callingCtx.string \mathbin{++} n$
   return $Context(f, newStr)$
end function

function $\textsc{Analyze}(ctx, \sigma_{in})$
   $\sigma'_{out} \leftarrow \textsc{Intraprocedural}(ctx, \sigma_{in})$
   $results[ctx] \leftarrow Summary(\sigma_{in}, \sigma'_{out})$
   return σ'_{out}
end function

function $\textsc{ResultsFor}(ctx, \sigma_{in})$
   if $ctx \in \mathrm{dom}(results)$ **then**

      return $results[ctx].output$

   else
      return $\textsc{Analyze}(ctx, \sigma_{in})$
   end if
end function

function $\textsc{Flow}(\llbracket n\colon x := f(y) \rrbracket, ctx, \sigma_n)$
   $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$
   $calleeCtx \leftarrow \textsc{GetCtx}(f, ctx, n, \sigma_{in})$
   $\sigma_{out} \leftarrow \textsc{ResultsFor}(calleeCtx, \sigma_{in})$
   return $\sigma_n[x \mapsto \sigma_{out}[result]]$
end function

type $Context$
    val $fn : Function$
    val $string : List[Int]$

Works for non-recursive contexts!

Context	σ_{in}	σ_{out}
<main, []>	T	{w->Z, Z->Z}
<double, [6]>	{x->N}	{x->N, y->N}
<double, [9]>	{x->Z}	{x->Z, y->Z}

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

function $\text{GETCTX}(f, callingCtx, n, \sigma_{in})$
    $newStr \leftarrow callingCtx.string \,\text{++}\, n$
    return $Context(f, newStr)$
end function

val $results : Map[Context, Summary]$

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

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

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

Recursion makes this a bit harder

```
fun factorial(n) {
  if n == 0 then
    return 1
  else
    return n * factorial(n-1)
}

fun main() {
  x := factorial(5)
}
```

```
1: fun main():
  2:  x := 5
  3:  result := factorial(x)
  4:  print result
  5:  halt
6: fun factorial(n):
  7:  one := 1
  8:  result := one
  9:  if n = 0 goto 13
  10: prev := n - one
  11: temp := factorial(prev)
  12: result := n * temp
  13: return result
```

Exercise: Work out why this is a problem
with both value-based contexts and call-strings based contexts

Recursion makes this a bit harder

```
fun factorial(n) {
  if n == 0 then
    return 1
  else
    return n * factorial(n-1)
}

fun main() {
  x := factorial(5)
}
```

```
1: fun main():
  2:  x := 5
  3:  result := factorial(x)
  4:  print result
  5:  halt
6: fun factorial(n):
  7:  one := 1
  8:  result := one
  9:  if n = 0 goto 13
  10: prev := n - one
  11: temp := factorial(prev)
  12: result := n * temp
  13: return result
```

Context	σ_{in}	σ_{out}
<main, T>	T	…
<factorial, N>	N	…
<factorial, T>	T	…

Exercise: Work out why this is a problem
with both value-based contexts and call-strings based contexts

Recursion makes this a bit harder

```
fun factorial(n) {
  if n == 0 then
    return 1
  else
    return n * factorial(n-1)
}

fun main() {
  x := factorial(5)
}
```

```
1: fun main():
  2:   x := 5
  3:   result := factorial(x)
  4:   print result
  5:   halt
6: fun factorial(n):
  7:   one := 1
  8:   result := one
  9:   if n = 0 goto 13
  10: prev := n - one
  11: temp := factorial(prev)
  12: result := n * temp
  13: return result
```

Context	σ_{in}	σ_{out}
<main, []>	T	...
<factorial, [3]>	N	...
<factorial, [3, 11]>	T	...
...		

Exercise: Work out why this is a problem
with both value-based contexts and call-strings based contexts

Key Idea: Worklist of Contexts

val $worklist : Set[Context]$
val $analyzing : Set[Context]$
val $results : Map[Context, Summary]$
val $callers : Map[Context, Set[Context]]$

Key Idea: Worklist of Contexts

val $worklist : Set[Context]$
val $analyzing : Set[Context]$
val $results : Map[Context, Summary]$
val $callers : Map[Context, Set[Context]]$

```
function ANALYZEPROGRAM
    initCtx ← GETCTX(main, nil, 0, ⊤)
    worklist ← {initCtx}
    results[initCtx] ← Summary(⊤, ⊥)
    while NOTEMPTY(worklist) do
        ctx ← REMOVE(worklist)
        ANALYZE(ctx, results[ctx].input)
    end while
end function
```

Key Idea: Worklist of Contexts

val $worklist : Set[Context]$
val $analyzing : Set[Context]$
val $results : Map[Context, Summary]$
val $callers : Map[Context, Set[Context]]$

function ANALYZEPROGRAM
    $initCtx \leftarrow$ GETCTX(main, **nil**, $0, \top$)
    $worklist \leftarrow \{initCtx\}$
    $results[initCtx] \leftarrow Summary(\top, \bot)$
    while NOTEMPTY($worklist$) **do**
        $ctx \leftarrow$ REMOVE($worklist$)
        ANALYZE($ctx, results[ctx].input$)
    end while
end function

function ANALYZE(ctx, σ_{in})
    $\sigma_{out} \leftarrow results[ctx].output$
    ADD($analyzing, ctx$)
    $\sigma'_{out} \leftarrow$ INTRAPROCEDURAL(ctx, σ_{in})
    REMOVE($analyzing, ctx$)
    if $\sigma'_{out} \not\sqsubseteq \sigma_{out}$ **then**
        $results[ctx] \leftarrow Summary(\sigma_{in}, \sigma_{out} \sqcup \sigma'_{out})$
        for $c \in callers[ctx]$ **do**
            ADD($worklist, c$)
        end for
    end if
    return σ'_{out}
end function

Key Idea: Worklist of Contexts

$\textbf{val } worklist : Set[Context]$
$\textbf{val } analyzing : Set[Context]$
$\textbf{val } results : Map[Context, Summary]$
$\textbf{val } callers : Map[Context, Set[Context]]$

$\textbf{function } \text{FLOW}(\llbracket n{:}\ x := f(y) \rrbracket, ctx, \sigma_n)$
$\quad \sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)] \qquad \triangleright$
$\quad calleeCtx \leftarrow \text{GETCTX}(f, ctx, n, \sigma_{in})$
$\quad \sigma_{out} \leftarrow \text{RESULTSFOR}(calleeCtx, \sigma_{in})$
$\quad \boxed{\text{ADD}(callers[calleeCtx], ctx)}$
$\quad \textbf{return } \sigma_n[x \mapsto \sigma_{out}[result]]$

$\textbf{function } \text{ANALYZE}(ctx, \sigma_{in})$
$\quad \sigma_{out} \leftarrow results[ctx].output$
$\quad \text{ADD}(analyzing, ctx)$
$\quad \sigma'_{out} \leftarrow \text{INTRAPROCEDURAL}(ctx, \sigma_{in})$
$\quad \text{REMOVE}(analyzing, ctx)$
$\quad \textbf{if } \sigma'_{out} \not\sqsubseteq \sigma_{out} \textbf{ then}$
$\quad\quad results[ctx] \leftarrow Summary(\sigma_{in}, \sigma_{out} \sqcup \sigma'_{out})$
$\quad\quad \textbf{for } c \in callers[ctx] \textbf{ do}$
$\quad\quad\quad \text{ADD}(worklist, c)$
$\quad\quad \textbf{end for}$
$\quad \textbf{end if}$
$\quad \textbf{return } \sigma'_{out}$
$\textbf{end function}$

```
function RESULTSFOR(ctx, σ_in)
    if ctx ∈ dom(results) then
        if σ_in ⊑ results[ctx].input then
            return results[ctx].output                    ▷ existing results are good
        else
            results[ctx].input ← results[ctx].input ⊔ σ_in  ▷ keep track of more general input
        end if
    else
        results[ctx] = Summary(σ_in, ⊥)                   ▷ initially optimisti
    end if
    if ctx ∈ analyzing then
        return results[ctx].output    ▷ ⊥ if it hasn't been analyzed yet; otherwis
    else
        return ANALYZE(ctx, results[ctx].input)
    end if
end function
```

```
function ANALYZE(ctx, σ_in)
    σ_out ← results[ctx].output
    ADD(analyzing, ctx)
    σ'_out ← INTRAPROCEDURAL(ctx, σ_in)
    REMOVE(analyzing, ctx)
    if σ'_out ⋢ σ_out then
        results[ctx] ← Summary(σ_in, σ_out ⊔ σ'_out)
        for c ∈ callers[ctx] do
            ADD(worklist, c)
        end for
    end if
    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)
    ADD(callers[calleeCtx], ctx)
    return σ_n[x ↦ σ_out[result]]
```

On Termination and Complexity

- Add to worklist C x H times (C = #contexts, H = lattice height)
- After each analysis, propagate result to N callers
- O(C x N x H) intraprocedural analyses
- = O(E x H) where E is #edges in context-sensitive call graph

- Is C finite???

Types of Context-Sensitivity

- No context sensitivity
- Call strings
- Value contexts
- k-limited call strings
- k-limited value contexts

Limited Context-Sensitivity

Value-based context-sensitivity

No context-sensitivity

```
type Context
    val fn : Function

function GETCTX(f, callingCtx, n, σ_in)
    return Context(f)
end function
```

```
function GETCTX(f, callingCtx, n, σ_in)
    return Context(f, σ_in)
end function
```

K-call-string context-sensitivity

```
type Context
    val fn : Function
    val string : List[Int]

function GETCTX(f, callingCtx, n, σ_in)
    newStr ← SUFFIX(callingCtx.string ++ n, CALL_STRING_CUTOFF)
    return Context(f, newStr)
end function
```

S3D
Software and Societal
Systems Department

Carnegie
Mellon
University

In Practice

- Value contexts = same precision as arbitrary-length call strings
  - Only former guaranteed to terminate, but still very expensive
- If flow functions are *distributive*, more efficient algorithms exist (e.g. IFDS)
- K-call strings is often used for general analyses