Lecture 9: Context-Sensitive Analysis

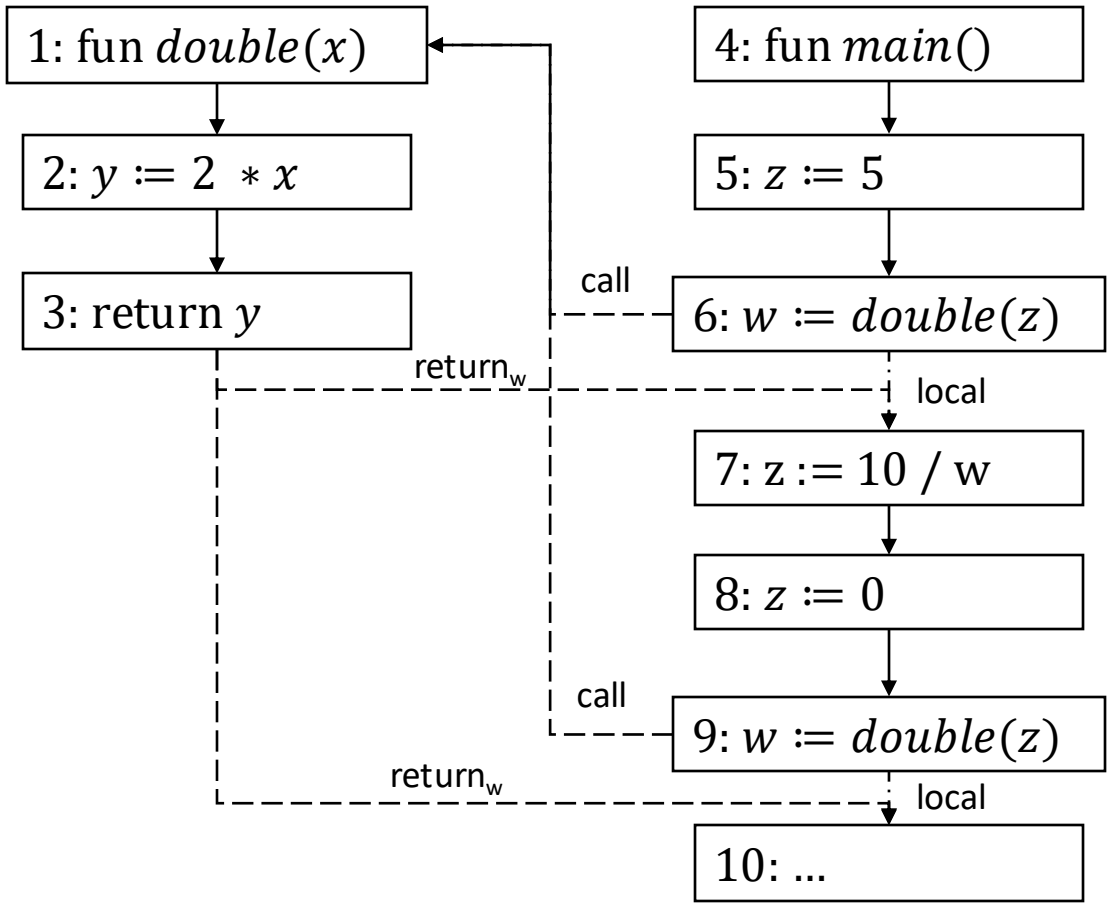
17-355/17-665/17-819: Program Analysis

Rohan Padhye

September 25, 2025

* Course materials developed with Jonathan Aldrich and Claire Le Goues

Recap: Interprocedural CFG



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

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

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

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

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?

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)
```

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

```
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 $output : \sigma$

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

val $results : Map[Context, Summary]$

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($[\![n\text{: } x := f(y)]\!], ctx, \sigma_n$)
  $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$
  $calleeCtx \leftarrow$ 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 $ctx \in \text{dom}(results)$ **then**
    return $results[ctx].output$
  else
    return ANALYZE(ctx, σ_{in})
  end if
end function

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

```
type Summary
    val input : σ
    val output : σ
```

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 GETCTX(f, callingCtx, n, σ_in)
    newStr ← callingCtx.string ++ n
    return Context(f, newStr)
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

            return results[ctx].output

    else
        return ANALYZE(ctx, σ_in)
    end if
end function
```

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

```
type Summary
    val input : σ
    val output : σ
```

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 GetCtx(f, callingCtx, n, σ_in)
    newStr ← callingCtx.string ++ n
    return Context(f, newStr)
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
```

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:  z := 5
    3:  x := factorial(z)
    4:  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)
}
```

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

```
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)
}
```

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

```
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

Key Idea: Worklist of Contexts

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

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 ← 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
```

```
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
```

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 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$(calleeCtx, \sigma_{in})$
  ADD$(callers[calleeCtx], ctx)$
  return $\sigma_n[x \mapsto \sigma_{out}[result]]$

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

```
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 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]]
```

```
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
```

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

No context-sensitivity

```
type Context
    val fn : Function

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

Value-based context-sensitivity

```
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
```

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