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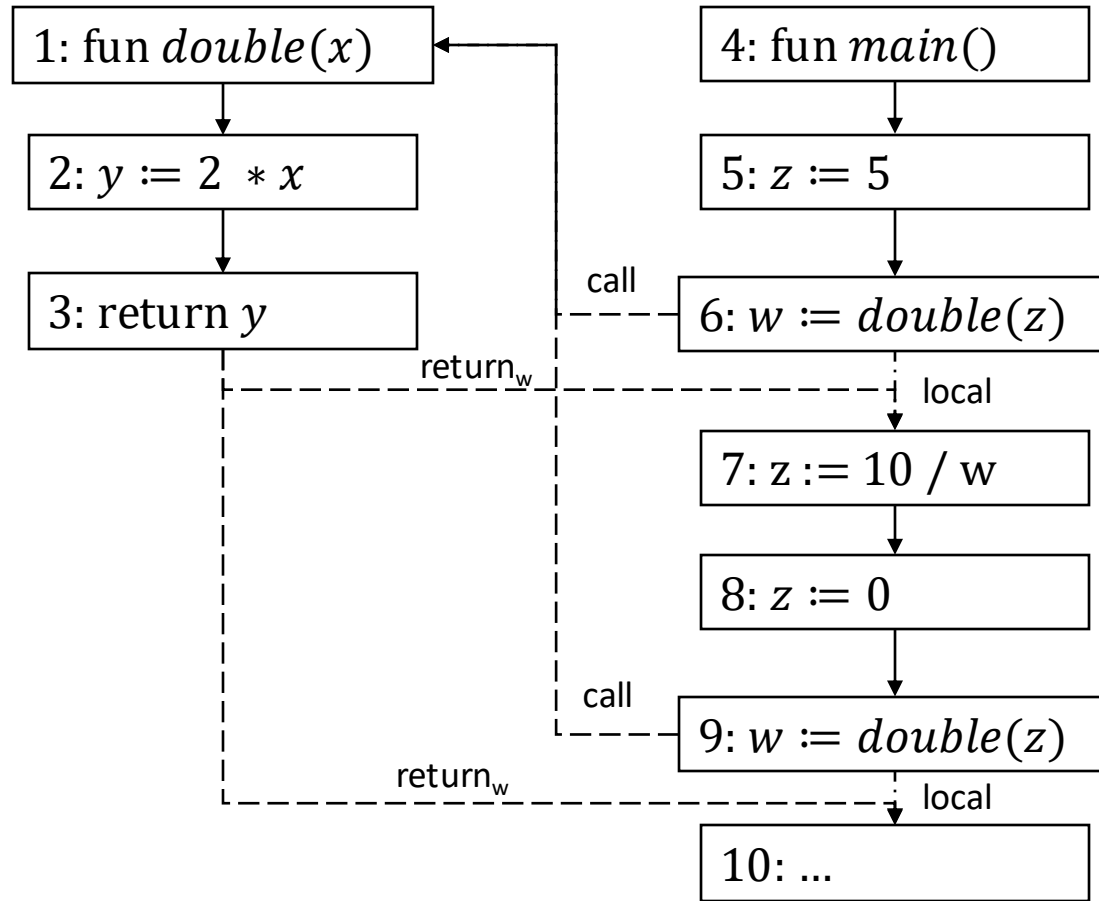
17-355/17-665/17-819: Program Analysis

Rohan Padhye

September 25, 2025

* Course materials developed with Jonathan Aldrich and Claire Le Goues

Recap: Interprocedural CFG

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

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

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

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

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

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

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 $double(x) : int$

$2:$ $\quad y := 2 * x$

$3:$ $\quad$ return y

$4:$ fun $main()$

$5:$ $\quad z := 5$

$6:$ $\quad w := double(z)$

$7:$ $\quad z := 10/w$

$8:$ $\quad z := 0$

$9:$ $\quad w := double(z)$

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

function $\textsc{ResultsFor}(ctx, \sigma_{in})$
   if $ctx \in \text{dom}(results)$ **then**
      if $\sigma_{in} \sqsubseteq results[ctx].input$ **then**
         return $results[ctx].output$
      else
         return $\textsc{Analyze}(ctx, results[ctx].input \sqcup \sigma_{in})$
      end if
   else
      return $\textsc{Analyze}(ctx, \sigma_{in})$
   end if
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 $\textsc{Flow}(\llbracket n\!: x := f(y) \rrbracket, ctx, \sigma_n)$
  $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$
  $calleeCtx \leftarrow \textsc{GetCtx}(f, ctx, n, \sigma_{in})$
  $\sigma_{out} \leftarrow \textsc{ResultsFor}(calleeCtx, \sigma_{in})$
  return $\sigma_n[x \mapsto \sigma_{out}[result]]$
end function

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

Recursion makes this a bit harder

```
int fact(int x) {
    if (x == 1)
        return 1;                              void main() {
    else                                           int y = fact(2);
        return x * fact(x-1);                       int z = fact(3);
}                                                   int w = fact(getInputFromUser());
                                               }
```

```
bar() { if (...) return 2 else return foo() }
foo() { if (...) return 1 else return bar() }

main() { foo(); }
```

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

function $\textsc{Analyze}(ctx, \sigma_{in})$
   $\sigma_{out} \leftarrow results[ctx].output$
   $\textsc{Add}(analyzing, ctx)$
   $\sigma'_{out} \leftarrow \textsc{Intraprocedural}(ctx, \sigma_{in})$
   $\textsc{Remove}(analyzing, ctx)$
   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**
         $\textsc{Add}(worklist, c)$
      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; otherwi
    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 Precision: Why return ⊥ when analyzing?

Exercise: Try running zero analysis on this program

```
int iterativeIdentity(x, y)
    if x <= 0
        return y
    else
        return iterativeIdentity(x-1, y)

void main(z)
    w = iterativeIdentity(z, 5)
```

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, σin)
    return Context(f)
end function
```

Value-based context-sensitivity

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

K-call-string context-sensitivity

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

function GETCTX(f, callingCtx, n, σin)
    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