

Lecture 10: Context-Sensitive Analysis

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

Rohan Padhye and Jonathan Aldrich

February 17, 2022

* Course materials developed with Jonathan Aldrich and Claire Le Goues

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	$\{x \rightarrow N\}$	$\{x \rightarrow N, y \rightarrow N\}$
Line 9	$\{x \rightarrow Z\}$	$\{x \rightarrow Z, y \rightarrow Z\}$

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* : σ

type *Summary*
val *input* : σ
val *output* : σ

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

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

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

```

```

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

```

```

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

```

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

```

```

function 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 ANALYZE( $ctx, results[ctx].input \sqcup \sigma_{in}$ )
    end if
  else
    return ANALYZE( $ctx, \sigma_{in}$ )
  end if
end function

```

Recursion makes this a bit harder

```
int fact(int x) {  
    if (x == 1)  
        return 1;  
    else  
        return x * fact(x-1);  
}
```

```
void main() {  
    int y = fact(2);  
    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  $\leftarrow$  GETCTX(main, nil, 0,  $\top$ )  
  worklist  $\leftarrow$  {initCtx}  
  results[initCtx]  $\leftarrow$  Summary( $\top$ ,  $\perp$ )  
  while NOTEMPTY(worklist) do  
    ctx  $\leftarrow$  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, T)
  worklist  $\leftarrow$  {initCtx}
  results[initCtx]  $\leftarrow$  Summary(T,  $\perp$ )
  while NOTEMPTY(worklist) do
    ctx  $\leftarrow$  REMOVE(worklist)
    ANALYZE(ctx, results[ctx].input)
  end while
end function
```

```
function ANALYZE(ctx,  $\sigma_{in}$ )
   $\sigma_{out} \leftarrow$  results[ctx].output
  ADD(analyzing, ctx)
   $\sigma'_{out} \leftarrow$  INTRAPROCEDURAL(ctx,  $\sigma_{in}$ )
  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  $\sigma'_{out}$ 
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( $\llbracket n: x := f(y) \rrbracket$ , ctx,  $\sigma_n$ )
   $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$ 
  calleeCtx  $\leftarrow$  GETCTX(f, ctx, n,  $\sigma_{in}$ )
   $\sigma_{out} \leftarrow$  RESULTSFOR(calleeCtx,  $\sigma_{in}$ )
  ADD(callers[calleeCtx], ctx)
  return  $\sigma_n[x \mapsto \sigma_{out}[result]]$ 
```

```
function ANALYZE(ctx,  $\sigma_{in}$ )
   $\sigma_{out} \leftarrow results[ctx].output$ 
  ADD(analyzing, ctx)
   $\sigma'_{out} \leftarrow$  INTRAPROCEDURAL(ctx,  $\sigma_{in}$ )
  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  $\sigma'_{out}$ 
end function
```

```

function RESULTSFOR( $ctx, \sigma_{in}$ )
  if  $ctx \in \text{dom}(\text{results})$  then
    if  $\sigma_{in} \sqsubseteq \text{results}[ctx].\text{input}$  then
      return  $\text{results}[ctx].\text{output}$   $\triangleright$  existing results are good
    else
       $\text{results}[ctx].\text{input} \leftarrow \text{results}[ctx].\text{input} \sqcup \sigma_{in}$   $\triangleright$  keep track of more general input
    end if
  else
     $\text{results}[ctx] = \text{Summary}(\sigma_{in}, \perp)$   $\triangleright$  initially optimistic
  end if
  if  $ctx \in \text{analyzing}$  then
    return  $\text{results}[ctx].\text{output}$   $\triangleright \perp$  if it hasn't been analyzed yet; otherwise
  else
    return  $\text{ANALYZE}(ctx, \text{results}[ctx].\text{input})$ 
  end if
end function

```

```

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

```

```

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

```

On Precision: Why return \perp 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 $C \times H$ times ($C = \text{\#contexts}$, $H = \text{lattice height}$)
- After each analysis, propagate result to N callers
- $O(C \times N \times H)$ intraprocedural analyses
- $= O(E \times H)$ 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,  $\sigma_{in}$ )  
  return Context(f)  
end function
```

Value-based context-sensitivity

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

K-call-string context-sensitivity

```
type Context  
  val fn : Function  
  val string : List[Int]  
  
function GETCTX(f, callingCtx, n,  $\sigma_{in}$ )  
  newStr  $\leftarrow$  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

OOP: Dynamic Dispatch

```
class A { A foo(A x) { return x; } }
class B extends A { A foo(A x) { return new D(); } }
class D extends A { A foo(A x) { return new A(); } }
class C extends A { A foo(A x) { return this; } }

// in main()
A x = new A();
while (...)
    x = x.foo(new B()); // may call A.foo, B.foo, or D.foo
A y = new C();
y.foo(x);                // only calls C.foo
```

OOP: Dynamic Dispatch

- Which function (method) is being called?
- Depends on what objects variables can *point to*
- Objects can be allocated on the heap
- Next up: Pointer Analysis