Lecture 10: Context-Sensitive Analysis

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

Rohan Padhye and Jonathan Aldrich

March 9, 2021

* Course materials developed with 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->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**
    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, σ_{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 GETCTX($f, callingCtx, n, \sigma_{in}$)
  $newStr \leftarrow callingCtx.string$ ++ n
  return $Context(f, newStr)$
end function

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**
    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, σ_{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

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

Key Idea: Worklist of Contexts

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

```
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
```

```
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 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, 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

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