Lecture 10: Context-Sensitive Analysis

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

Rohan Padhye

February 17, 2022

* Course materials developed with Jonathan Aldrich and Claire Le Goues

institute for SOFTWARE RESEARCH
Carnegie Mellon University
School of Computer Science

Context-Sensitive Analysis Example

```
fun double(x) : int
    y := 2 * x
    return y

fun main()
    z := 5
    w := double(z)
    z := 10/w
    z := 0
    w := double(z)
```

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

(“context” = some statically definable condition)

Context-Sensitive Analysis Example

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

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

$$
\begin{array}{l}
\text{fun } \mathit{double}(x) : \mathit{int} \\
\quad y := 2 * x \\
\quad \text{return } y \\
\\
\text{fun } \mathit{main}() \\
\quad z := 5 \\
\quad \boxed{w := \mathit{double}(z)} \\
\quad z := 10/w \\
\quad z := 0 \\
\quad \boxed{w := \mathit{double}(z)}
\end{array}
$$

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

val $results : Map[Context, Summary]$

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

$$\begin{array}{l}\textbf{type } \mathit{Context} \\ \quad \textbf{val } fn : \mathit{Function} \\ \quad \textbf{val } \mathit{string} : \mathit{List}[\mathit{Int}]\end{array}$$

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!

$$\begin{array}{l}\textbf{function } \textsc{GetCtx}(f, \mathit{callingCtx}, n, \sigma_{in}) \\ \quad \mathit{newStr} \leftarrow \mathit{callingCtx}.\mathit{string} \mathrel{++} n \\ \quad \textbf{return } \mathit{Context}(f, \mathit{newStr}) \\ \textbf{end function}\end{array}$$

$$\begin{array}{l}\textbf{type } \mathit{Summary} \\ \quad \textbf{val } \mathit{input} : \sigma \\ \quad \textbf{val } \mathit{output} : \sigma\end{array}$$

$$\textbf{val } \mathit{results} : \mathit{Map}[\mathit{Context}, \mathit{Summary}]$$

$$\begin{array}{l}\textbf{function } \textsc{Analyze}(ctx, \sigma_{in}) \\ \quad \sigma'_{out} \leftarrow \textsc{Intraprocedural}(ctx, \sigma_{in}) \\ \quad \mathit{results}[ctx] \leftarrow \mathit{Summary}(\sigma_{in}, \sigma'_{out}) \\ \quad \textbf{return } \sigma'_{out} \\ \textbf{end function}\end{array}$$

$$\begin{array}{l}\textbf{function } \textsc{Flow}([\![n\colon x := f(y)]\!], ctx, \sigma_n) \\ \quad \sigma_{in} \leftarrow [\mathit{formal}(f) \mapsto \sigma_n(y)] \\ \quad \mathit{calleeCtx} \leftarrow \textsc{GetCtx}(f, ctx, n, \sigma_{in}) \\ \quad \sigma_{out} \leftarrow \textsc{ResultsFor}(\mathit{calleeCtx}, \sigma_{in}) \\ \quad \textbf{return } \sigma_n[x \mapsto \sigma_{out}[\mathit{result}]] \\ \textbf{end function}\end{array}$$

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

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

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

Key Idea: Worklist of Contexts

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

function ANALYZEPROGRAM
    $initCtx \leftarrow \textsc{GetCtx}(\texttt{main}, \textbf{nil}, 0, \top)$
    $worklist \leftarrow \{initCtx\}$
    $results[initCtx] \leftarrow Summary(\top, \bot)$
    while $\textsc{NotEmpty}(worklist)$ **do**
        $ctx \leftarrow \textsc{Remove}(worklist)$
        $\textsc{Analyze}(ctx, results[ctx].input)$
    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 $\textsc{Flow}([\![n\colon x := f(y)]\!], ctx, \sigma_n)$
    $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$ $\triangleright$
    $calleeCtx \leftarrow \textsc{GetCtx}(f, ctx, n, \sigma_{in})$
    $\sigma_{out} \leftarrow \textsc{ResultsFor}(calleeCtx, \sigma_{in})$
    $\boxed{\textsc{Add}(callers[calleeCtx], ctx)}$
    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 $\textsc{ResultsFor}(ctx, \sigma_{in})$
  if $ctx \in \mathrm{dom}(results)$ **then**
    if $\sigma_{in} \sqsubseteq results[ctx].input$ **then**
      return $results[ctx].output$ ▷ existing results are good
    else
      $results[ctx].input \leftarrow results[ctx].input \sqcup \sigma_{in}$ ▷ keep track of more general input
    end if
  else
    $results[ctx] \leftarrow \boxed{Summary(\sigma_{in}, \bot)}$ ▷ initially optimisti
  end if
  if $ctx \in analyzing$ **then**
    return $results[ctx].output$ ▷ $\bot$ if it hasn't been analyzed yet; otherwis
  else
    return $\textsc{Analyze}(ctx, results[ctx].input)$
  end if
end function

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

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

Value-based context-sensitivity

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

K-call-string context-sensitivity

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

function $\textsc{GetCtx}(f, callingCtx, n, \sigma_{in})$
  $newStr \leftarrow \textsc{Suffix}(callingCtx.string +\!+\ n, \text{CALL\_STRING\_CUTOFF})$
  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