Lecture 9: Interprocedural Analysis

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

Rohan Padhye and Jonathan Aldrich

March 4, 2021

* Course materials developed with Claire Le Goues

Extend WHILE with functions

Extend WHILE3ADDR with functions

$$
\begin{array}{lcl}
F & ::= & \textbf{fun}\ f(x)\ \{\ \overline{n : I}\ \} \\
I & ::= & \ldots \mid \textbf{return}\ x \mid y := f(x)
\end{array}
$$

Extend WHILE3ADDR with functions

$$
\begin{array}{lcl}
F & ::= & \textbf{fun}\ f(x)\ \{\ \overline{n : I}\ \} \\
I & ::= & \ldots \mid \textbf{return}\ x \mid y := f(x)
\end{array}
$$

$$
\begin{array}{ll}
1: & \textbf{fun}\ double(x) : int \\
2: & \quad y := 2 * x \\
3: & \quad \textbf{return}\ y \\
4: & \textbf{fun}\ main() : void \\
5: & \quad z := 0 \\
6: & \quad w := double(z)
\end{array}
$$

Extend WHILE3ADDR with functions

```
1 :  fun divByX(x) : int
2 :      y := 10/x
3 :      return y

4 :  fun main() : void
5 :      z := 5
6 :      w := divByX(z)
```

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

4 :  fun main() : void
5 :      z := 0
6 :      w := double(z)
```

Data-Flow Analysis

HOW DO WE ANALYZE THESE PROGRAMS?

Approach #1: Analyze functions independently

- Pretend function *f()* cannot see the source of function *g()*
- Simulates separate compilation and dynamic linking (e.g. C, Java)
- Create CFG for each function body and run **intraprocedural** analysis
- **Q**: What should be is σ_0 and$f_Z[\![x := g(y)]\!]$ and $f_Z[\![\text{return } x]\!]$ for zero analysis?

$$\sigma_0 =$$

$$f[\![x := g(y)]\!](\sigma) =$$

$$f[\![\textbf{return } x]\!](\sigma) =$$

Can we show that division on line 2 is safe?

```
1 :  fun divByX(x) : int
2 :      y := 10/x
3 :      return y

4 :  fun main() : void
5 :      z := 5
6 :      w := divByX(z)
```

Approach #2: User-defined Annotations

@NonZero -> @NonZero

$$
\begin{array}{ll}
1: & \text{fun } divByX(x) : int \\
2: & \quad y := 10/x \\
3: & \quad \text{return } y \\
4: & \text{fun } main() : void \\
5: & \quad z := 5 \\
6: & \quad w := divByX(z)
\end{array}
$$

$$
\begin{array}{lll}
f[\![x := g(y)]\!](\sigma) & = \sigma[x \mapsto annot[\![g]\!].r] & (\text{error if } \sigma(y) \notin annot[\![g]\!].a) \\
f[\![\text{return } x]\!](\sigma) & = \sigma & (\text{error if } \sigma(x) \notin annot[\![g]\!].r)
\end{array}
$$

Approach #2: User-defined Annotations

@NonZero -> @NonZero

```
1 :  fun divByX(x) : int
2 :      y := 10/x
3 :      return y

4 :  fun main() : void
5 :      z := 5
6 :      w := divByX(z)
```

@NonZero -> @NonZero

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

4 :  fun main() : void
5 :      z := 0
6 :      w := double(z)  Error!
```

$$f[\![x := g(y)]\!](\sigma) = \sigma[x \mapsto annot[\![g]\!].r] \quad (\text{error if } \sigma(y) \notin annot[\![g]\!].a)$$

$$f[\![\text{return } x]\!](\sigma) = \sigma \quad (\text{error if } \sigma(x) \notin annot[\![g]\!].r)$$

Approach #2: User-defined Annotations

@NonZero -> @NonZero

```
1 :  fun divByX(x) : int
2 :      y := 10/x
3 :      return y

4 :  fun main() : void
5 :      z := 5
6 :      w := divByX(z)
```

@Any -> @NonZero

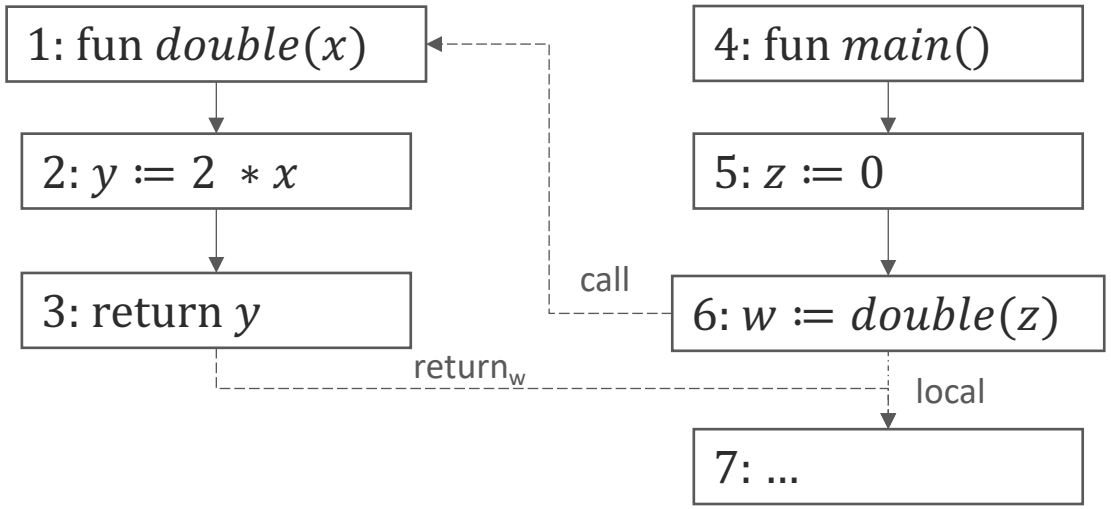
```
1 :  fun double(x) : int
2 :      y := 2 * x
3 :      return y      Error!

4 :  fun main() : void
5 :      z := 0
6 :      w := double(z)
```

$$f[\![x := g(y)]\!](\sigma) = \sigma[x \mapsto annot[\![g]\!].r] \quad (\text{error if } \sigma(y) \notin annot[\![g]\!].a)$$

$$f[\![\text{return } x]\!](\sigma) = \sigma \quad (\text{error if } \sigma(x) \notin annot[\![g]\!].r)$$

Approach #3: Interprocedural CFG



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

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

$$f_Z[\![x := g(y)]\!]_{call}(\sigma) = \{formal(g) \to \sigma(y)\}$$

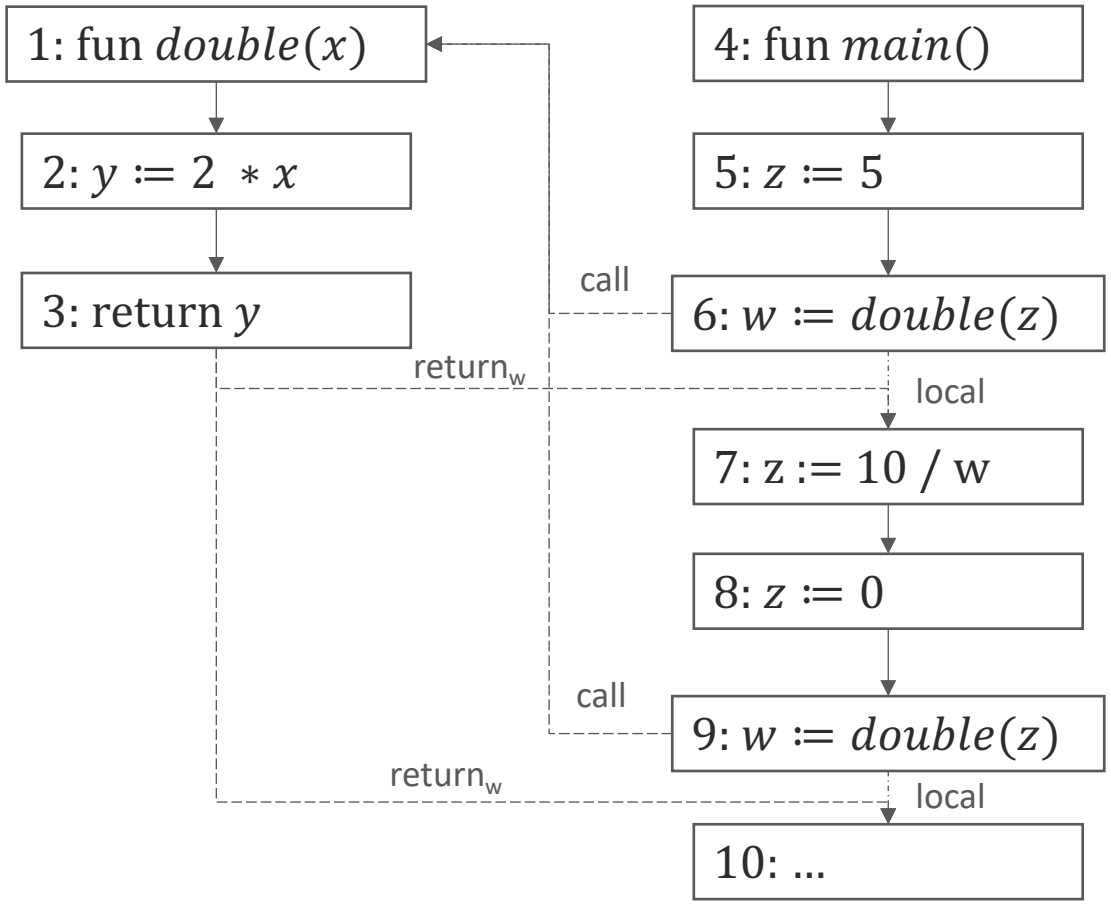
$$f_Z[\![\text{return } x]\!]_{ret}(\sigma) = \{z \to \sigma(z) \mid z \in Globals\} \cup \{ret \to \sigma(x)\}$$

Approach #3: Interprocedural CFG

Exercise: What would be the result of zero analysis for this program on line 7 and at the end?

```
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)
```

Approach #3: Interprocedural CFG




```
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)
```

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?

Enter:

CONTEXT-SENSITIVE ANALYSIS

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

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());
}
```

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
    worklist ← {Context(main, ⊤)}
    results[Context(main, ⊤)].input ← ⊤
    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
  $worklist \leftarrow \{Context(\texttt{main}, \top)\}$
  $results[Context(\texttt{main}, \top)].input \leftarrow \top$
  while NOTEMPTY($worklist$) **do**
    $ctx \leftarrow$ REMOVE($worklist$)
    ANALYZE($ctx, results[ctx].input$)
  end while
end function

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

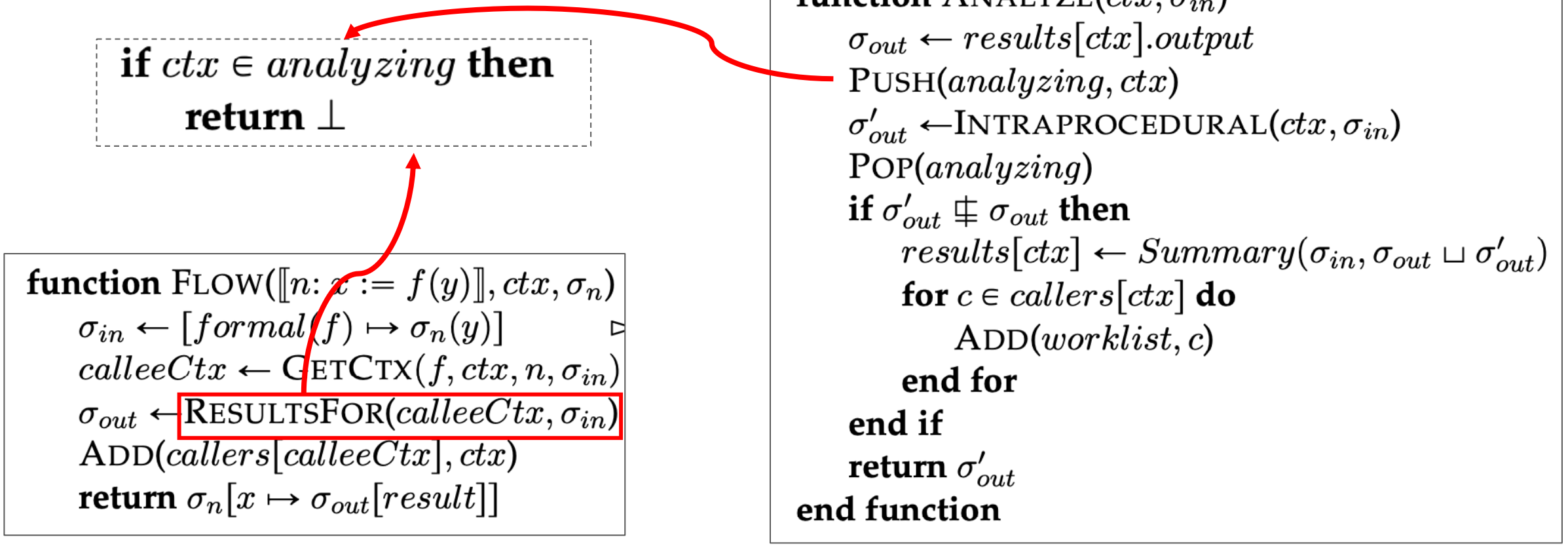
Key Idea: Worklist of Contexts

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

function FLOW($[\![n\colon x := f(y)]\!], ctx, \sigma_n$)
  $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$ ▷
  $calleeCtx \leftarrow$ 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})
  $\sigma_{out} \leftarrow results[ctx].output$
  PUSH$(analyzing, ctx)$
  $\sigma'_{out} \leftarrow$INTRAPROCEDURAL(ctx, σ_{in})
  POP$(analyzing)$
  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

Key Idea: Worklist of Contexts



On Precision: Why return ⊥ when analyzing?

Exercise: Try running constant propagation on this program

```
int iterativeIdentity(x, y)
    if x <= 0
        return y
    else
        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
- Limited call strings
- 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
```