

# Static Program Analysis

## Part 7 – interprocedural analysis

<http://cs.au.dk/~amoeller/spa/>

Anders Møller & Michael I. Schwartzbach  
Computer Science, Aarhus University

# Interprocedural analysis

- Analyzing the body of a single function:
  - *intra*procedural analysis
- Analyzing the whole program with function calls:
  - *inter*procedural analysis
- For now, we consider TIP without functions as first-class values (so we only have direct calls)
- A naive approach:
  - analyze each function in isolation
  - be maximally pessimistic about results of function calls
  - rarely sufficient precision...

# CFG for whole programs

The idea:

- construct a CFG for each function
- then glue them together to reflect function calls and returns

We need to take care of:

- parameter passing
- return values
- values of local variables across calls  
(including recursive functions, so not enough to assume unique variable names)

# A simplifying assumption

- Assume that all function calls are of the form

$$X = f(E_1, \dots, E_n);$$

- This can always be obtained by normalization

# Interprocedural CFGs (1/3)

Split each original call node

$$X = f(E_1, \dots, E_n)$$

into two nodes:

$$\square = f(E_1, \dots, E_n)$$

$$X = \square$$

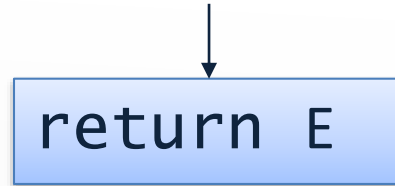
← the “call node”

← the “after-call node”

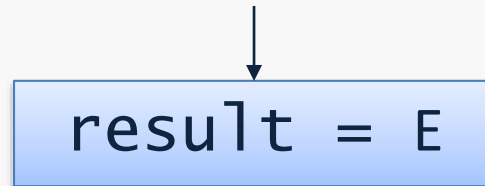
a special edge that  
connects the call node  
with its after-call node

# Interprocedural CFGs (2/3)

Change each return node



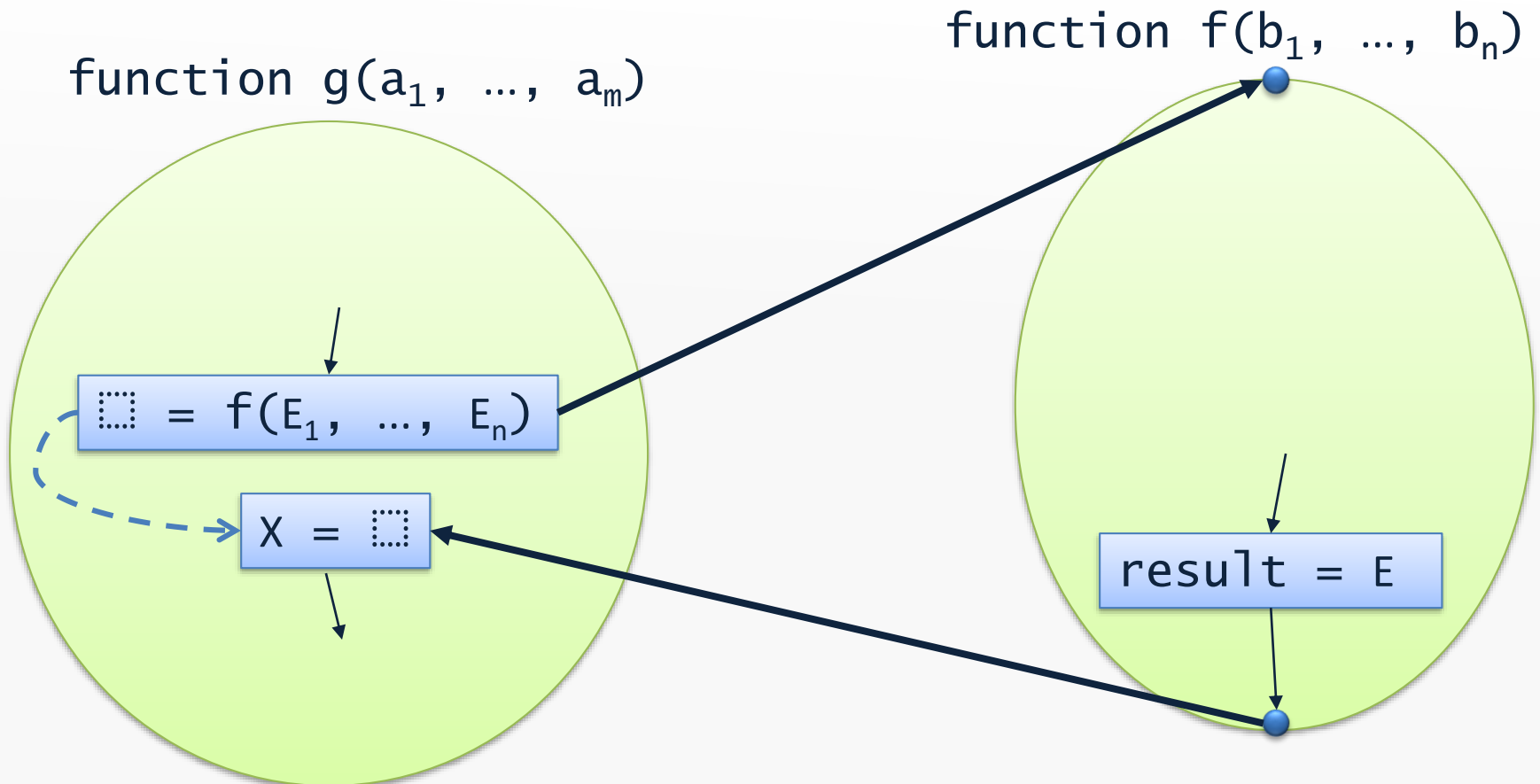
into an assignment:



(where `result` is a fresh variable)

# Interprocedural CFGs (3/3)

Add call edges and return edges:



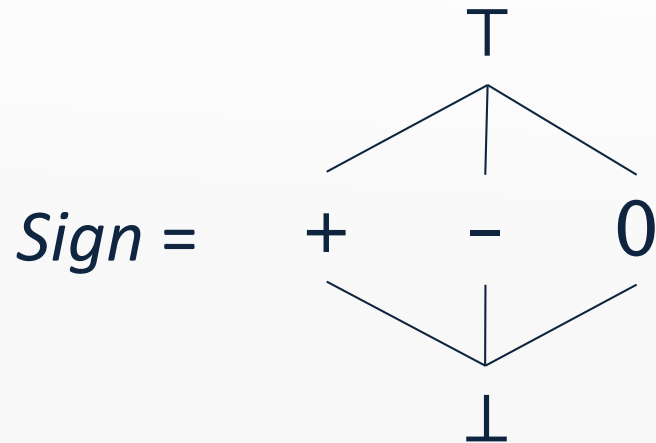
# Constraints

- For call/entry nodes:
  - be careful to model evaluation of *all* the actual parameters before binding them to the formal parameter names (otherwise, it may fail for recursive functions)
- For after-call/exit nodes:
  - like an assignment: `X = result`
  - but also restore local variables from before the call using the `call`  $\leadsto$  `after-call` edge
- The details depend on the specific analysis...



# Example: interprocedural sign analysis

- Recall the intraprocedural sign analysis...
- Lattice for abstract values:



- Lattice for abstract states:  
 $Vars \rightarrow Sign$

# Example: interprocedural sign analysis

- Constraint for entry node  $v$  of function  $f(b_1, \dots, b_n)$ :

$$\llbracket v \rrbracket = \bigsqcup_{w \in \text{pred}(v)} \perp [b_1 \rightarrow \text{eval}(\llbracket w \rrbracket, E_1^w), \dots, b_n \rightarrow \text{eval}(\llbracket w \rrbracket, E_n^w)]$$

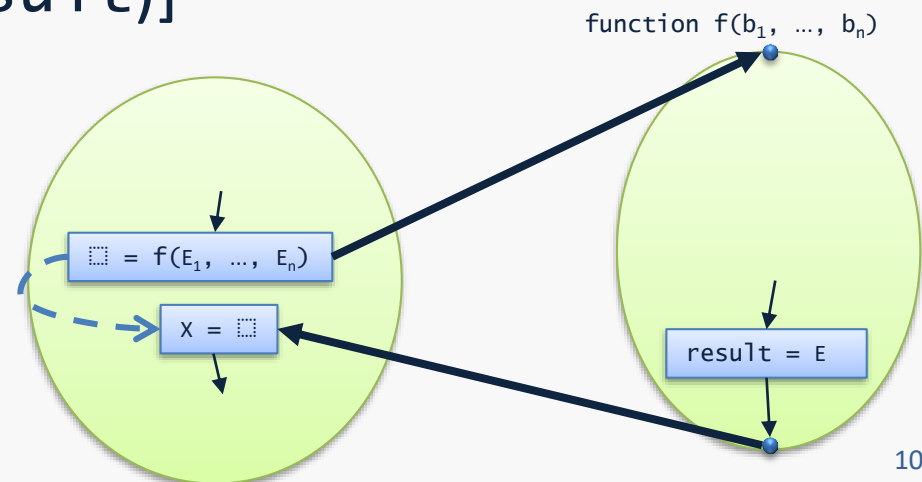
where  $E_i^w$  is  $i$ 'th argument at  $w$

- Constraint for after-call node  $v$  labeled  $X = \square$ , with call node  $v'$ :

$$\llbracket v \rrbracket = \llbracket v' \rrbracket [X \rightarrow \llbracket w \rrbracket(\text{result})]$$

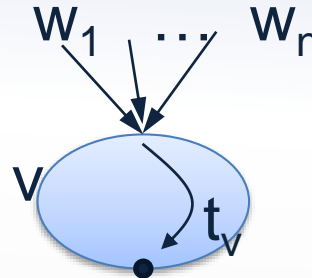
where  $w \in \text{pred}(v)$

(Recall: no global variables, no heap, and no higher-order functions)



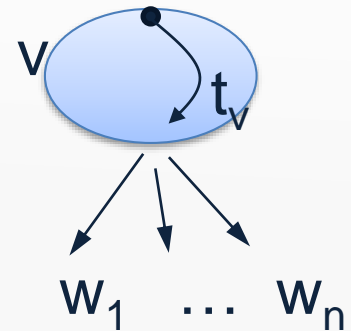
# Alternative formulations

1) 
$$\llbracket v \rrbracket = t_v(\bigsqcup_{w \in \text{pred}(v)} \llbracket w \rrbracket)$$



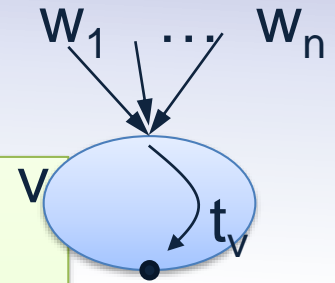
2) 
$$\forall w \in \text{succ}(v): t_v(\llbracket v \rrbracket) \sqsubseteq \llbracket w \rrbracket$$

- recall “solving inequations”
- may require fewer join operations if there are many CFG edges
- more suitable for *interprocedural* flow



# The worklist algorithm (original version)

```
x1 = ⊥; ... xn = ⊥  
W = {v1, ..., vn}  
while (W ≠ ∅) {  
    vi = W.removeNext()  
    y = fi(x1, ..., xn)  
    if (y ≠ xi) {  
        for (vj ∈ dep(vi)) {  
            W.add(vj)  
        }  
        xi = y  
    }  
}
```

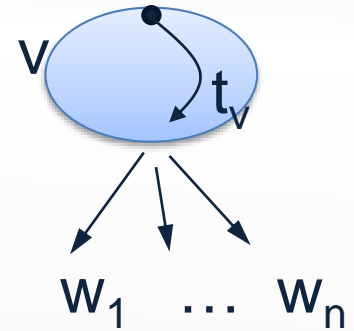


# The worklist algorithm (alternative version)

```

 $x_1 = \perp; \dots x_n = \perp$ 
 $W = \{v_1, \dots, v_n\}$ 
while ( $W \neq \emptyset$ ) {
     $v_i = W.removeNext()$ 
     $y = t_i(x_i)$ 
    for ( $v_j \in dep(v_i)$ ) {
        propagate( $y, v_j$ )
    }
}

```



```

propagate( $y, v_j$ ) {
     $z = x_j \sqcup y$ 
    if ( $z \neq x_j$ ) {
         $x_j = z$ 
         $W.add(v_j)$ 
    }
}

```

Implementation: `worklistFixpointPropagationSolver`

# Agenda

- Interprocedural analysis
- **Context-sensitive  
interprocedural analysis**

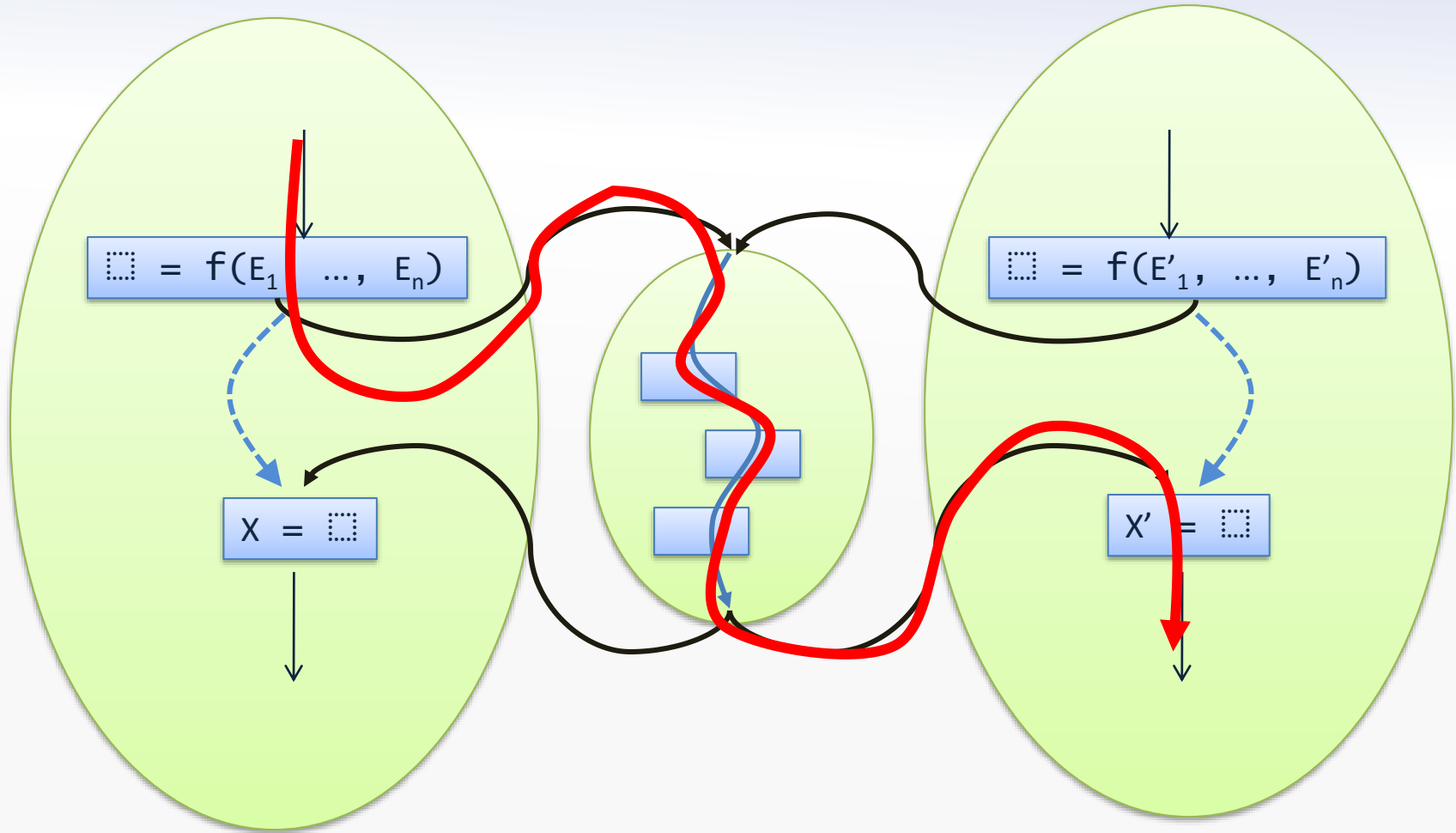
# Motivating example

What is the sign of the return value of g?

```
f(z) {  
    return z*42;  
}  
  
g() {  
    var x,y;  
    x = f(0);  
    y = f(87);  
    return x + y;  
}
```

Our current analysis says “T”

# Interprocedurally invalid paths





# Function cloning

## (alternatively, function inlining)

- Clone functions such that each function has only one callee
- Can avoid interprocedurally invalid paths 😊
- For high nesting depths, gives exponential blow-up 😐
- Doesn't work on (mutually) recursive functions 😞
- Use heuristics to determine when to apply (trade-off between CFG size and precision)

# Example, with cloning

What is the sign of the return value of g?

```
f1(z1) {  
    return z1*42;  
}
```

```
f2(z2) {  
    return z2*42;  
}
```

```
g() {  
    var x,y;  
    x = f1(0);  
    y = f2(87);  
    return x + y;  
}
```

# Context sensitive analysis

- Function cloning provides a kind of context sensitivity (also called polyvariant analysis)
- Instead of physically copying the function CFGs, do it *logically*
- Replace the lattice for abstract states, States, by

Contexts  $\rightarrow$  lift(States)

where Contexts is a set of ***call contexts***

- the contexts are abstractions of the state at function entry
  - Contexts must be finite to ensure finite height of the lattice
  - the bottom element of lift(States) represents “unreachable” contexts
- Different strategies for choosing the set Contexts...

# Constraints for CFG nodes that do not involve function calls and returns

Easily adjusted to the new lattice Contexts  $\rightarrow$  lift(States)

Example if  $v$  is an assignment node  $x = E$  in sign analysis:

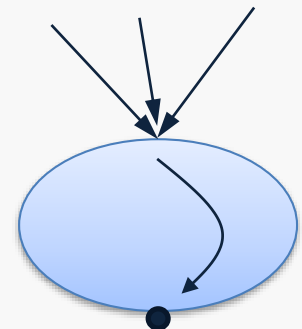
$$\llbracket v \rrbracket = JOIN(v)[x \mapsto eval(JOIN(v), E)]$$

becomes

$$\llbracket v \rrbracket(c) = \begin{cases} s[x \mapsto eval(s, E)] & \text{if } s = JOIN(v, c) \in \text{States} \\ \text{unreachable} & \text{if } JOIN(v, c) = \text{unreachable} \end{cases}$$

$$\text{and } JOIN(v) = \bigsqcup_{w \in pred(v)} \llbracket w \rrbracket$$

$$\text{becomes } JOIN(v, c) = \bigsqcup_{w \in pred(v)} \llbracket w \rrbracket(c)$$



# One-level cloning

- Let  $c_1, \dots, c_n$  be the call nodes in the program
- Define  $\text{Contexts} = \{c_1, \dots, c_n\} \cup \{\varepsilon\}$ 
  - each call node now defines its own “call context”  
(using  $\varepsilon$  to represent the call context at the main function)
  - the context is then like the return address of the top-most stack frame in the call stack
- Same effect as one-level cloning, but without actually copying the function CFGs
- Usually straightforward to generalize the constraints for a context insensitive analysis to this lattice
- (Example: context-sensitive sign analysis – later...)

# The call string approach

- Let  $c_1, \dots, c_n$  be the call nodes in the program
- Define Contexts as the set of strings over  $\{c_1, \dots, c_n\}$  of length  $\leq k$ 
  - such a string represents the top-most  $k$  call locations on the call stack
  - the empty string  $\varepsilon$  again represents the initial call context at the main function
- For  $k=1$  this amounts to one-level cloning

Implementation: CallStringSignAnalysis

# Example:

## interprocedural sign analysis with call strings ( $k=1$ )

Lattice for abstract states:  $\text{Contexts} \rightarrow \text{lift}(\text{Vars} \rightarrow \text{Sign})$   
where  $\text{Contexts} = \{\varepsilon, c_1, c_2\}$

```
f(z) {  
  var t1, t2;  
  t1 = z*6;  
  t2 = t1*7;  
  return t2;  
}  
...  
x = f(0); // c1  
y = f(87); // c2  
...
```

$[\varepsilon \mapsto \text{unreachable},$   
 $c_1 \mapsto \perp[z \mapsto 0, t_1 \mapsto 0, t_2 \mapsto 0],$   
 $c_2 \mapsto \perp[z \mapsto +, t_1 \mapsto +, t_2 \mapsto +]]$

What is an example program  
that requires  **$k=2$**   
to avoid loss of precision?

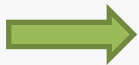
# Context sensitivity with call strings

## function entry nodes, for $k=1$

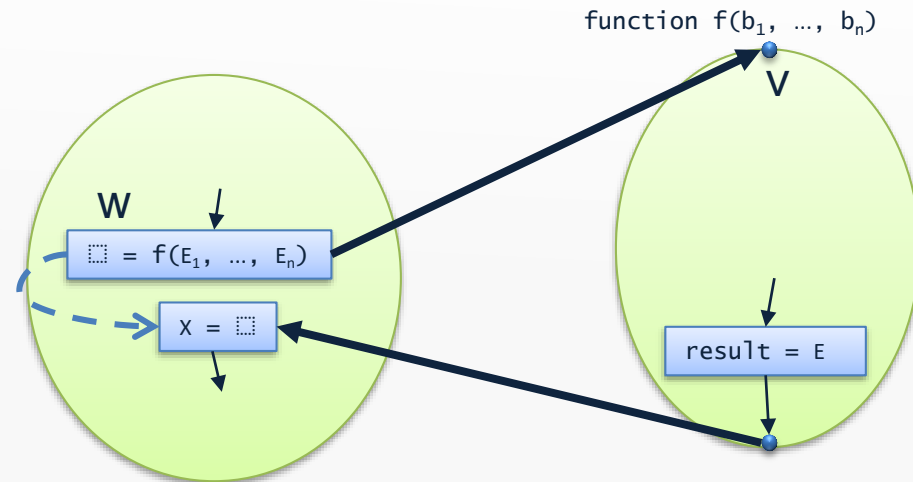
Constraint for entry node  $v$  of function  $f(b_1, \dots, b_n)$ :  
(if not 'main')

$$\llbracket v \rrbracket(c) = \bigsqcup_{w \in \text{pred}(v) \wedge c = w \wedge c' \in \text{Contexts}} S_w^{c'}$$

only consider  
the call node  $w$   
that matches  
the context  $c$



$$w \in \text{pred}(v) \wedge c = w \wedge c' \in \text{Contexts}$$



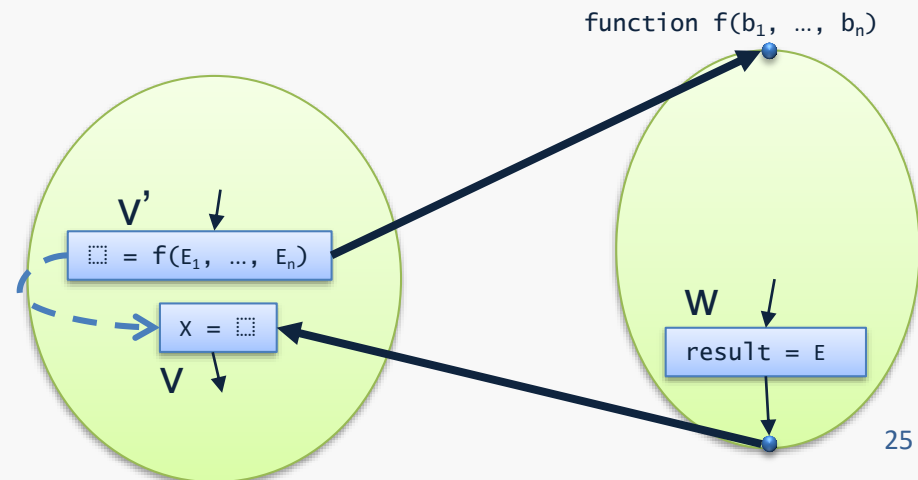
$$S_w^{c'} = \begin{cases} \text{unreachable} & \text{if } \llbracket w \rrbracket(c') = \text{unreachable} \\ \perp[b_1 \rightarrow \text{eval}(\llbracket w \rrbracket(c'), E_1^w), \dots, b_n \rightarrow \text{eval}(\llbracket w \rrbracket(c'), E_n^w)] & \text{otherwise} \end{cases}$$



# Context sensitivity with call strings after-call nodes, for $k=1$

Constraint for after-call node  $v$  labeled  $X = \square$ ,  
with call node  $v'$  and exit node  $w \in \text{pred}(v)$ :

$$\llbracket v \rrbracket(c) = \begin{cases} \text{unreachable} & \text{if } \llbracket v' \rrbracket(c) = \text{unreachable} \vee \llbracket w \rrbracket(v') = \text{unreachable} \\ \llbracket v' \rrbracket(c)[X \rightarrow \llbracket w \rrbracket(v')(\text{result})] & \text{otherwise} \end{cases}$$



# The functional approach

- The call string approach considers *control flow*
  - but why distinguish between two different call sites if their abstract states are the same?
- The functional approach instead considers *data*
- In the most general form, choose
$$\text{Contexts} = \text{States}$$
(requires States to be finite)
- Each element of the lattice  $\text{States} \rightarrow \text{lift}(\text{States})$  is now a map  $m$  that provides an element  $m(x)$  from States (or “unreachable”) for each possible  $x$  where  $x$  describes the state at function entry

# Example:

## interprocedural sign analysis with the functional approach

Lattice for abstract states:  $\text{Contexts} \rightarrow \text{lift}(\text{Vars} \rightarrow \text{Sign})$

where  $\text{Contexts} = \text{Vars} \rightarrow \text{Sign}$

```
f(z) {  
  var t1, t2;  
  t1 = z*6;  
  t2 = t1*7;  
  return t2;  
}
```

...

```
x = f(0);
```

```
y = f(87);
```

...

$\left[ \begin{array}{l} \perp[z \mapsto 0] \mapsto \perp[z \mapsto 0, t1 \mapsto 0, t2 \mapsto 0], \\ \perp[z \mapsto +] \mapsto \perp[z \mapsto +, t1 \mapsto +, t2 \mapsto +], \\ \text{all other contexts} \mapsto \text{unreachable} \end{array} \right]$


# Another example:

## interprocedural sign analysis with the functional approach

Lattice for abstract states:  $\text{Contexts} \rightarrow \text{lift}(\text{Vars} \rightarrow \text{Sign})$

where  $\text{Contexts} = \text{Vars} \rightarrow \text{Sign}$

```
f(z) {  
  var t1, t2;  
  t1 = z*6;  
  t2 = t1*7;  
  return t2;  
}  
g(a) {  
  return f(a);  
}  
...  
x = g(0);  
y = g(87);
```



$$\begin{aligned} &[\perp[z \mapsto 0] \mapsto \perp[z \mapsto 0, t1 \mapsto 0, t2 \mapsto 0], \\ &\perp[z \mapsto +] \mapsto \perp[z \mapsto +, t1 \mapsto +, t2 \mapsto +], \\ &\text{all other contexts} \mapsto \text{unreachable} ] \end{aligned}$$

# The functional approach

- The lattice element for a function exit node is thus a ***function summary*** that maps abstract function input to abstract function output
- This can be exploited at call nodes!
- When entering a function with abstract state  $x$ :
  - consider the function summary  $s$  for that function
  - if  $s(x)$  already has been computed, use that to model the entire function body, then proceed directly to the after-call node
- Avoids the problem with interprocedurally invalid paths!
- ...but may be expensive if States is large

Implementation: FunctionalSignAnalysis

# Example:

## interprocedural sign analysis with the functional approach

Lattice for abstract states:  $\text{Contexts} \rightarrow \text{lift}(\text{Vars} \rightarrow \text{Sign})$

where  $\text{Contexts} = \text{Vars} \rightarrow \text{Sign}$

```
f(z) {  
  var t1, t2;  
  t1 = z*6;  
  t2 = t1*7;  
  return t2;  
}
```

...

```
x = f(0);  
y = f(87);  
z = f(42);
```

...

The abstract state at the exit of  $f$  can be used as a function summary

$[ \perp[z \mapsto 0] \mapsto \perp[z \mapsto 0, t1 \mapsto 0, t2 \mapsto 0, \text{result} \mapsto 0],$   
 $\perp[z \mapsto +] \mapsto \perp[z \mapsto +, t1 \mapsto +, t2 \mapsto +, \text{result} \mapsto +],$   
all other contexts  $\mapsto \text{unreachable} ]$


At this call, we can reuse the already computed exit abstract state of  $f$  for the context  $\perp[z \mapsto +]$

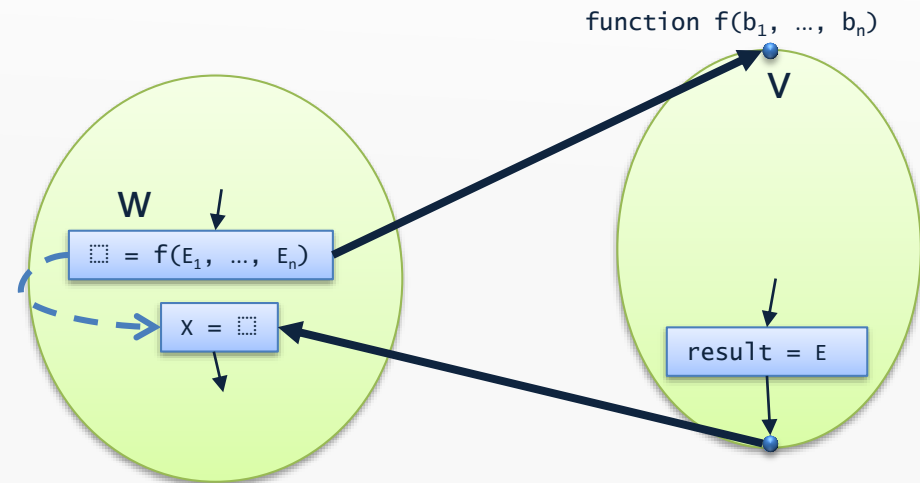
# Context sensitivity with the functional approach

## function entry nodes

Constraint for entry node  $v$  of function  $f(b_1, \dots, b_n)$ :  
(if not 'main')

$$\llbracket v \rrbracket(c) = \bigsqcup_{w \in \text{pred}(v) \wedge c = s_w^{c'} \wedge c' \in \text{Contexts}} s_w^{c'}$$

only consider  the call node  $w$   
if the abstract state  
from that node  
matches the context  $c$



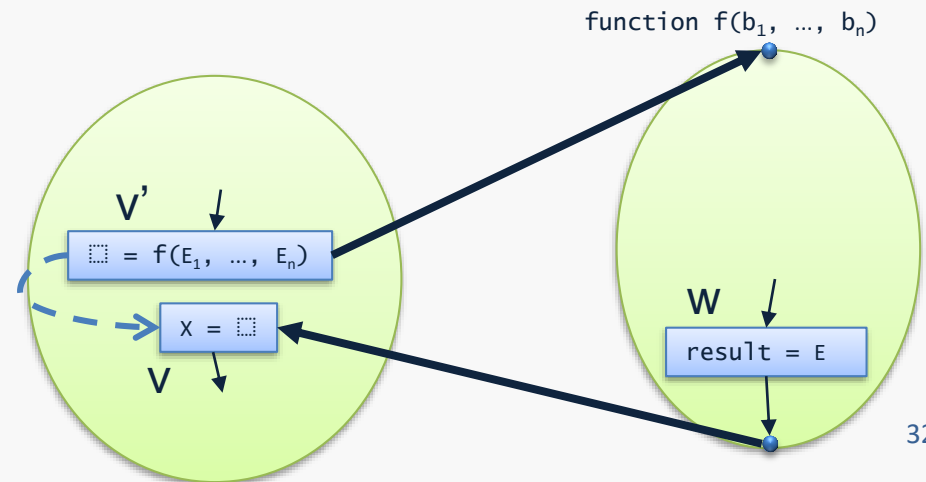
where  $s_w^{c'}$  is defined as before

# Context sensitivity with the functional approach

## after-call nodes

Constraint for after-call node  $v$  labeled  $X = \square$ ,  
with call node  $v'$  and exit node  $w \in \text{pred}(v)$ :

$$\llbracket v \rrbracket(c) = \begin{cases} \text{unreachable} & \text{if } \llbracket v' \rrbracket(c) = \text{unreachable} \vee \llbracket w \rrbracket(s_{v'}^c) = \text{unreachable} \\ \llbracket v' \rrbracket(c)[X \rightarrow \llbracket w \rrbracket(s_{v'}^c)(\text{result})] & \text{otherwise} \end{cases}$$





# Choosing the right context sensitivity strategy

- The call string approach is expensive for  $k > 1$ 
  - solution: choose  $k$  adaptively for each call site
- The functional approach is expensive if States is large
  - solution: only consider selected parts of the abstract state as context, for example abstract information about the function parameter values (called *parameter sensitivity*), or, in object-oriented languages, abstract information about the receiver object 'this' (called *object sensitivity* or *type sensitivity*)