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 function pointers and indirect 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

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

- This can always be obtained by normalization

Interprocedural CFGs (1/3)

Split each original call node

$$X = f(E_1, ..., E_n)$$

into two nodes:

$$\square = f(E_1, ..., E_n)$$ ← the "call node"

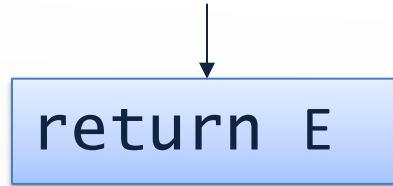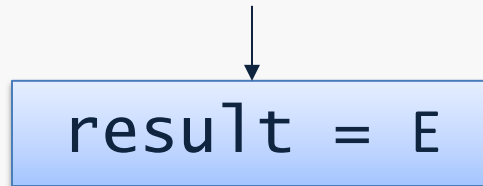$$X = \square$$ ← the "after-call node"

a special edge that connects the call node with its after-call node
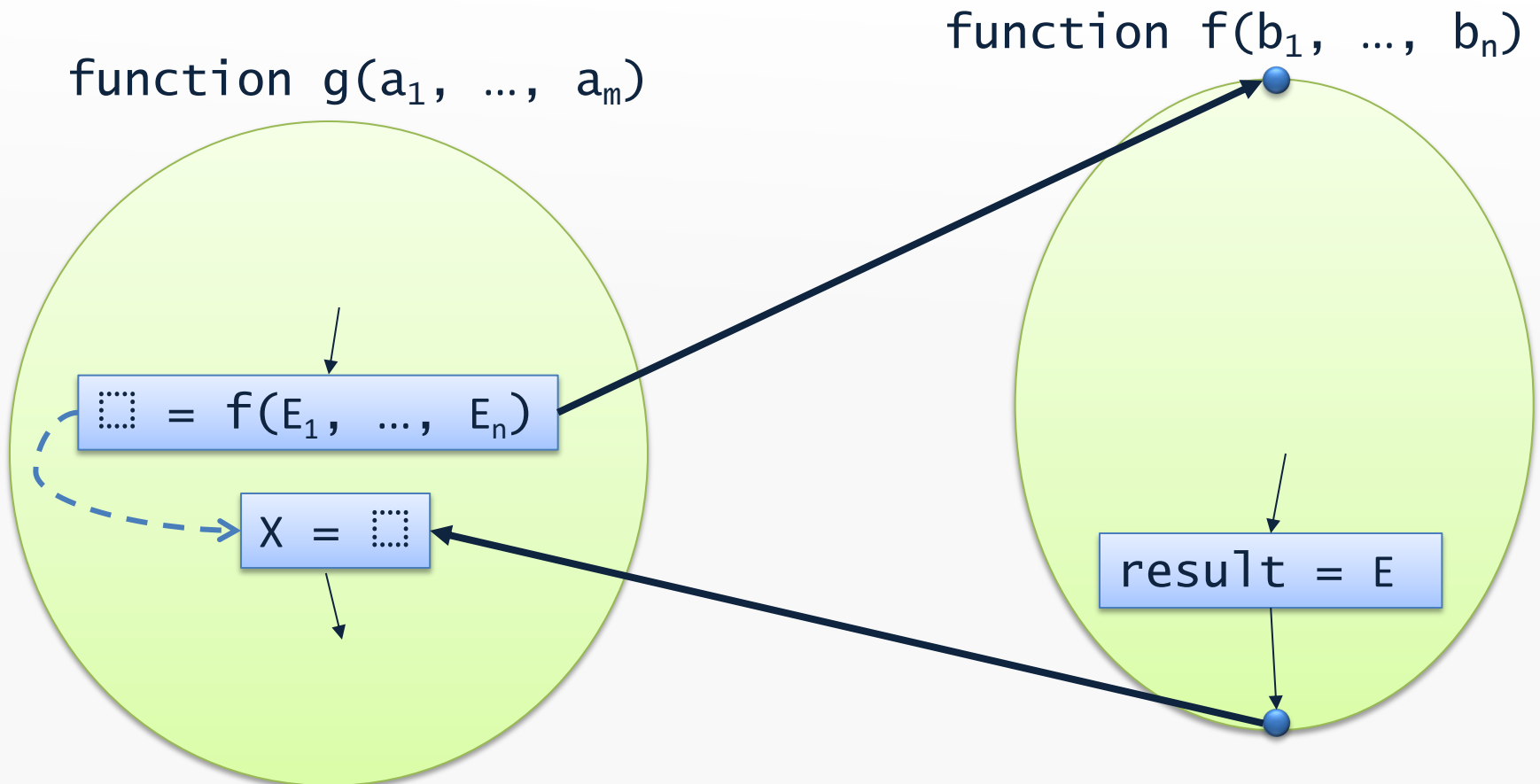
Interprocedural CFGs (2/3)

Change each return node

```
return E
```

into an assignment:

```
result = E
```

(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

- For after-call/exit nodes:
  - like an assignment: X = `result`
  - but also restore local variables from before the call using the call⌢after-call edge

- The details depend on the specific analysis…

Example: interprocedural sign analysis

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

$$Sign = \quad \begin{array}{c} \top \\ \diagup \mid \diagdown \\ + \quad - \quad 0 \\ \diagdown \mid \diagup \\ \bot \end{array}$$
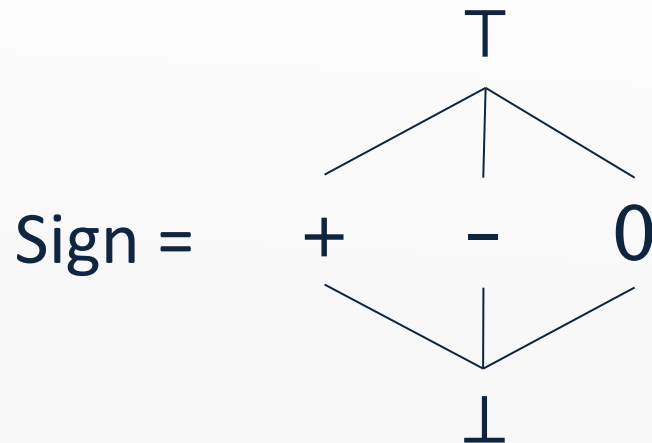
- Lattice for abstract states:
    $Vars \rightarrow Sign$

Example: interprocedural sign analysis

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

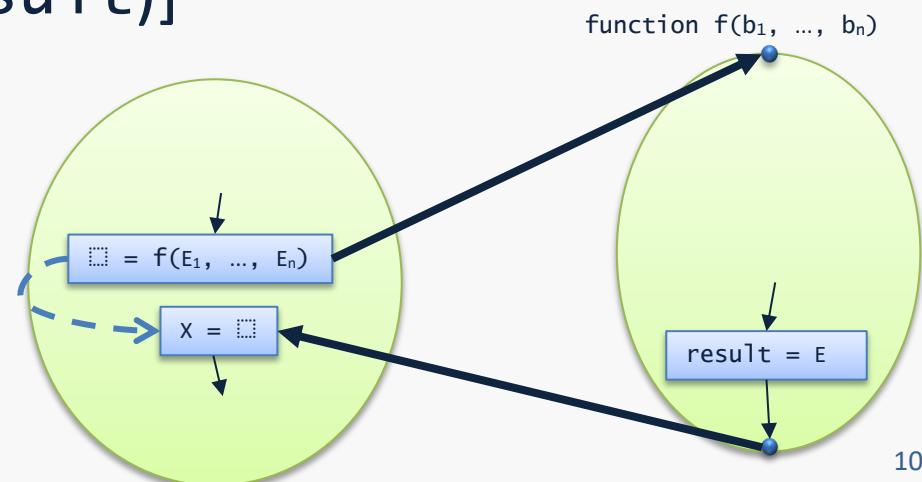$$[\![v]\!] = \bigsqcup_{w \in \text{pred}(v)} \bot[b_1 \rightarrow eval([\![w]\!], E_1^w), \ldots, b_n \rightarrow eval([\![w]\!], E_n^w)]$$

where E_i^w is i'th argument at w

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

$$[\![v]\!] = [\![v']\!][X \rightarrow [\![w]\!](\texttt{result})]$$

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

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

`function f(b₁, …, bₙ)`

`□ = f(E₁, …, Eₙ)`

`X = □`

`result = E`

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
  }
}
```
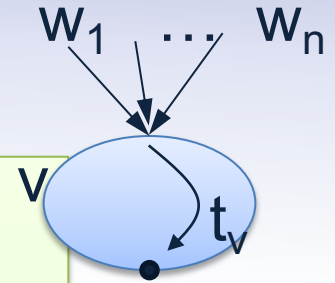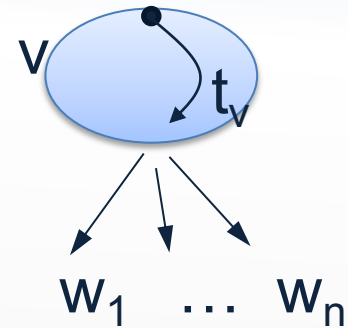
The worklist algorithm (alternative version)

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



```
propagate(y,vⱼ) {
  z = xⱼ ⊔ y
  if (z≠xⱼ) {
    xⱼ = z
    W.add(vⱼ)
  }
}
```
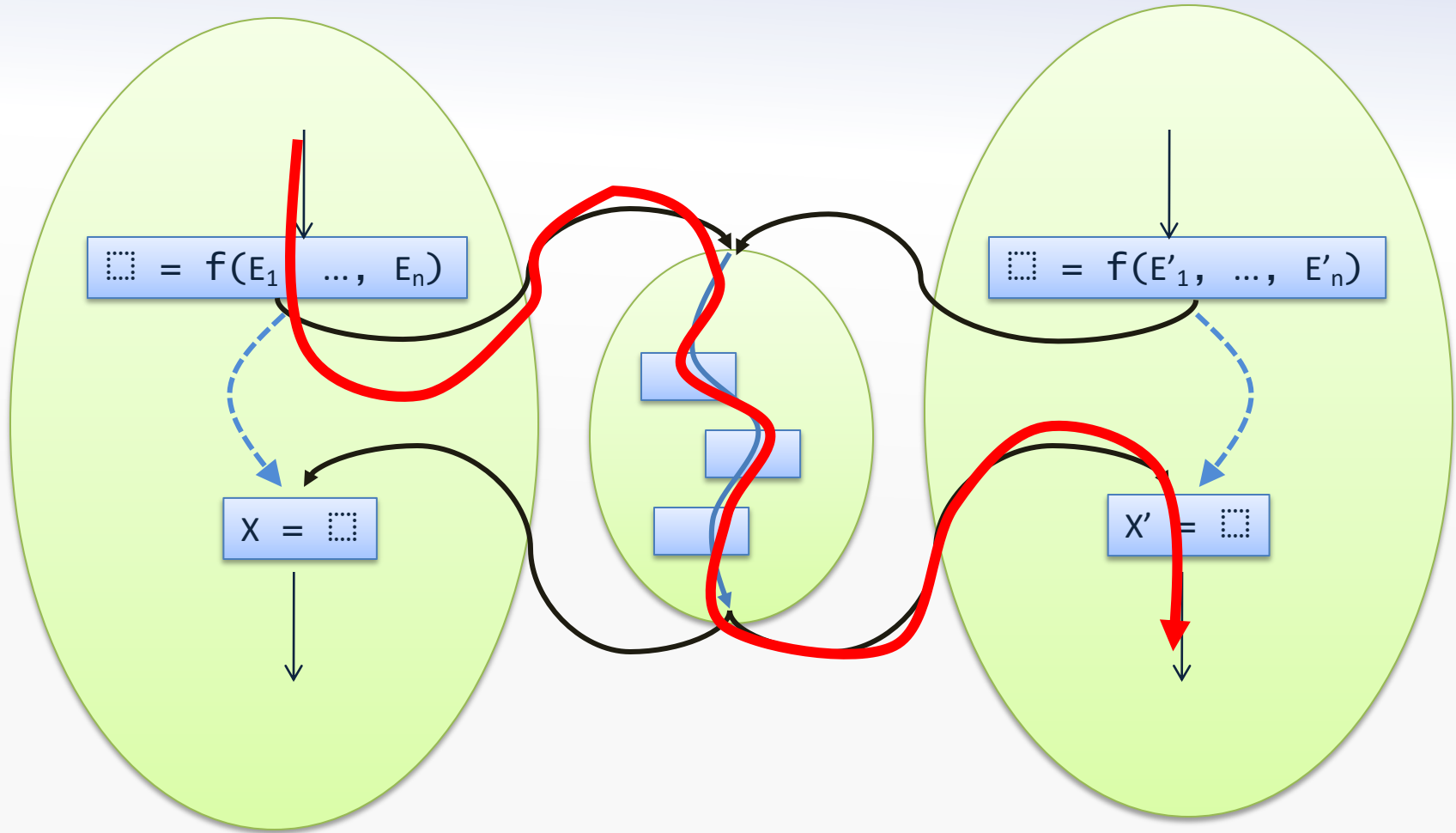
Implementation: `WorklistFixpointPropagationSolver`

13

Agenda

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

Interprocedurally invalid paths

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 "⊤"

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

$$\text{Contexts} \rightarrow \text{lift(States)}$$

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…

One-level cloning

- Let $c_1,\ldots,c_n$ be the call nodes in the program

- Define Contexts=$\{c_1,\ldots,c_n\} \cup \{\varepsilon\}$
  - each call node now defines its own "call context" (using ε 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 $c_1,...,c_n$ be the call nodes in the program

- Define Contexts as the set of strings over $\{c_1,...,c_n\}$ of length $\leq k$

  – such a string represents the top-most k call locations on the call stack

  – the empty string ε again represents the 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: Contexts $\rightarrow$ lift(Vars $\rightarrow$ Sign)
where 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
...
```

$[\varepsilon \mapsto$ unreachable,

$c1 \mapsto \bot[z\mapsto0, t1\mapsto0, t2\mapsto0],$
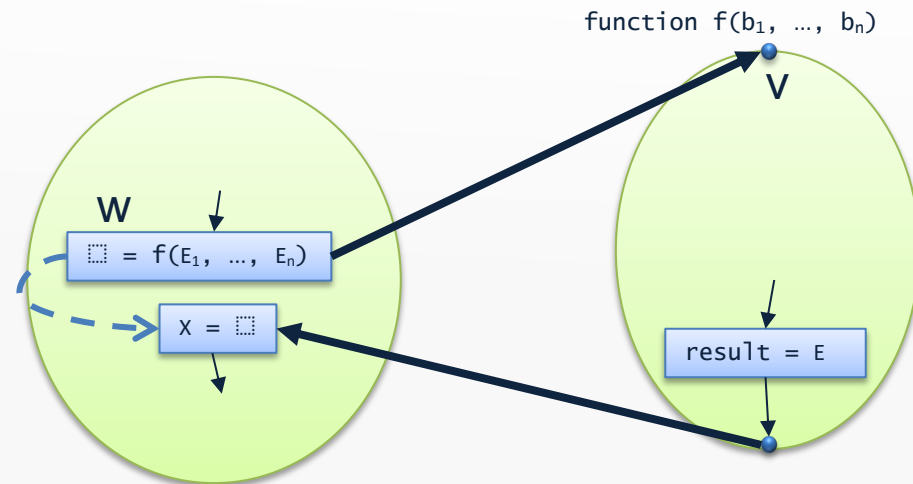
$c2 \mapsto \bot[z\mapsto+, t1\mapsto+, t2\mapsto+]]$

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')

$$[\![v]\!](c) = \bigsqcup_{\substack{w \in \text{pred}(v)\, \wedge \\ c = w\, \wedge \\ c' \in \text{Contexts}}} s_w^{c'}$$


```
function f(b₁, …, bₙ)
                          v

  w
   □ = f(E₁, …, Eₙ)

       X = □            result = E
```
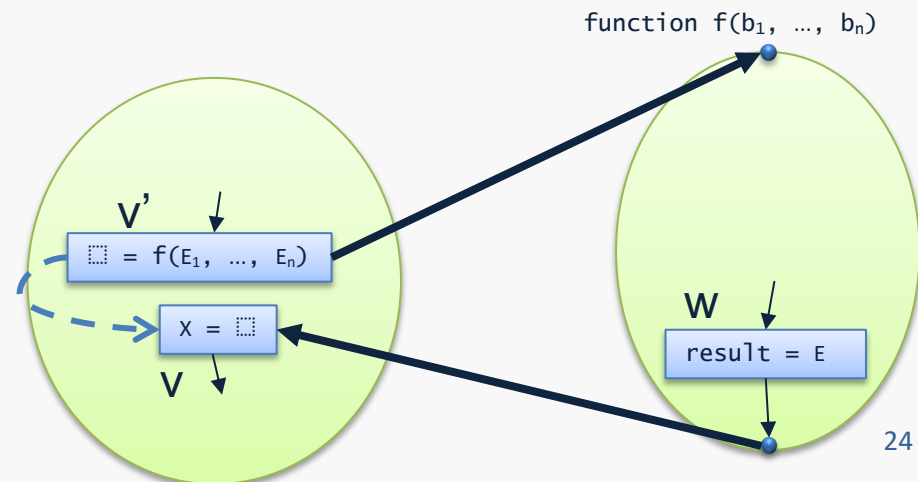
$$s_w^{c'} = \begin{cases} \text{unreachable} & \text{if } [\![w]\!](c') = \text{unreachable} \\ \bot[b_1 \rightarrow eval([\![w]\!](c'), E_1^w), \dots, b_n \rightarrow eval([\![w]\!](c'), E_n^w)] & \text{otherwise} \end{cases}$$

23

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

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

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

function f(b_1, …, b_n)

v'
$\Box$ = f(E_1, …, E_n)

X = $\Box$

v

w
result = E

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

<div align="center">Contexts = States</div>

(requires States to be finite)

- Each element of the lattice States → lift(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: Contexts → lift(Vars → Sign)
where Contexts = Vars → Sign

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

$[\perp[z \mapsto 0] \mapsto \perp[z \mapsto 0, t1 \mapsto 0, t2 \mapsto 0],$
$\perp[z \mapsto +] \mapsto \perp[z \mapsto +, t1 \mapsto +, t2 \mapsto +],$

all other contexts $\mapsto$ unreachable $]$

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 to skip function!

- 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!
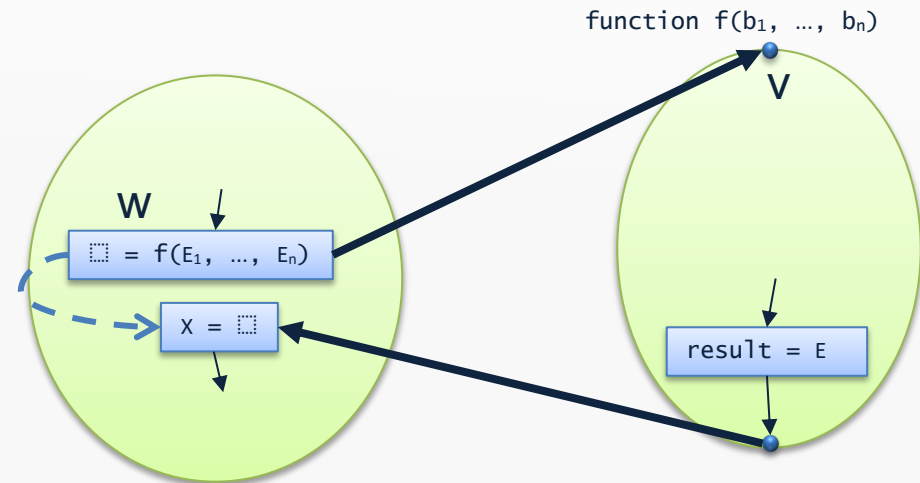
- ...but may be expensive if States is large

Implementation: `FunctionalSignAnalysis`

Context sensitivity with the functional approach
function entry nodes

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

$$[\![v]\!](c) = \bigsqcup_{\substack{w \in pred(v)\ \wedge \\ c = s_w^{c'}\ \wedge \\ c' \in \text{Contexts}}} s_w^{c'}$$

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$\in$pred(v):

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

function f(b$_1$, …, b$_n$)

v'
$\square$ = f(E$_1$, …, E$_n$)

X = $\square$

v

w
result = E