Static Program Analysis

Part 7 – interprocedural analysis

https://cs.au.dk/~amoeller/spa/

Anders Møller

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

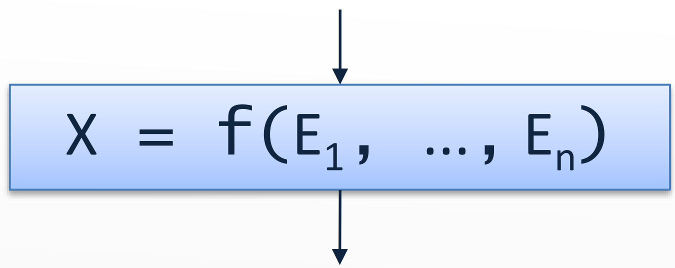
- Assume that all function calls are of the form

```
X = f(E1, …, En);
```

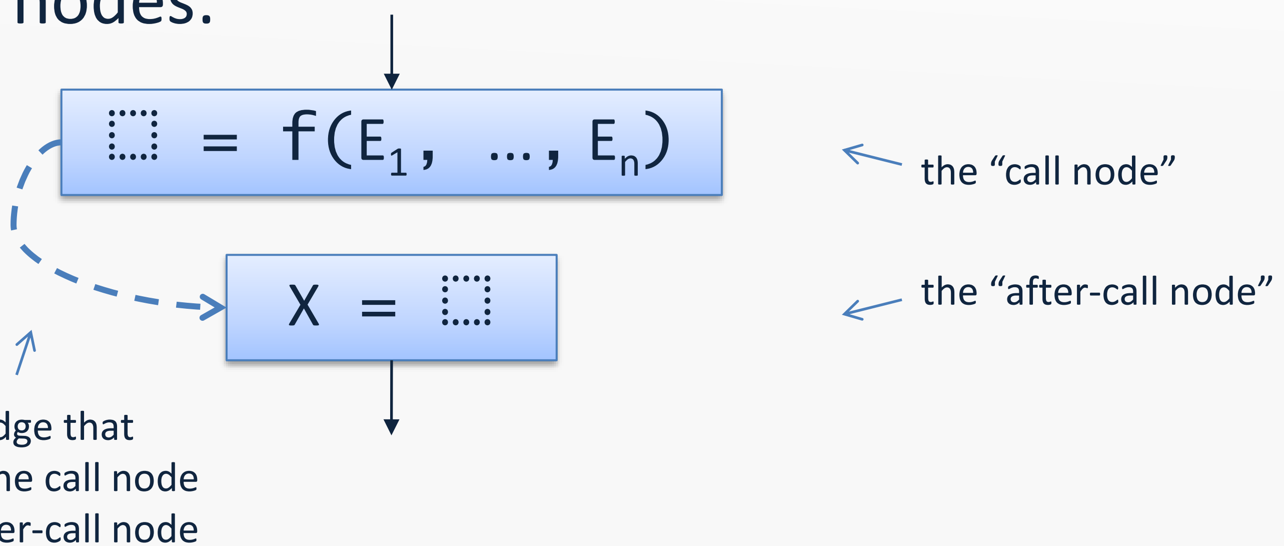
- This can always be obtained by normalization

Interprocedural CFGs (1/3)

Split each original call node



into two nodes:



Interprocedural CFGs (2/3)

Change each return node

```
return E
```

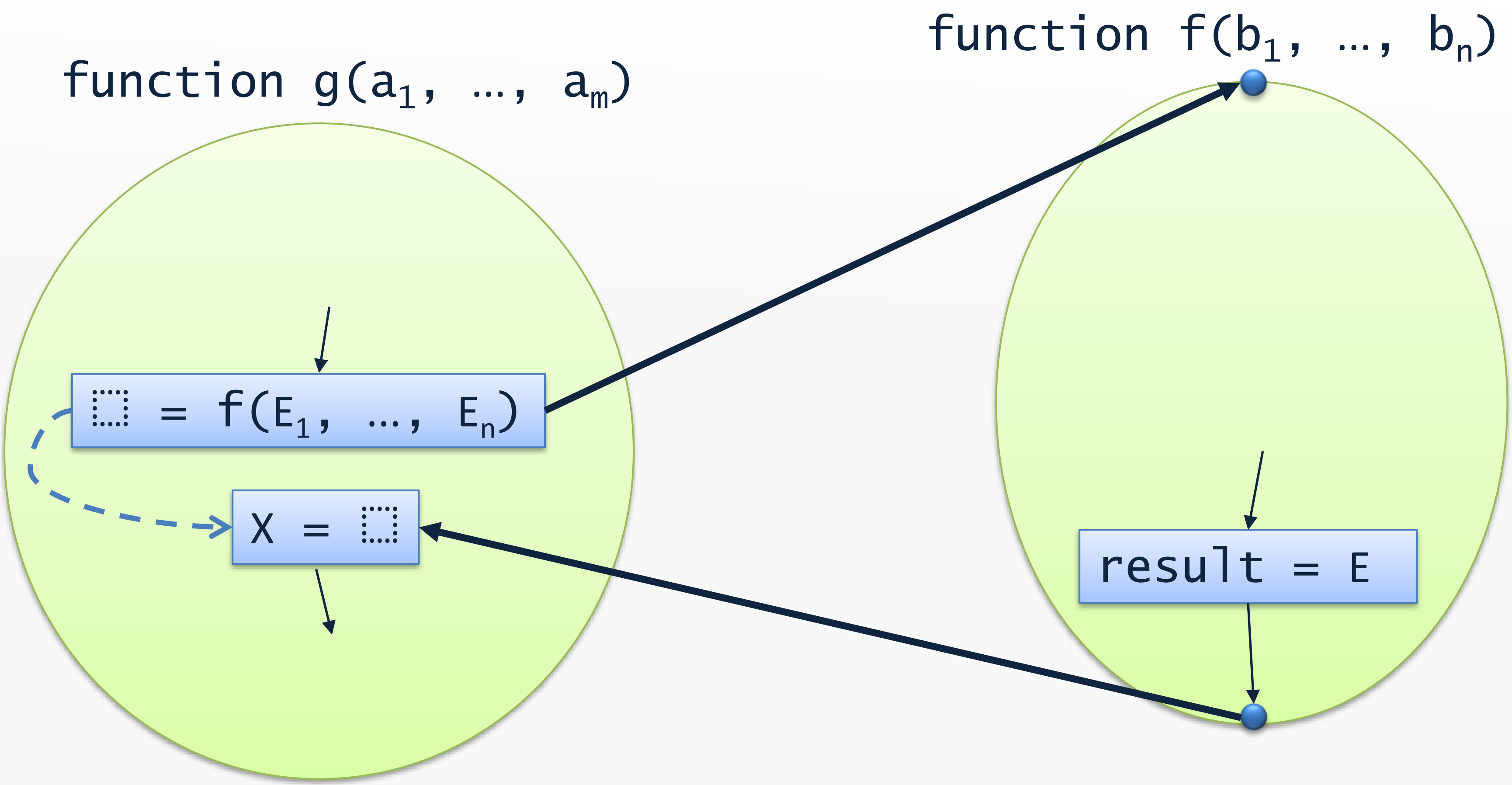
into an assignment:

```
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 (otherwise, it may fail for recursive functions)

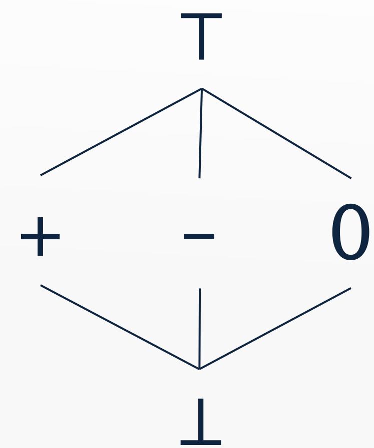
- For after-call/exit nodes:
  - like an assignment: `X = result`
  - but also restore local variables from before the call using the call↷after-call edge

- The details depend on the specific analysis...

Example: interprocedural sign analysis

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

Sign =



- Lattice for abstract states:

  $Var \rightarrow Sign$

Example: interprocedural sign analysis

- Constraint for entry node v of function f(b_1, …, b_n):

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

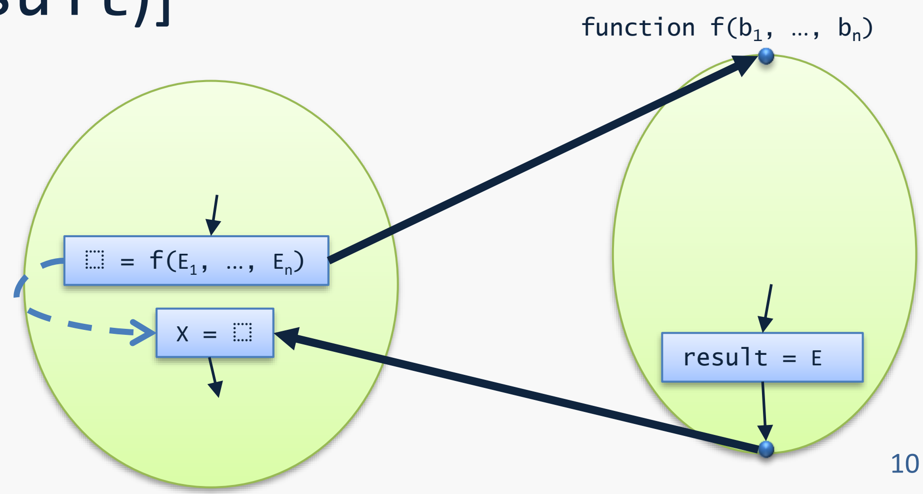
where E_i^w is i'th argument at w

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

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

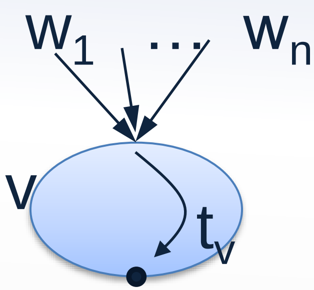
where w∈pred(v)

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



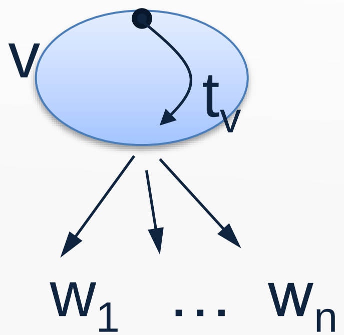
Alternative formulations

1) $[\![v]\!] = t_v(\bigsqcup_{w \in pred(v)} [\![w]\!])$

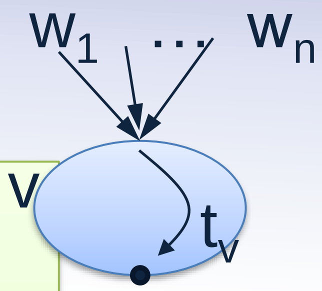


2) $\forall w \in succ(v)$: $t_v([\![v]\!]) \sqsubseteq [\![w]\!]$

– recall "solving inequations"
– may require fewer join operations if there are many CFG edges
– more suitable for *inter*procedural flow



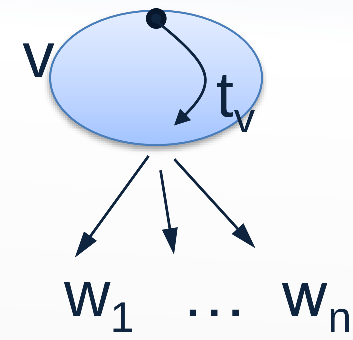
The worklist algorithm (original version)




```
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)

```
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`

Agenda

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

Motivating example

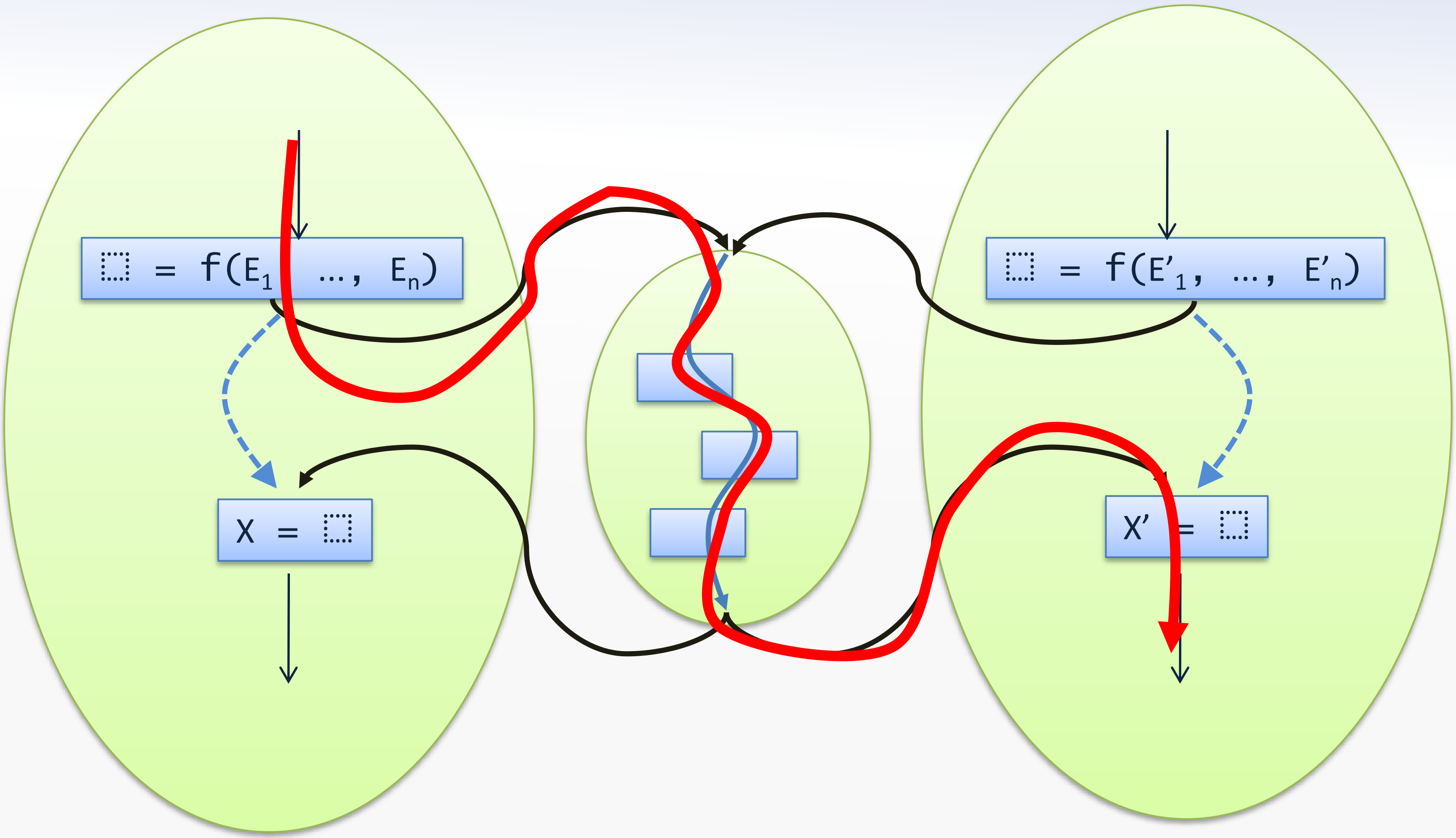
What is the sign of the return value of g?

```
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, *State*, by

$$\boxed{\mathit{Context} \to \mathit{lift}(\mathit{State})}$$

  where *Context* is a set of ***call contexts***
  - the contexts are abstractions of the state at function entry
  - *Context* must be finite to ensure finite height of the lattice
  - the bottom element of *lift*(*State*) represents "unreachable" contexts
- Different strategies for choosing the set *Context*…

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

Easily adjusted to the new lattice $\mathit{Context} \rightarrow \mathit{lift}(\mathit{State})$

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

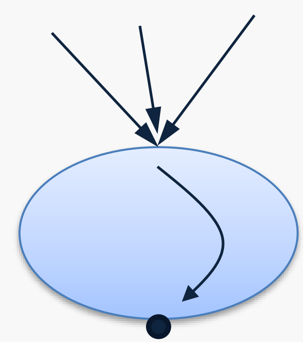
$[\![\, \mathrm{v} \,]\!] = \mathit{JOIN}(\mathrm{v})[x \mapsto \mathit{eval}(\mathit{JOIN}(\mathrm{v}),E)]$

becomes

$$[\![\, \mathrm{v} \,]\!](\mathrm{c}) = \begin{cases} s[x \mapsto \mathit{eval}(s,E)] & \text{if } s = \mathit{JOIN}(\mathrm{v},\mathrm{c}) \in \mathit{State} \\ \text{unreachable} & \text{if } \mathit{JOIN}(\mathrm{v},\mathrm{c}) = \text{unreachable} \end{cases}$$

and $\mathit{JOIN}(\mathrm{v}) = \bigsqcup_{\mathrm{w} \in \mathit{pred}(\mathrm{v})} [\![\mathrm{w}]\!]$

becomes $\mathit{JOIN}(\mathrm{v},\mathrm{c}) = \bigsqcup_{\mathrm{w} \in \mathit{pred}(\mathrm{v})} [\![\mathrm{w}]\!](\mathrm{c})$



One-level cloning

- Let $c_1,\dots,c_n$ be the call nodes in the program
- Define *Context*$=\{c_1,\dots,c_n\} \cup \{\varepsilon\}$
  - 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 $c_1,\ldots,c_n$ be the call nodes in the program
- Define *Context* as the set of strings over $\{c_1,\ldots,c_n\}$ 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 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: $Context \rightarrow \text{lift}(Var \rightarrow Sign)$

where $Context = \{\varepsilon, c_1, c_2\}$

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

$[\varepsilon \mapsto \text{unreachable},$
$\texttt{c1} \mapsto \bot[\texttt{z}\mapsto 0, \texttt{t1}\mapsto 0, \texttt{t2}\mapsto 0],$
$\texttt{c2} \mapsto \bot[\texttt{z}\mapsto +, \texttt{t1}\mapsto +, \texttt{t2}\mapsto +]]$

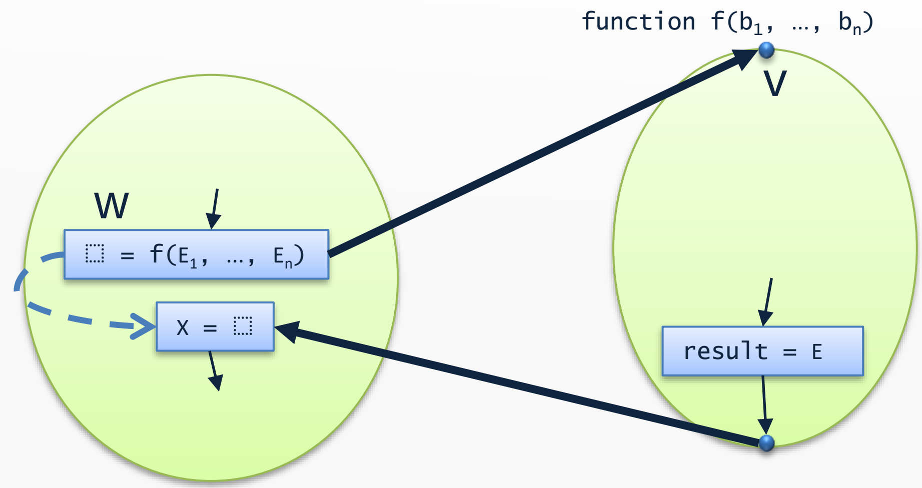
What is an example program that requires **k=2** to avoid loss of precision?

Context sensitivity with call strings
function entry nodes, for k=1

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

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

only consider the call node w that matches the context c



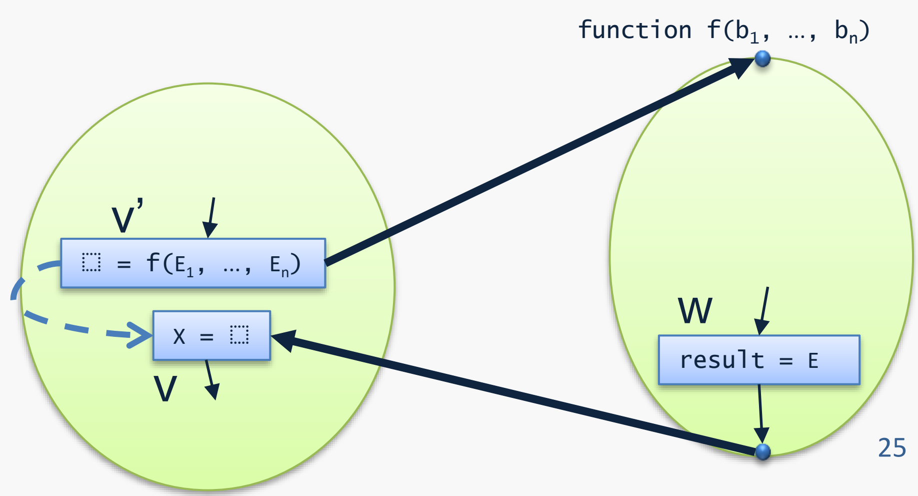
$$s_w^{c'} = \begin{cases} \text{unreachable} & \text{if } [\![w]\!](c') = \text{unreachable} \\ \bot[b_1 \rightarrow eval([\![w]\!](c'), E_1^w), \ldots, b_n \rightarrow eval([\![w]\!](c'), E_n^w)] & \text{otherwise} \end{cases}$$

Context sensitivity with call strings

after-call nodes, for k=1

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

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



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

  $$\mathit{Context} = \mathit{State}$$

  (requires *State* to be finite)
- Each element of the lattice $\mathit{State} \to \mathit{lift}(\mathit{State})$ is now a map m that provides an element m(x) from *State* (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: $\mathit{Context} \rightarrow \mathit{lift}(\mathit{Var} \rightarrow \mathit{Sign})$

where $\mathit{Context} = \mathit{Var} \rightarrow \mathit{Sign}$

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

$[\bot[z\mapsto 0] \mapsto \bot[z\mapsto 0,\ t1\mapsto 0,\ t2\mapsto 0],$
$\bot[z\mapsto +] \mapsto \bot[z\mapsto +,\ t1\mapsto +,\ t2\mapsto +],$
all other contexts $\mapsto$ unreachable $]$

Another example:
interprocedural sign analysis with the functional approach

Lattice for abstract states: $\mathit{Context} \to \mathit{lift}(\mathit{Var} \to \mathit{Sign})$

where $\mathit{Context} = \mathit{Var} \to \mathit{Sign}$

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

[⊥[z↦0] ↦ ⊥[z↦0, t1↦0, t2↦0],
⊥[z↦+] ↦ ⊥[z↦+, t1↦+, t2↦+],
all other contexts ↦ 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!
- 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 *State* is large

Implementation: `FunctionalSignAnalysis`

Example:
interprocedural sign analysis with the functional approach

Lattice for abstract states: $Context \to lift(Var \to Sign)$

where $Context = Var \to Sign$

```
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

[⊥[z↦0] ↦ ⊥[z↦0, `t1`↦0, `t2`↦0, `result`↦0],
⊥[z↦+] ↦ ⊥[z↦+, `t1`↦+, `t2`↦+, `result`↦+],
all other contexts ↦ unreachable]

At this call, we can reuse the already computed exit abstract state of `f` for the context ⊥[z↦+]

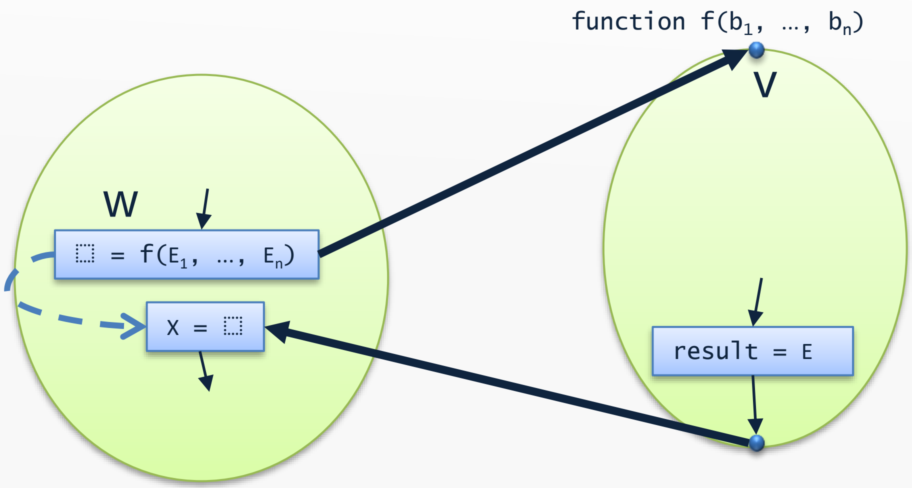
Context sensitivity with the functional approach

function entry nodes

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

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

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



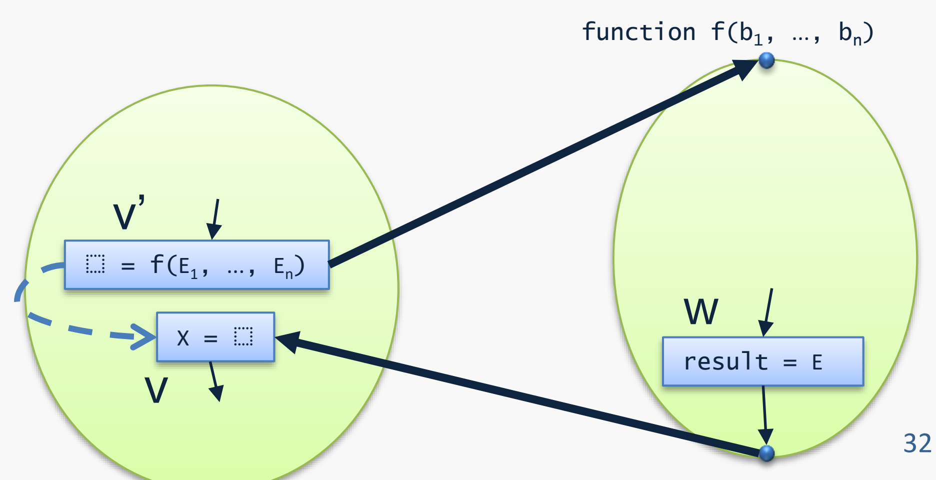
where $s_w^{c'}$ is defined as before

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 $w \in pred(v)$:

$$[\![v]\!](c) = \begin{cases} \text{unreachable} & \text{if } [\![v']\!](c) = \text{unreachable} \vee [\![w]\!](s^c_{v'}) = \text{unreachable} \\ [\![v']\!](c)[X \rightarrow [\![w]\!](s^c_{v'})(\mathtt{result})] & \text{otherwise} \end{cases}$$



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 *State* 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*)