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(E1, …, En);
```

- This can always be obtained by normalization

Interprocedural CFGs (1/3)

Split each original call node

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

into two nodes:

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

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
```

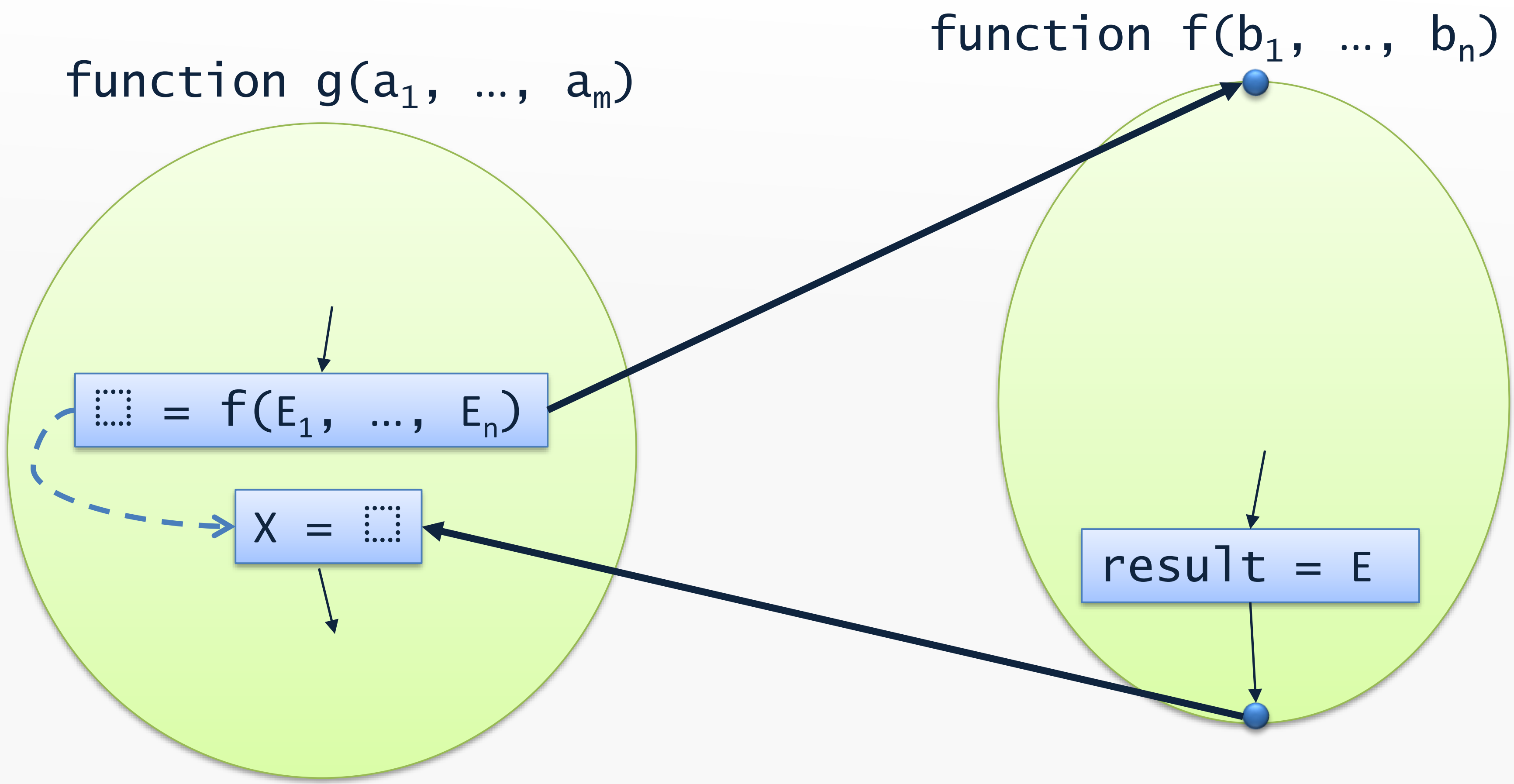
into an assignment:

```
result = E
```

(where `result` is a fresh variable)

Interprocedural CFGs (3/3)

Add call edges and return edges:



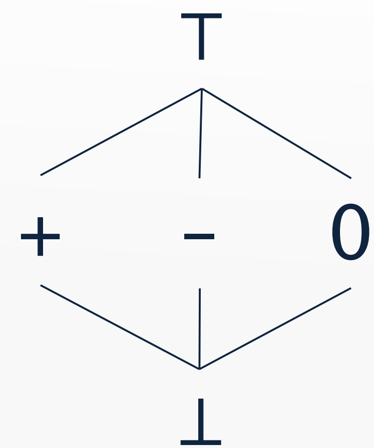
Constraints

- For call/entry nodes:
  - be careful to model evaluation of *all* the actual parameters before binding them to the formal parameter names (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:

  $Vars \rightarrow Sign$

Example: interprocedural sign analysis

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

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

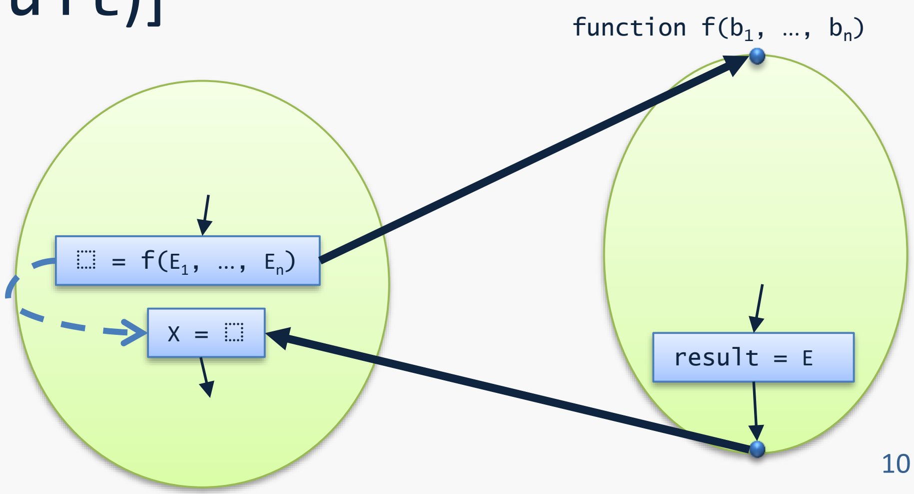
where E_i^w is i'th argument at w

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

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

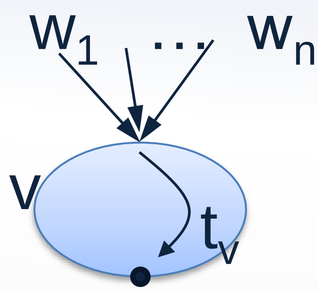
where w∈pred(v)

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



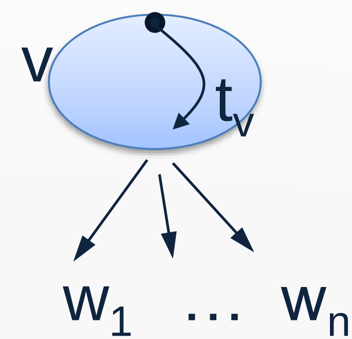
Alternative formulations

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

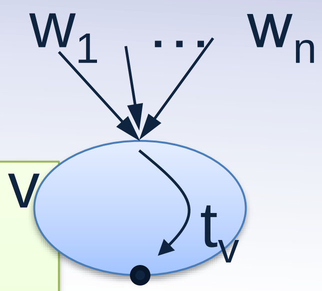


2) $\forall w \in \mathit{succ}(v)$: $t_v([\![v]\!]) \sqsubseteq [\![w]\!]$

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



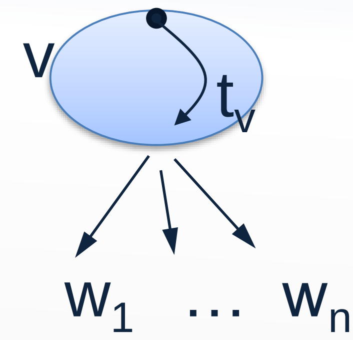
The worklist algorithm (original version)




```
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)

```
x_1 = ⊥; ... x_n = ⊥
W = {v_1, ..., v_n}
while (W≠∅) {
   v_i = W.removeNext()
   y = t_i(x_i)
   for (v_j ∈ dep(v_i)) {
      propagate(y,v_j)
   }
}
```



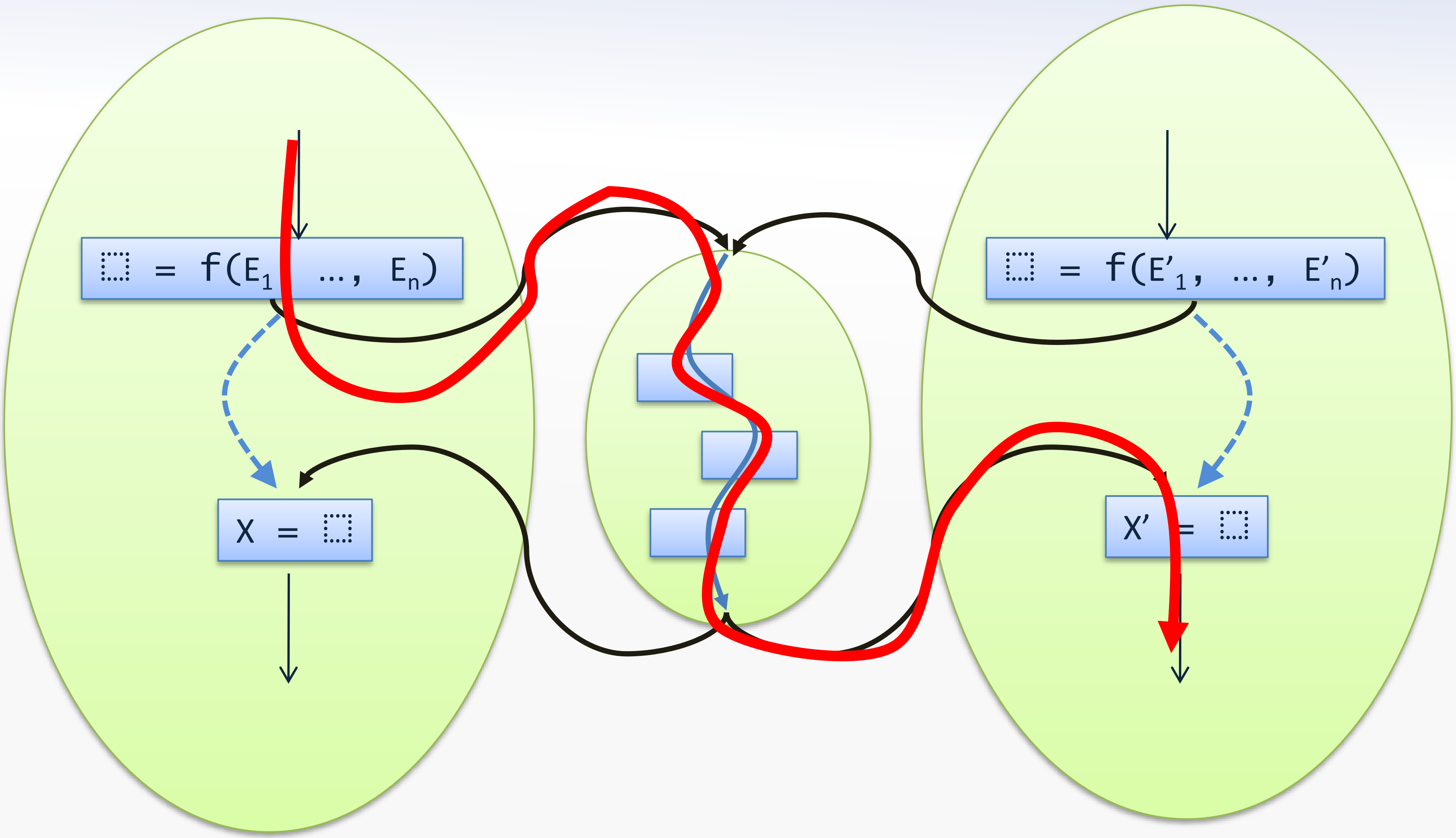

```
propagate(y,v_j) {
   z = x_j ⊔ y
   if (z≠x_j) {
      x_j = z
      W.add(v_j)
   }
}
```

Implementation: `WorklistFixpointPropagationSolver`

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 “T”

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

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

  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 $c_1,\ldots,c_n$ be the call nodes in the program
- Define Contexts 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 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 → lift(Vars → Sign)
where Contexts={ε,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$\mapsto$0, t1$\mapsto$0, t2$\mapsto$0],
c2 $\mapsto \bot$[z$\mapsto$+, t1$\mapsto$+, t2$\mapsto$+]]

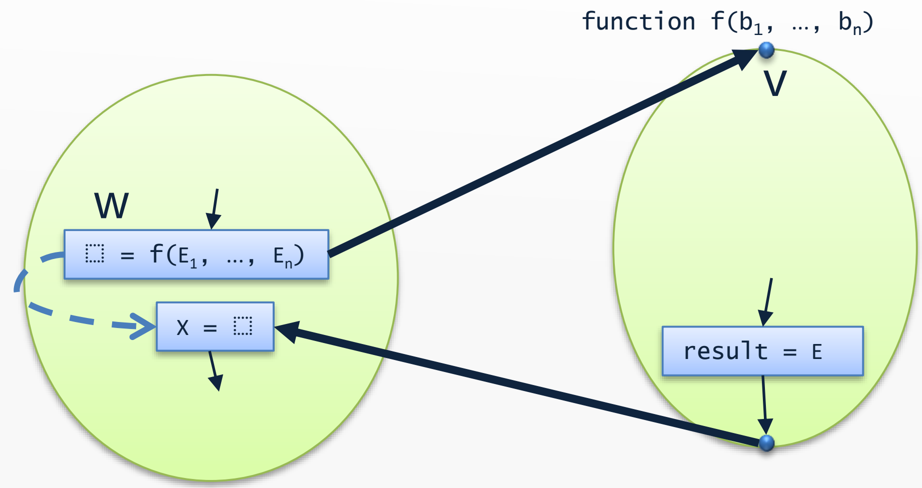
What is an example program that requires **k=2** to avoid loss of precision?

Context sensitivity with call strings

function entry nodes, for k=1

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

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



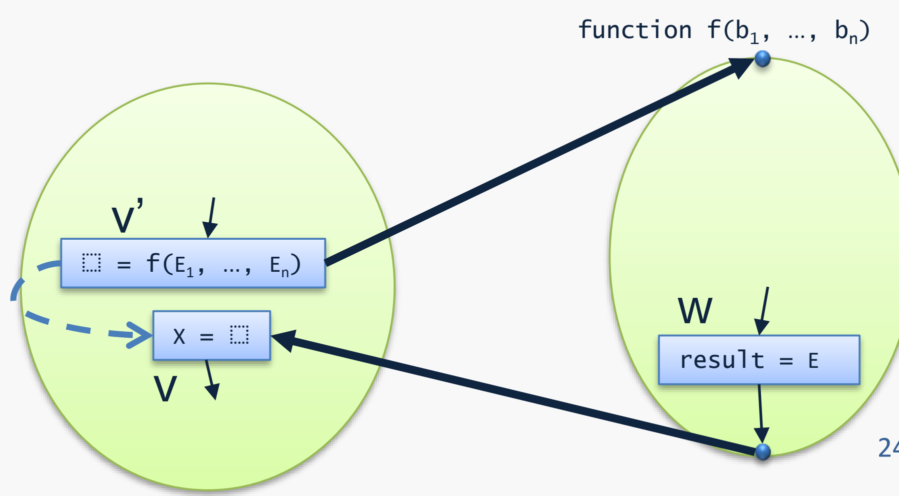
$$s_w^{c'} = \begin{cases} \text{unreachable} & \text{if } [\![w]\!](c') = \text{unreachable} \\ \bot[b_1 \rightarrow eval([\![w]\!](c'),E_1^w), \ldots, b_n \rightarrow eval([\![w]\!](c'),E_n^w)] & \text{otherwise} \end{cases}$$

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$\in$pred(v):

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



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

  Contexts = States

  (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);
...
```

[⊥[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 States is large

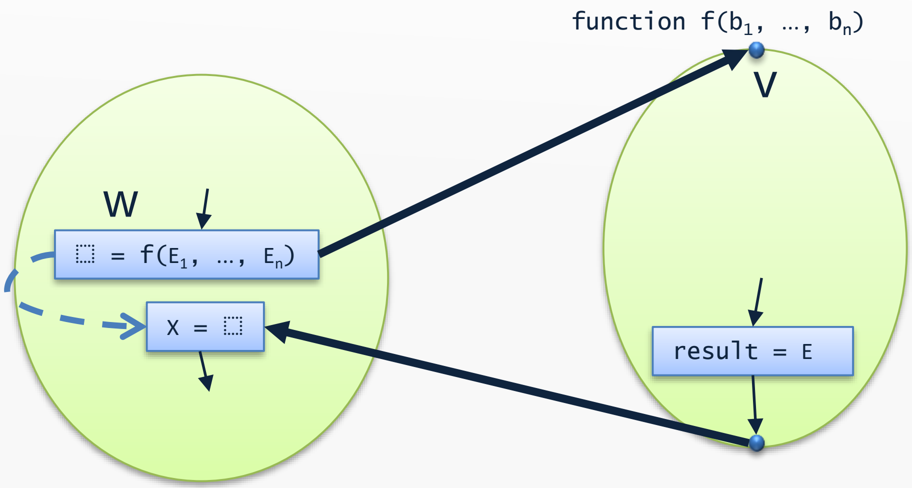
Implementation: `FunctionalSignAnalysis`

Context sensitivity with the functional approach

function entry nodes

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

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



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

Context sensitivity with the functional approach

after-call nodes

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

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