Interprocedural Analysis

Yu Zhang

Most content comes from http://cs.au.dk/~amoeller/spa/
http://staff.ustc.edu.cn/~yuzhang/pldpa

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 (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

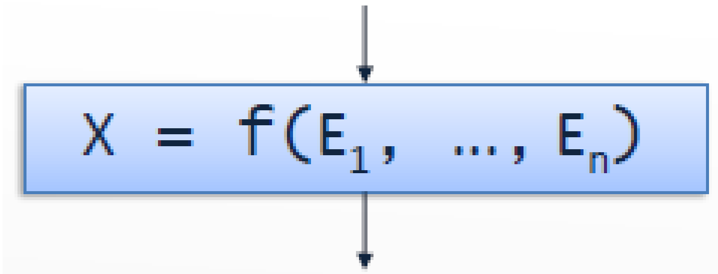
- Assume that all function calls are of the form

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

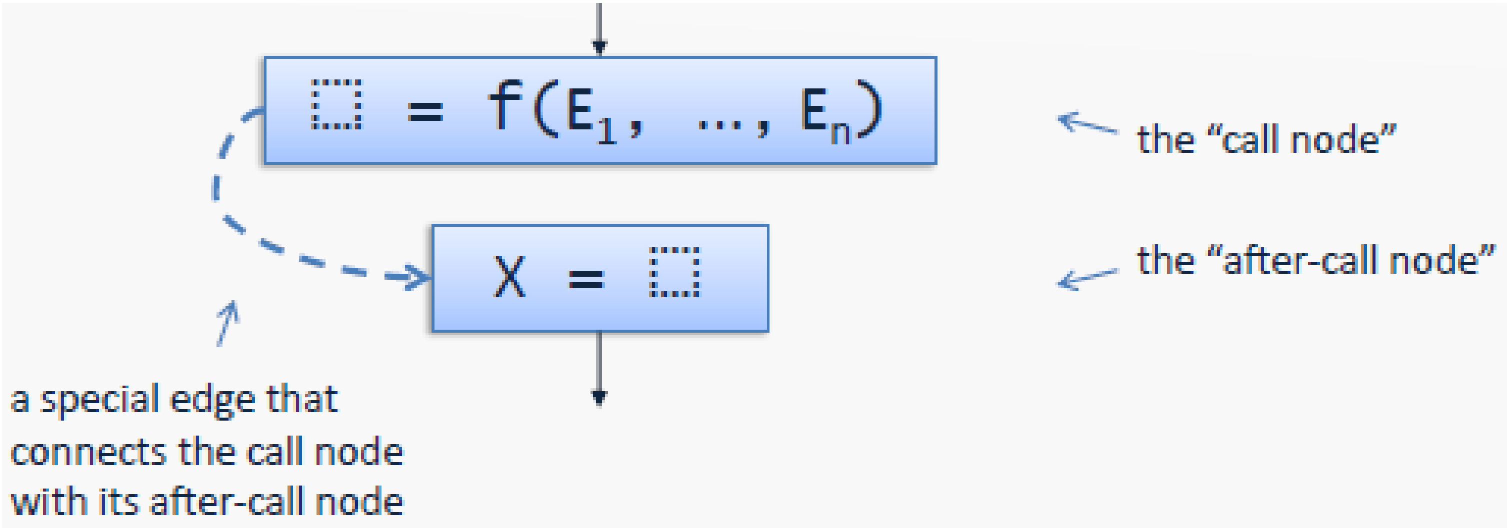
- This can always be obtained by normalization

Interprocedural CFGs (1/3)

Split each original call node

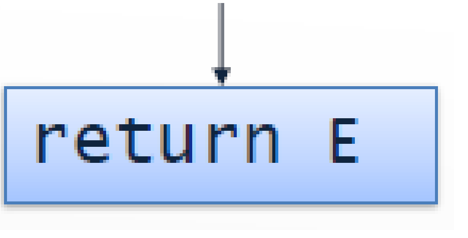


into two nodes:

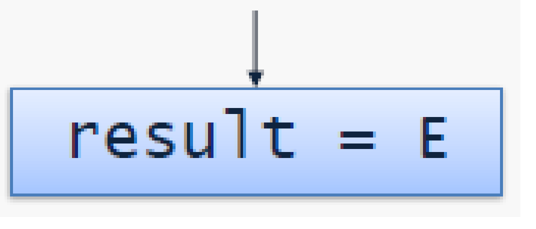


Interprocedural CFGs (2/3)

Change each return node



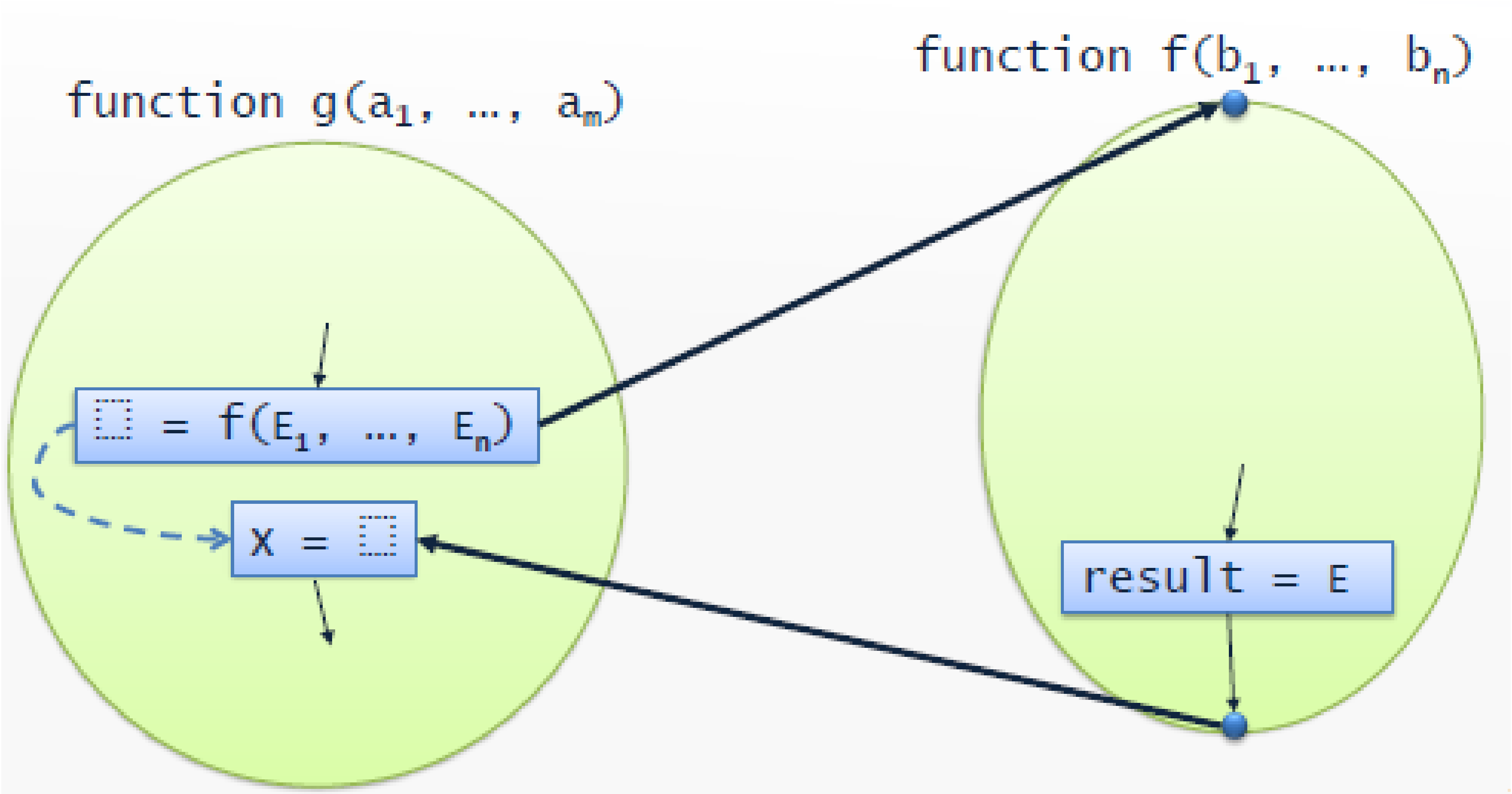
into an assignment:



(where result is a fresh variable)

Interprocedural CFGs (3/3)

Add call edges and return edges:

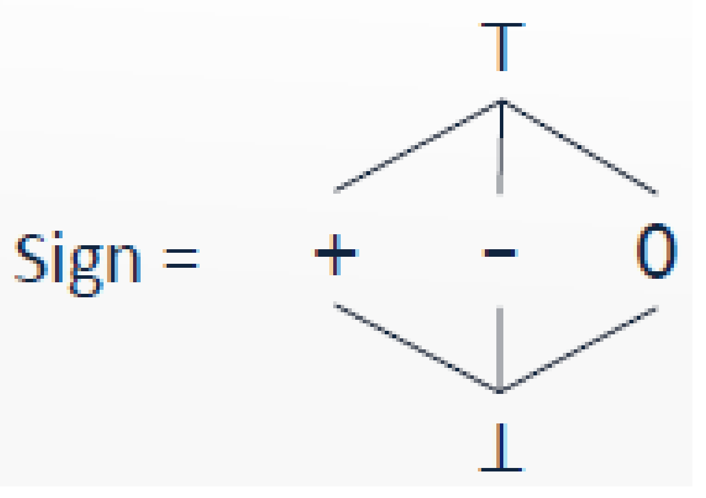


Constraints

- 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 ↷ 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→*Sign*

Example: Interprocedural Sign Analysis

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

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

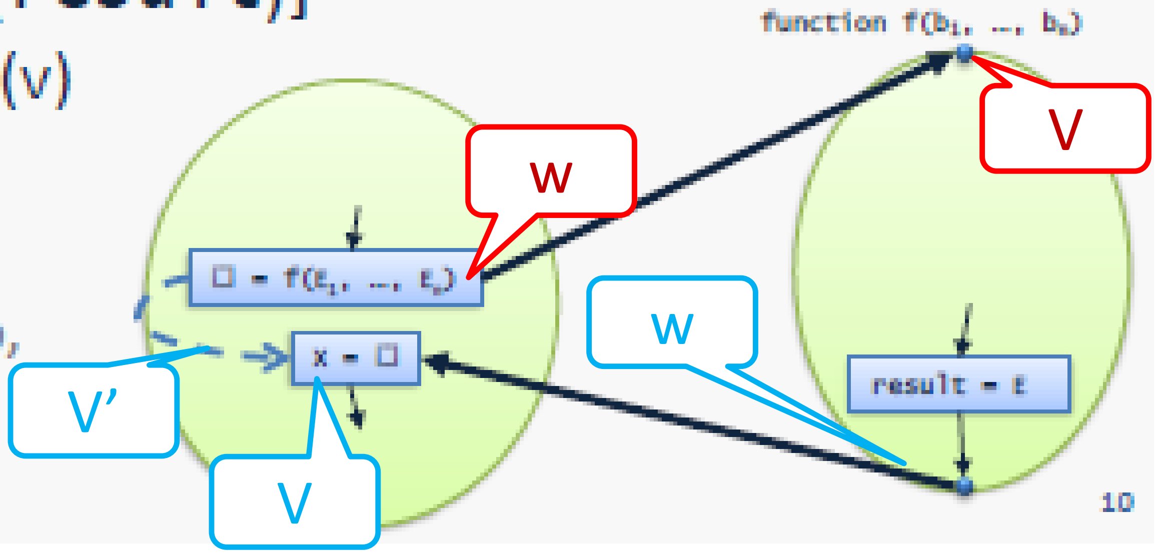
where E_i^w is i'th argument at w

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

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

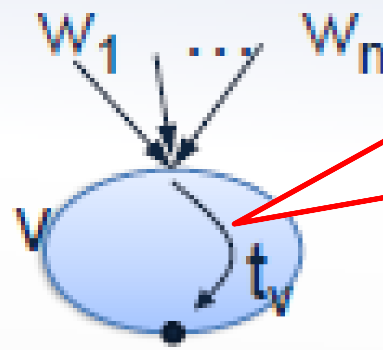
where w∈pred(v)

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



Alternative Formulations

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



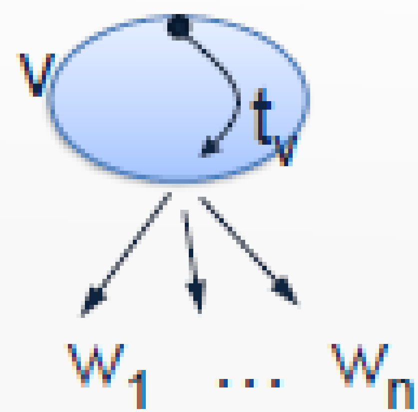
v是对每个call节点w_i进行汇集，t_v是在主调v后对callee进行参数传递

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

– recall "solving inequations"

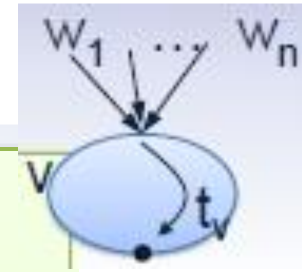
– may require fewer join operations if there are many CFG edges

– more suitable for *inter*procedural flow



t_v是将返回的退出点v应用到每个主调的after-call节点w_i进行返回值的接收处理

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
   }
}
```

基于过程间的大CFG

处理call节点

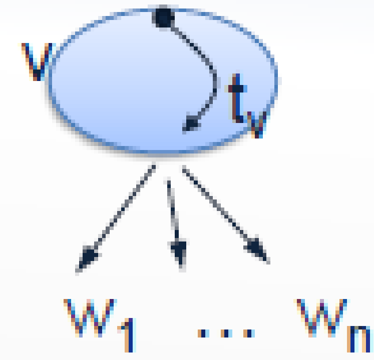
CFG结点vi 的变迁函数

如果CFG结点vi 的语义值发生变化，则将计算vi 语义值所依赖的结点vj 加入到worklist

The Worklist Algorithm (alternative version)

处理after-call节点

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

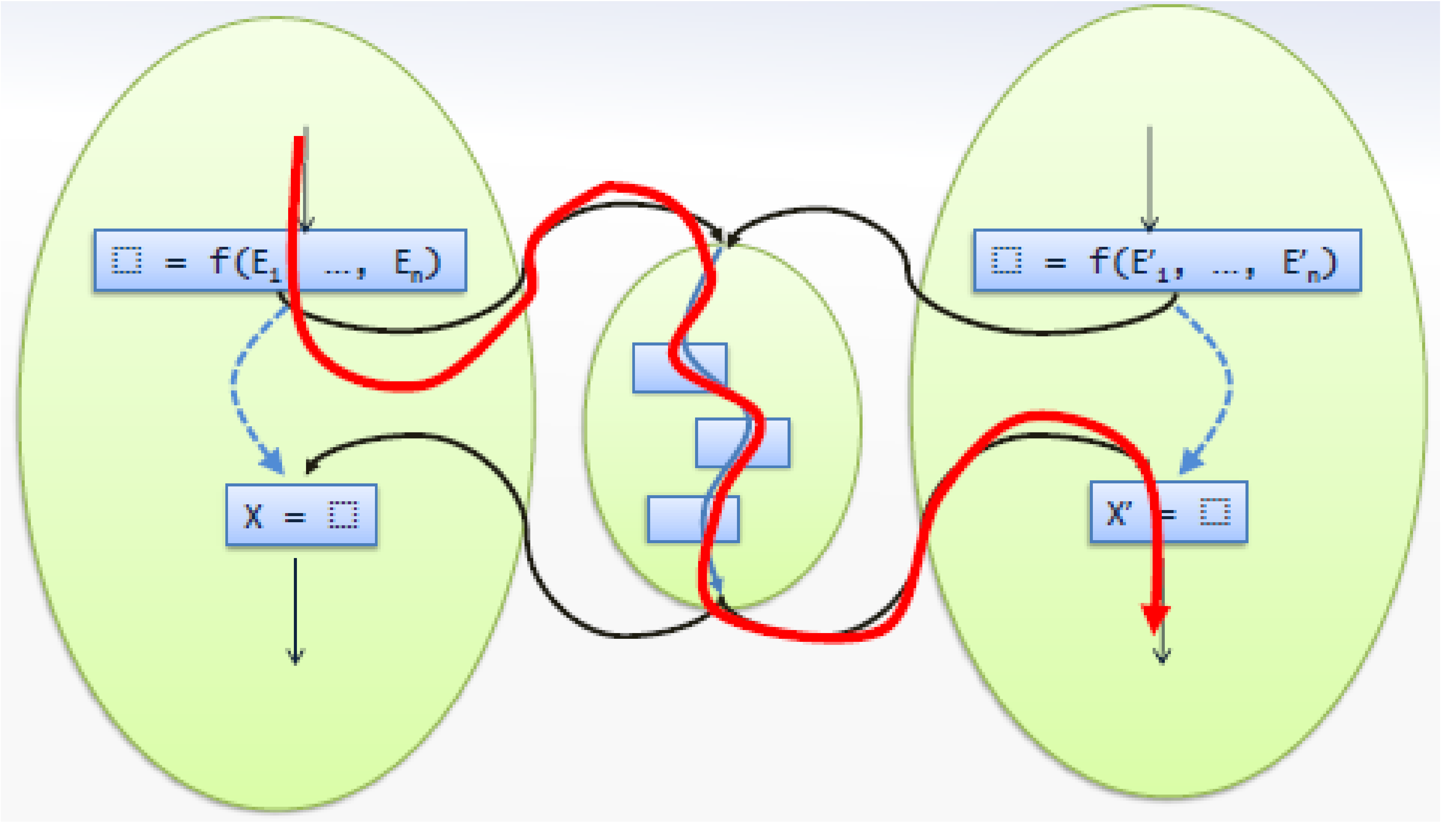



```
propagate(y,vⱼ) {
  z = xⱼ ⊔ y
  if (z≠xⱼ) {
    xⱼ = z
    W.add(vⱼ)
  }
}
```

Implementation: WorklistFixpointPropagationSolver

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

Interprocedurally Invalid Paths

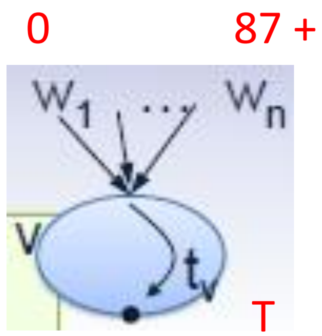


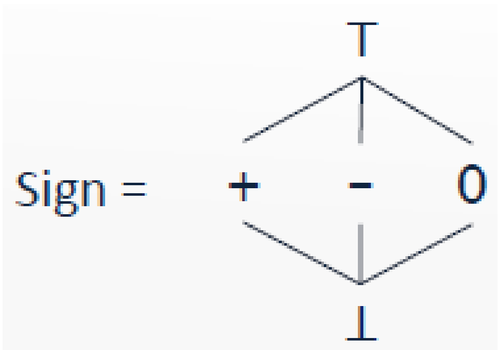
Example

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, give exponential blow-up☹
- Don'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;
}
```

对f副本的调用

优点：分析精度提升
缺点：代码膨胀

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 → 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 Contexts → lift(States)

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

  ⟦v⟧=*JOIN*(v)[*x*→ *eval*(*JOIN*(v), *E*)]

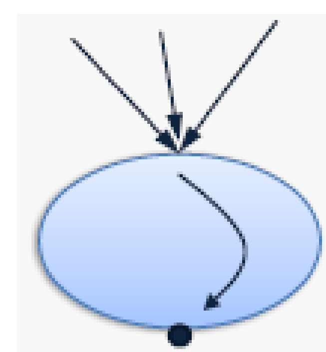
  becomes

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

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

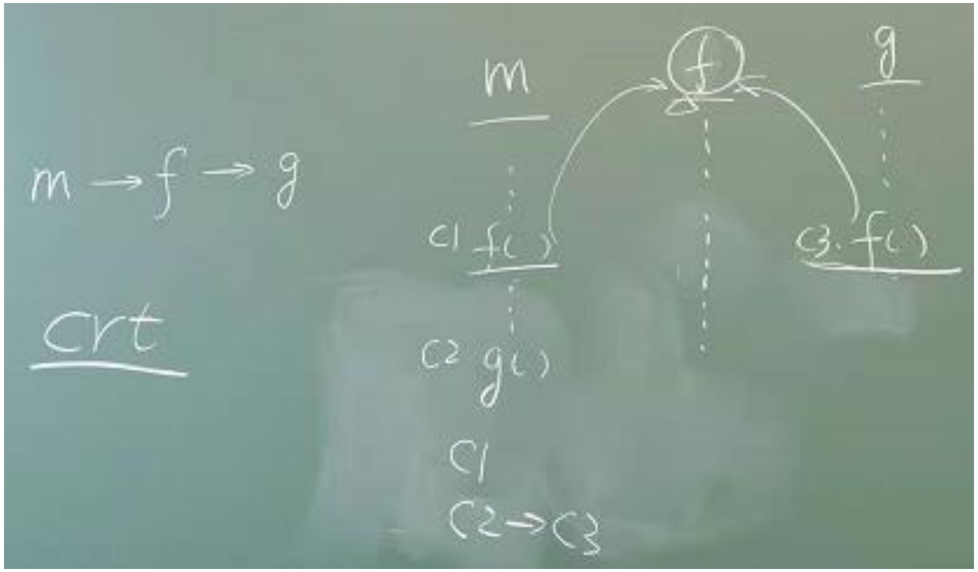
becomes

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



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



crt：C RunTime

a set of execution startup routines linked into a C program that performs any initialization work required before calling the program's main function.

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
...
```

[ε ↦ unreachable,
 c1 ↦ ⊥[z↦0, t1↦0, t2↦0],
 c2 ↦ ⊥[z↦+, t1↦+, t2↦+]]

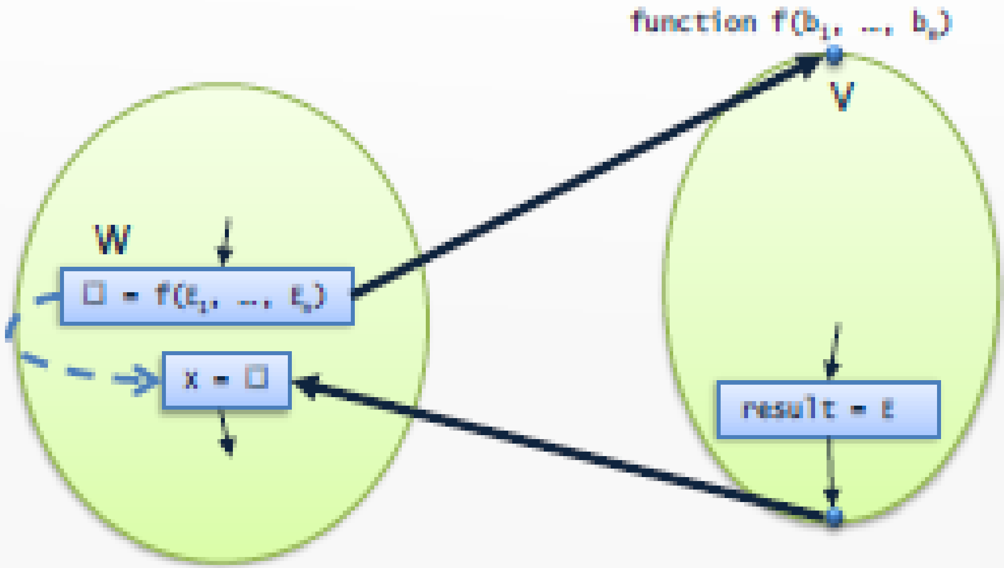
What is an example program
that requires k=2
to avoid loss of precision?

Context Sensitivity with Call Strings
function entry nodes, for k=1

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

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

Only consider the call node w that matches the context c



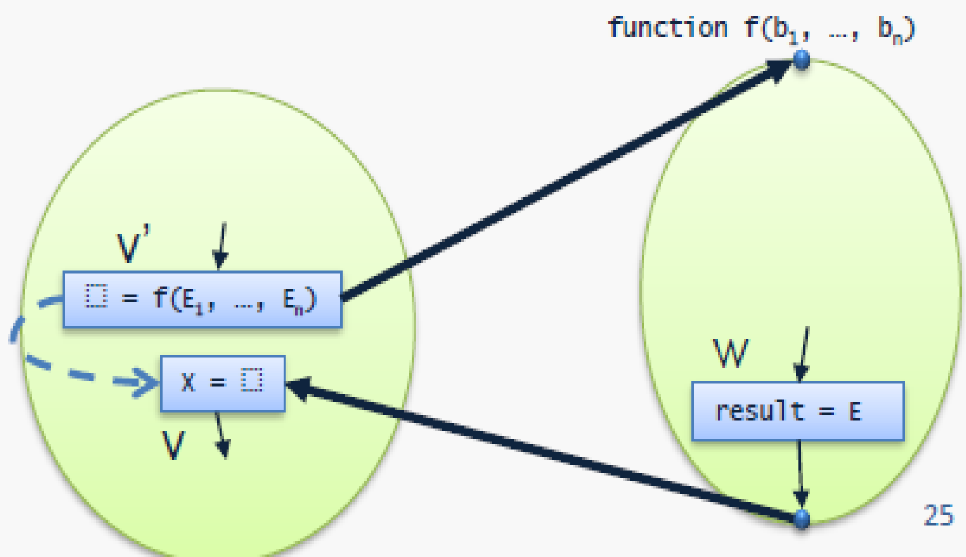
$$s_w^{c'} = \begin{cases} \text{unreachable} & \text{if } [\![w]\!](c') = \text{unreachable} \\ \bot[b_1 \rightarrow eval([\![w]\!](c'), E_1^w), \ldots, b_n \rightarrow eval([\![w]\!](c'), E_n^w)] & \text{otherwise} \end{cases}$$

Context Sensitivity with Call Strings

after-call nodes, for k=1

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

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



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);
...
```

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

Another 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;
}
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 States is large

Implementation: FunctionalSignAnalysis

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 = 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

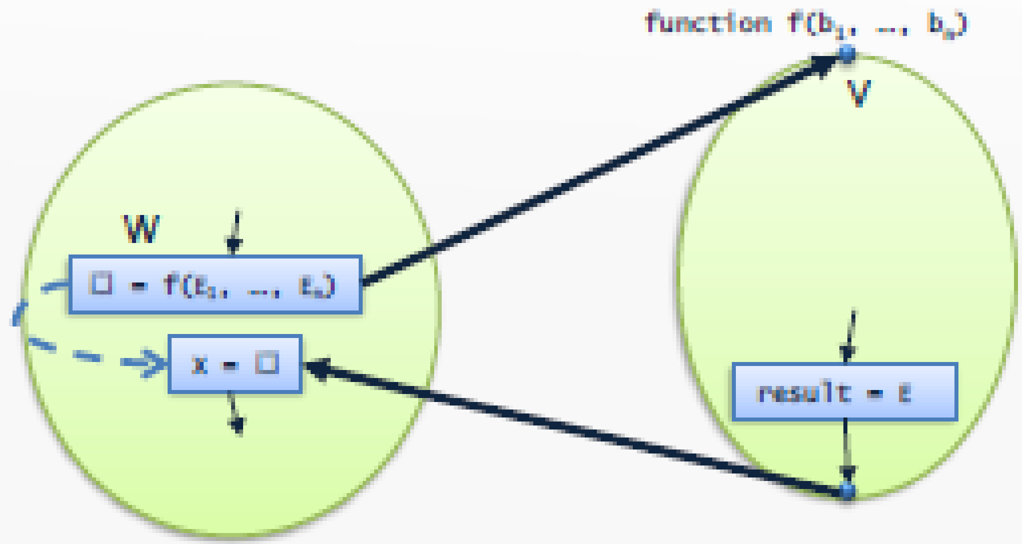
function entry nodes

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

(if not 'main')

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

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



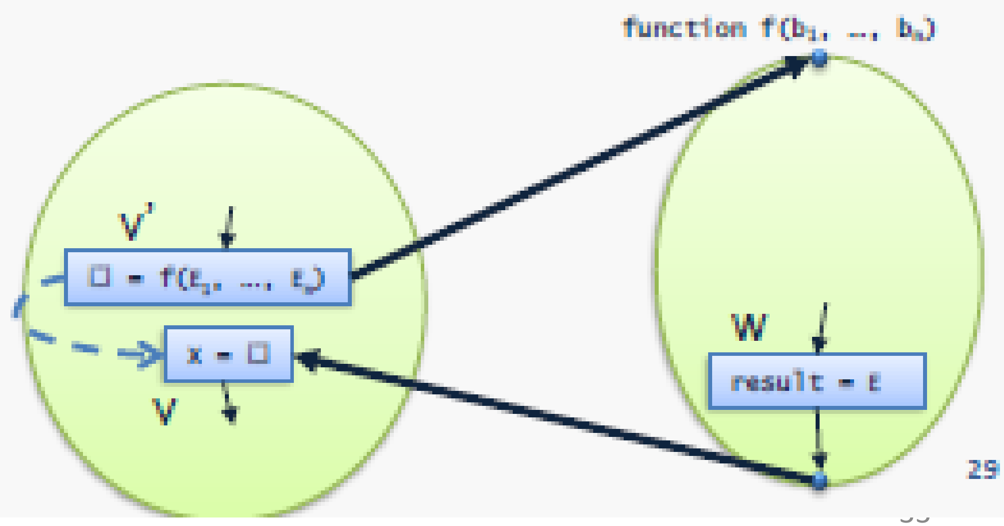
where $s_w^{c'}$ is defined as before

Context sensitivity with the functional approach

after-call nodes

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

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



Choose 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*)