Alias Analysis with IFDS

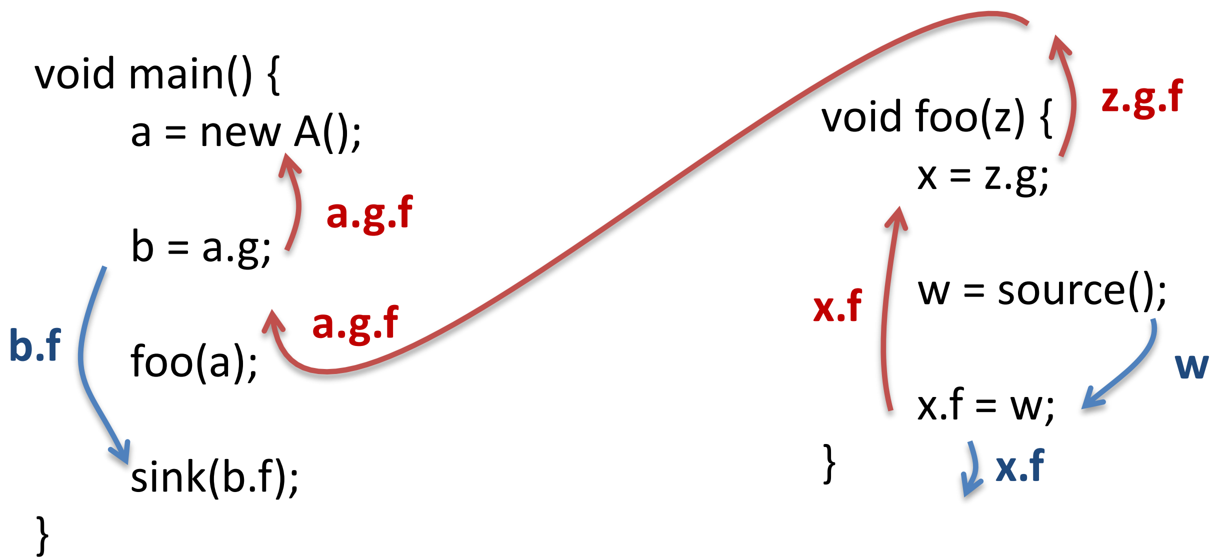
Applied Static Analysis 2016

Johannes Lerch

Dr. Michael Eichberg, Ben Hermann, Sebastian Proksch, Karim Ali Ph.D.

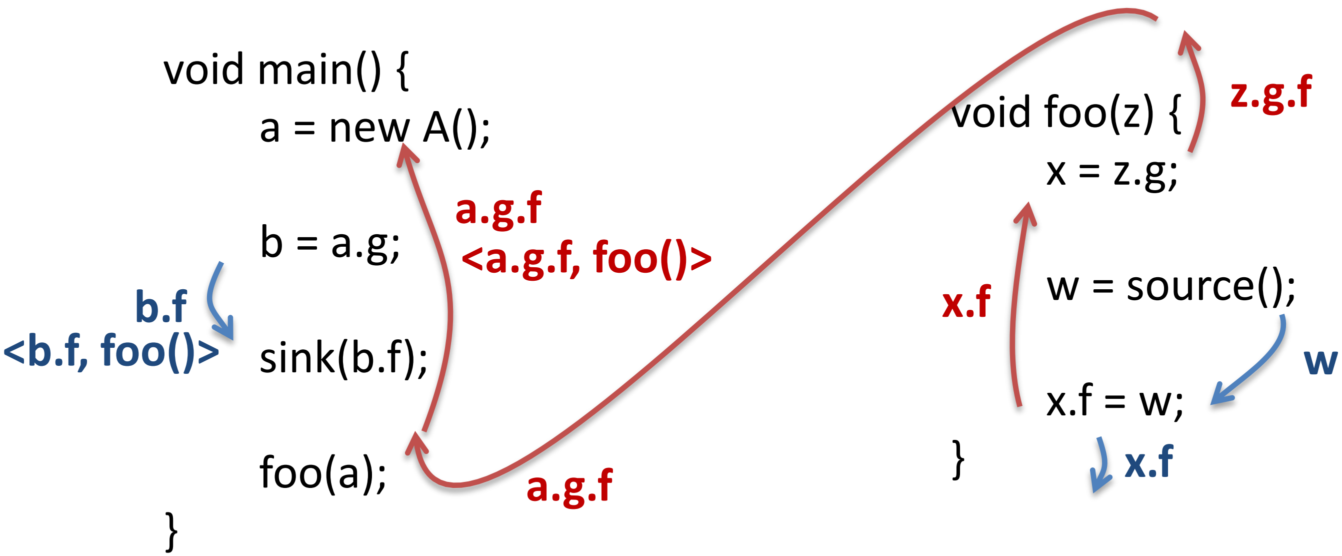
Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Oteau, and Patrick McDaniel: FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. PLDI'14

On-Demand Alias Analysis



- Spawn a backward analysis searching for aliases, when writing a field
- Spawn a forward analysis for each found alias

Activation Statement



- Spawn a backward analysis searching for aliases, when writing a field
- Spawn a forward analysis for each found alias
- Use activation statement to enable a taint only after passing that statement

Context-Free Language Reachability Problem

Applied Static Analysis 2016

Johannes Lerch

Dr. Michael Eichberg, Ben Hermann, Sebastian Proksch, Karim Ali Ph.D.

Thomas Reps: Program analysis via graph reachability. ILPS'97

David Melski, and Thomas Reps: Interconvertibility of a Class of Set Constraints and Context-Free-Language Reachability. Theoretical Computer Science Journal, Volume 248, 2000

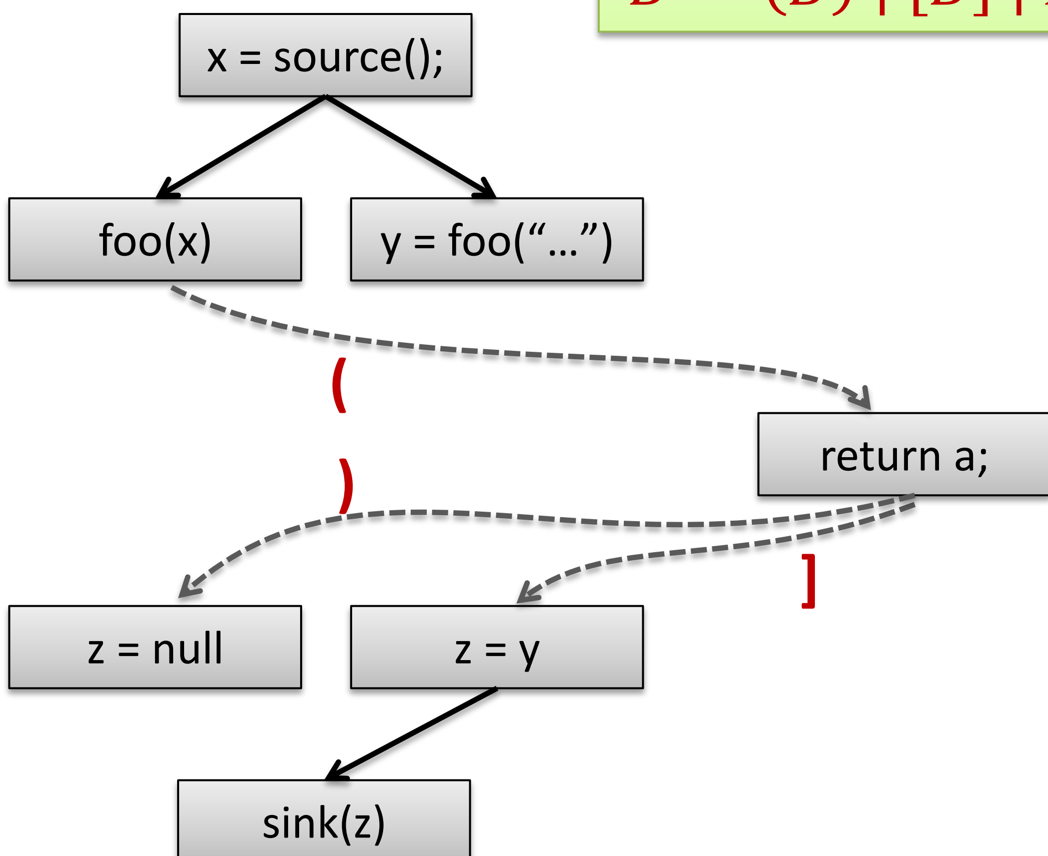
Context-Sensitive Analysis

```
main() {
	x = source();
	if(unknown()) {
		y = foo(x);
		z = null;
	}
	else {
		y = foo("const");
		z = y;
	}
	sink(z);
}

foo(a) {
	return a;
}
```

Describe Valid Paths via Language:

$B \rightarrow (B) \mid [B] \mid BB \mid \epsilon$



Context-Free Language Reachability Problem

- Label edges in the graph
- Each path in the graph defines a word by concatenating labels of its edges
- A path is valid, if the corresponding word is in some (context-free) language

Algorithm to Solve CFL-RP

1. Normalize the grammar, such that right-hand sides only contain at most 2 symbols:
$$A \to BCD \;\Rightarrow\; A \to BA' \quad A' \to CD$$
2. Create initial worklist:
Add to worklist W all edges of the graph
3. Add edges for ϵ-productions:
for each rule $A \to \epsilon$ and each node i
add $A\langle i, i \rangle$ to the graph and worklist

Edge labeled A from i to i

Algorithm to Solve CFL-RP (2)

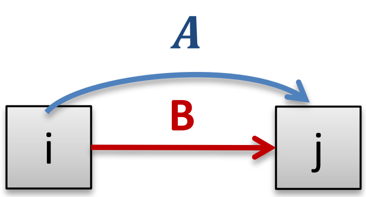
4. Add edges for other productions:

   while W is not empty

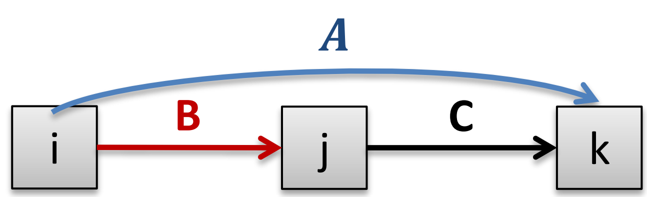
   select and remove edge $B\langle i, j\rangle$ from W

   add edges to graph and W (if not already in graph):

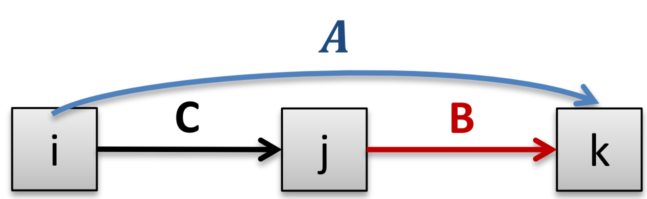
for $A \rightarrow B$:



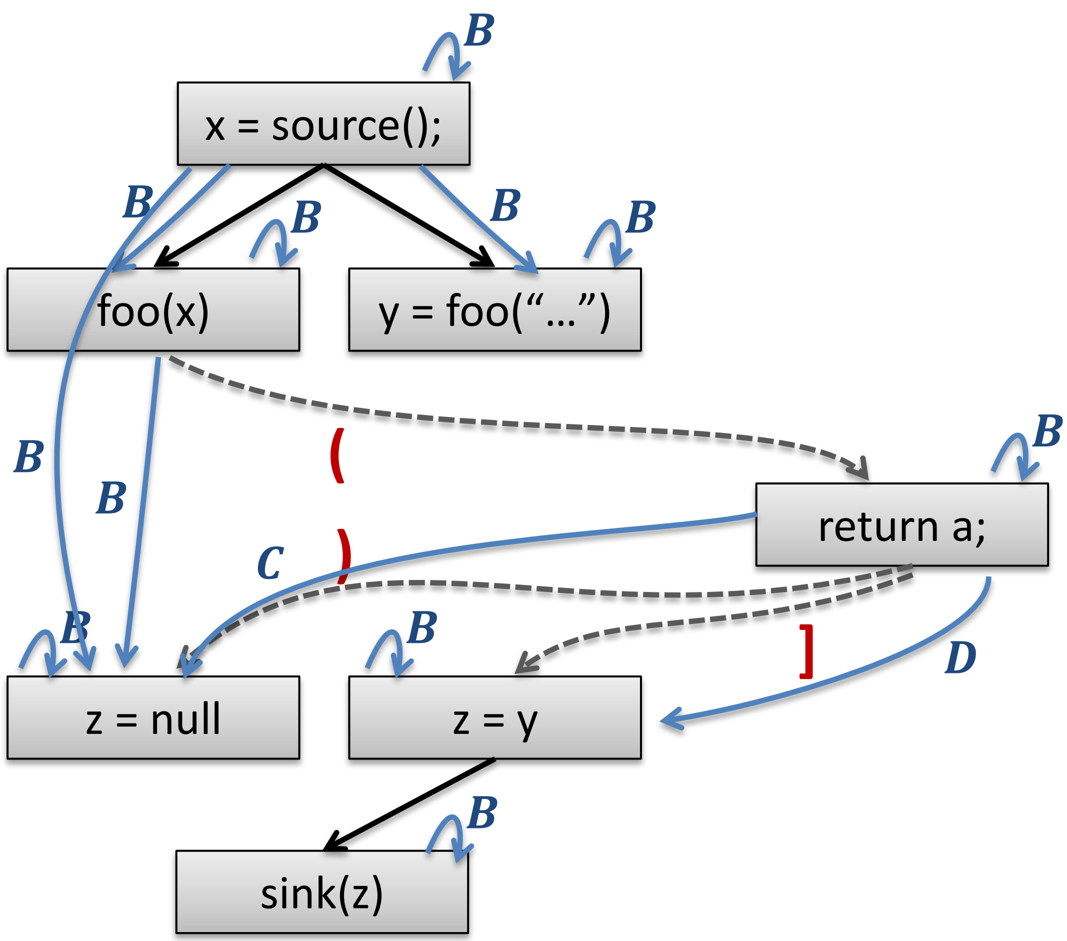
for $A \rightarrow BC$:



for $A \rightarrow CB$:



Context-Sensitive Analysis (2)



Describe Valid Paths via Language:

$B \rightarrow (B) \mid [B] \mid BB \mid \epsilon$

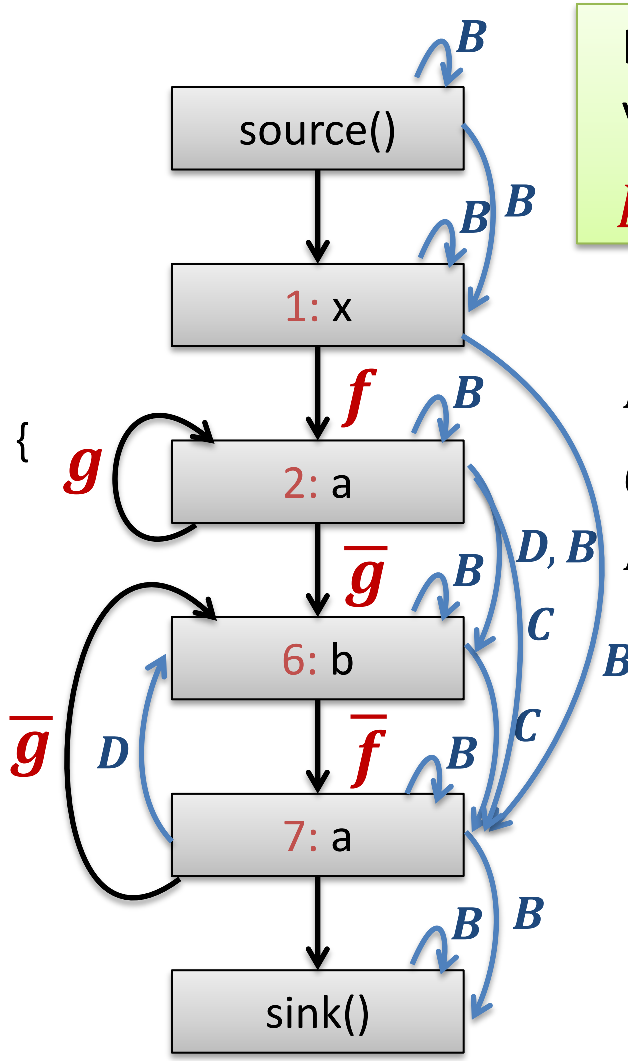
$B \rightarrow (C \mid [D \mid BB \mid \epsilon$

$C \rightarrow B)$

$D \rightarrow B]$

Field-Sensitive Analysis

```
    main() {
1:      x = source();
2:      a = new A();
3:      a.f = x;
4:      a.g = a;
5:      while(unknown()) {
6:          b = a.g
7:          a = b.f
8:      }
9:      sink(a);
    }
```



Describe Valid Paths
via Language:

$B \to f B \overline{f} \mid g B \overline{g} \mid BB \mid \epsilon$

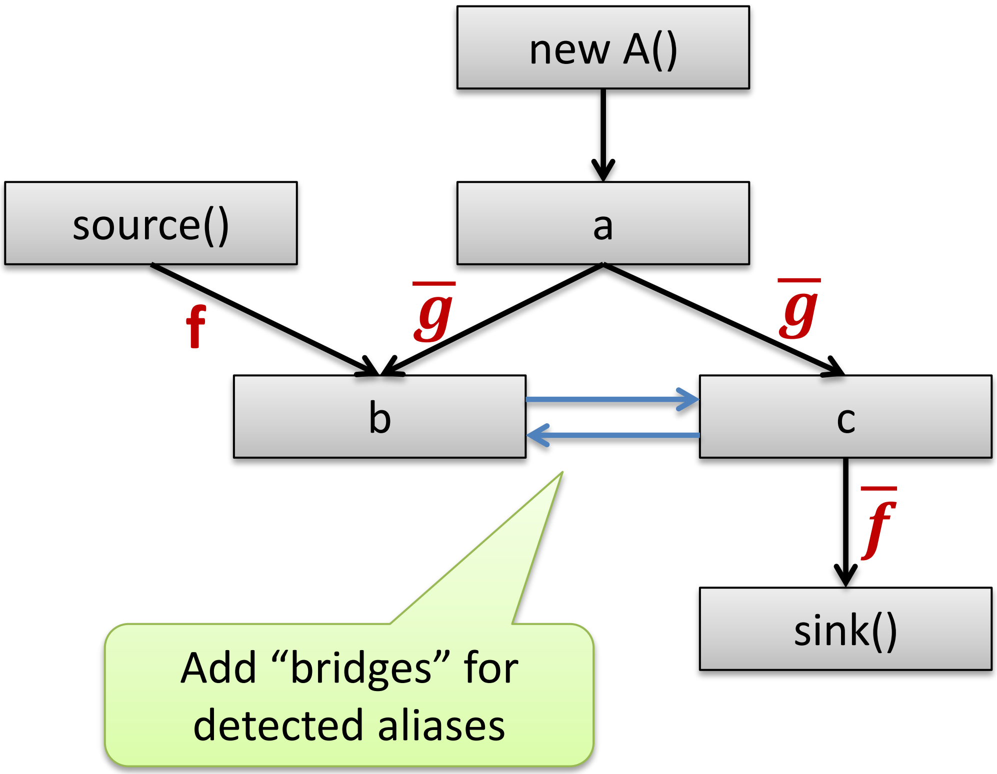
$B \to fC \mid gD \mid BB \mid \epsilon$

$C \to B\overline{f}$

$D \to B\overline{g}$

Aliasing

```
main() {
        a = new A();
        b = a.g;
        b.f = source();
        c = a.g;
        sink(c.f);
}
```



Context- and Field-Sensitive Analysis

Context-Sensitive: $A \rightarrow (A) \mid [A] \mid AA \mid \epsilon \mid fA \mid gA \mid A\overline{f} \mid A\overline{g}$

Field-Sensitive: $B \rightarrow fB\overline{f} \mid gB\overline{g} \mid BB \mid \epsilon \mid (B \mid [B \mid B) \mid B]$

Context- and Field-Sensitive: $L(A) \cap L(B)$

In general, intersection of context-free languages is an undecidable problem.

Should be a valid path: $[f(g)\overline{gf}]$

context- and field-sensitive analysis is proven to be undecidable:

Thomas Reps: Undecidability of Context-Sensitive Data-Dependence Analysis. TOPLAS 2000.

IDE Framework

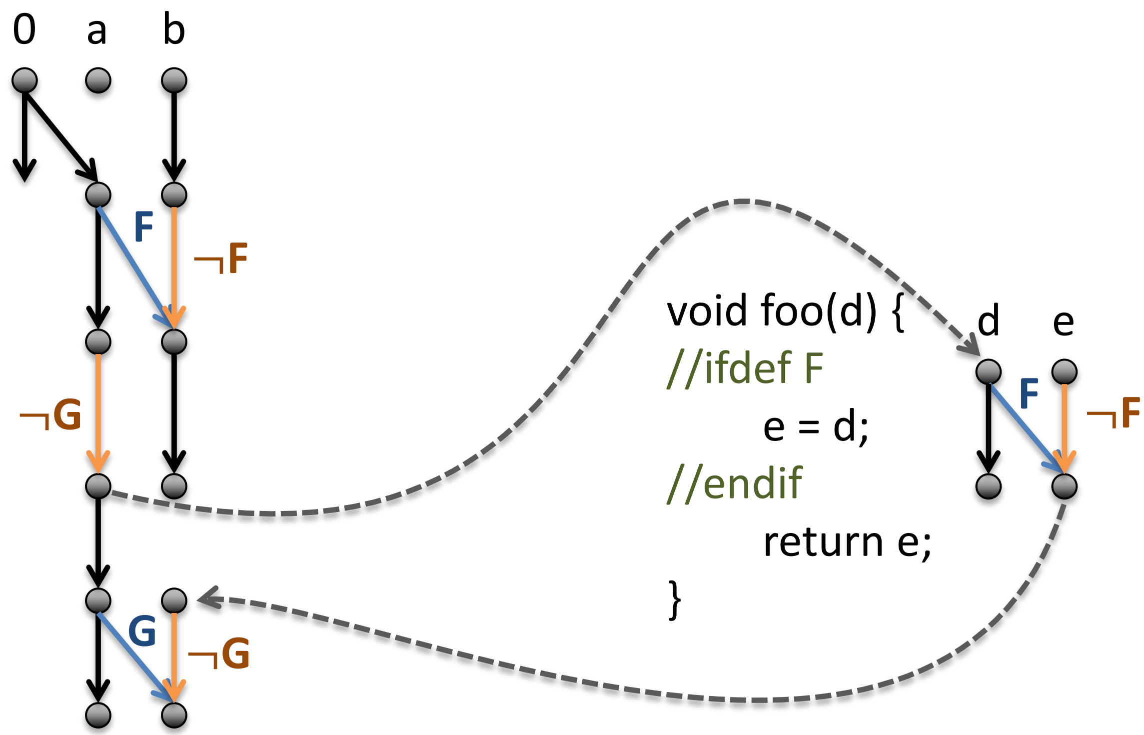
Applied Static Analysis 2016

Johannes Lerch
Dr. Michael Eichberg, Ben Hermann, Sebastian Proksch, Karim Ali Ph.D.

Mooly Sagiv , Thomas Reps, and Susan Horwitz : Precise interprocedural dataflow analysis with applications to constant propagation. TAPSOFT '95

SPL^{LIFT}

```
void main() {
        a = source();
//ifdef F
        b = a;
//endif
//ifdef G
        a = null;
//endif
        b = foo(a);
//ifdef G
        b = a;
//endif
        sink(b);
}
```




```
void foo(d) {
//ifdef F
        e = d;
//endif
        return e;
}
```

Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini: SPL^{LIFT}: statically analyzing software product lines in minutes instead of years. PLDI‘13

Interprocedural, Distributive, Environment Problems

Environment Transformer / Edge Function:

$$e: V \rightarrow V$$

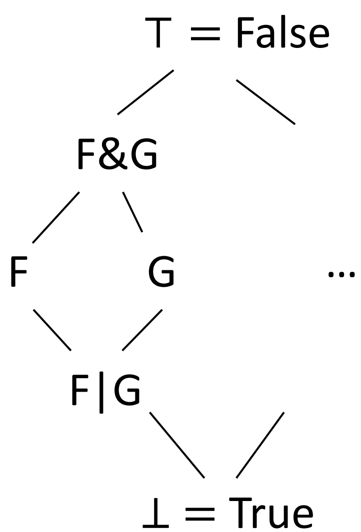
Finite height semi-lattice

distributive function

Characteristics:

- meet operator can be chosen arbitrarily
- computes composition of edge functions
  (always possible, but ideally generates early results)

Lattice used in SPL^{LIFT}



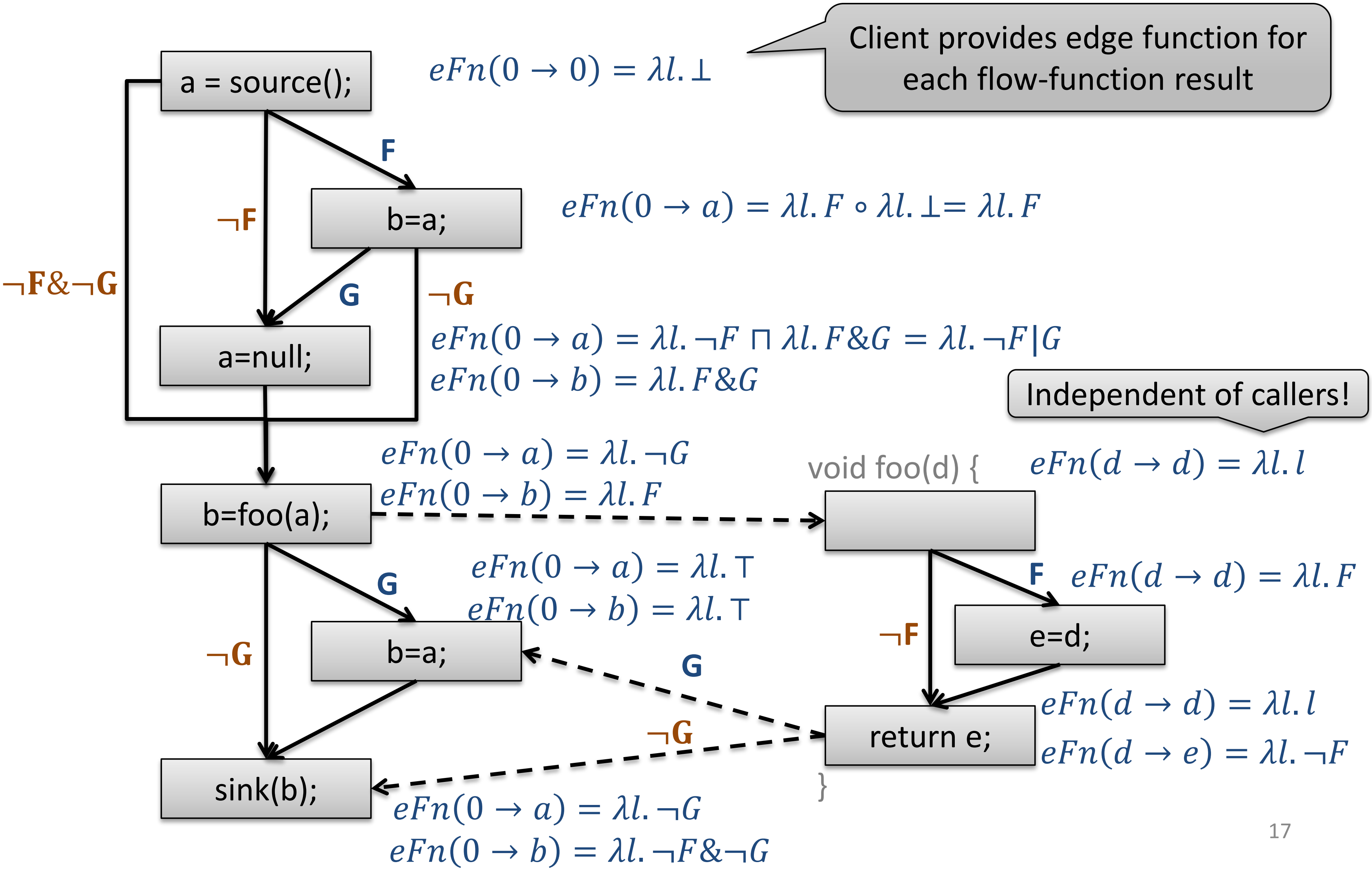
all edges initialized with ⊤

move down when joining facts

move up when composing functions

at initial seeds we use ⊥

SPL^{LIFT}


Client provides edge function for each flow-function result
$eFn(0 \to 0) = \lambda l.\bot$
a = source();
F
b=a;
$eFn(0 \to a) = \lambda l.F \circ \lambda l.\bot = \lambda l.F$
¬F
¬F&¬G
G
¬G
a=null;
$eFn(0 \to a) = \lambda l.\neg F \sqcap \lambda l.F\&G = \lambda l.\neg F|G$
$eFn(0 \to b) = \lambda l.F\&G$
Independent of callers!
$eFn(0 \to a) = \lambda l.\neg G$
$eFn(0 \to b) = \lambda l.F$
void foo(d) {
$eFn(d \to d) = \lambda l.l$
b=foo(a);
G
$eFn(0 \to a) = \lambda l.\top$
$eFn(0 \to b) = \lambda l.\top$
F
$eFn(d \to d) = \lambda l.F$
e=d;
¬F
¬G
b=a;
G
$eFn(d \to d) = \lambda l.l$
¬G
return e;
$eFn(d \to e) = \lambda l.\neg F$
sink(b);
}
$eFn(0 \to a) = \lambda l.\neg G$
$eFn(0 \to b) = \lambda l.\neg F\&\neg G$

IDE-Exercise

Extend the Analysis built in the IFDS Exercise to an IDE Analysis that Considers Correlated Calls

Correlated Method Calls

Could be type B or C at runtime

```
main() {
    A a = unknown();
    x = source();
    b = a.foo(x);
    c = a.bar(x);
    sink(c);
}
```

```
interface A {
    X foo(X);
    X bar(X);
}
```

```
class B implements A {
    X foo(X x) {
        return x;
    }

    X bar(X x) {
        return null;
    }
}
```

```
class C implements A {
    X foo(X x) {
        return null;
    }

    X bar(X x) {
        return x;
    }
}
```

Marianna Rapoport, Ondřej Lhoták, and Frank Tip: Precise Data Flow Analysis in the Presence of Correlated Method Calls. SAS’15

Task

- Use Edge Functions to track the upper type boundaries of variables used as receiver

```
main() {
    A a = unknown();
    x = source();
    b = a.foo(x);
    c = a.bar(x);
    sink(c);
}
```

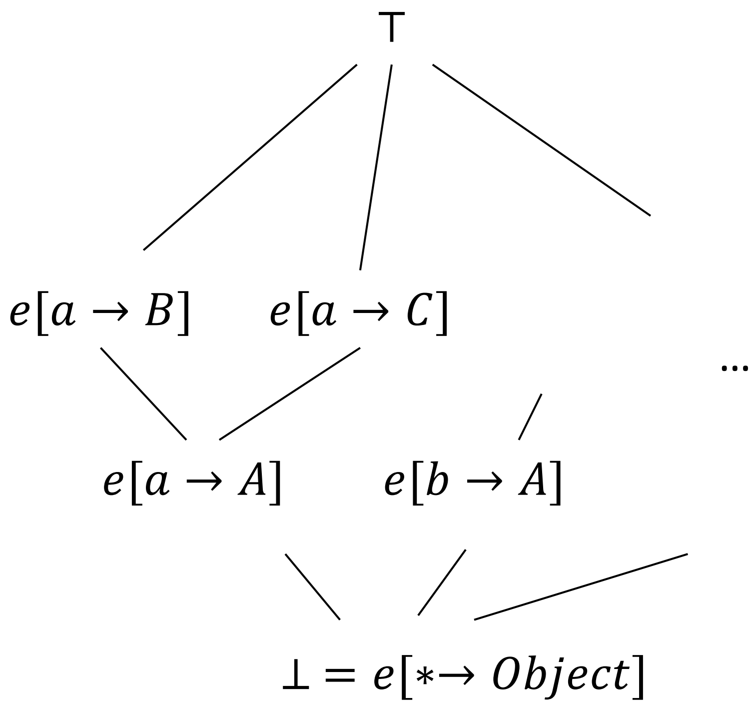
$\lambda e.\, e[a \to B]$

$\lambda e.\, e[a \to C] \circ \lambda e.\, e[a \to B] = \lambda e.\, T$

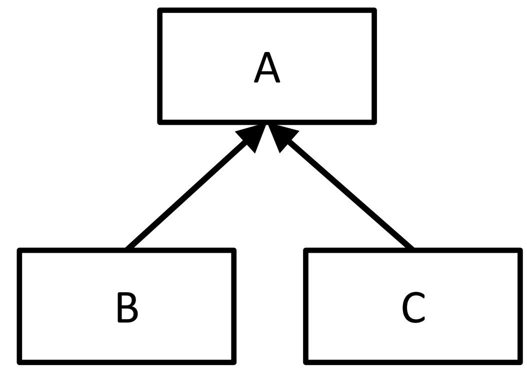
```
class B implements A {
    X foo(X x) {
        return x;
    }

    X bar(X x) {
        return null;
    }
}
```

Lattice



$$\lambda e.\, e[a \to B] \circ \lambda e.\, e[a \to A] = \lambda e.\, e[a \to B]$$
$$\lambda e.\, e[a \to B] \circ \lambda e.\, e[a \to C] = \lambda e.\, \top$$
$$\lambda e.\, e[a \to B] \sqcap \lambda e.\, e[a \to C] = \lambda e.\, e[a \to A]$$



Partial-order function
is the subtype relation for
each mapped variable

EdgeFunctions Interface

```
public interface EdgeFunctions<N, D, M, V> {
	public EdgeFunction<V> getNormalEdgeFunction(N curr, D currNode,
			N succ, D succNode);

	public EdgeFunction<V> getCallEdgeFunction(N callStmt, D srcNode,
			M destinationMethod, D destNode);

	public EdgeFunction<V> getReturnEdgeFunction(N callSite, M calleeMethod,
			N exitStmt, D exitNode, N returnSite, D retNode);

	public EdgeFunction<V> getCallToReturnEdgeFunction(N callSite, D callNode,
			N returnSite, D returnSideNode);
}
```