IFDS-Exercise Set-Up

Compiling OPAL may take some time, therefore start with the set up now, if not already done.

git clone https://bitbucket.org/delors/opal.git
git clone https://github.com/Sable/heros.git
git clone https://github.com/stg-tud/apsa.git
cd opal
git checkout develop
sbt publishLocal
cd ../heros
cp ant.settings.template ant.settings
mkdir javadoc
ant publish-Local
cd ../apsa/2016/ifds/ifds-exercise
sbt eclipse
Import projects IFDS-exercise and IFDS-testcases in Eclipse
Verify set-up: should compile without errors, some tests should succeed

From within Eclipse select Run As → Ant Build… on the build.xml file

IFDS Framework

Applied Static Analysis 2016

Johannes Lerch

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

Thomas Reps, Susan Horwitz, and Mooly Sagiv: Precise Interprocedural Dataflow Analysis via Graph Reachability. PoPL'95

Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez: Practical Extensions to the IFDS Algorithm. CC'10

A Framework for Interprocedural, Finite, Distributive, Subset Problems

Inputs to the Framework:

– Interprocedural Control-Flow Graph (ICFG)

– Flow Function for each ICFG-Edge

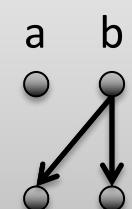
$$f : S \to \mathcal{P}(S)$$

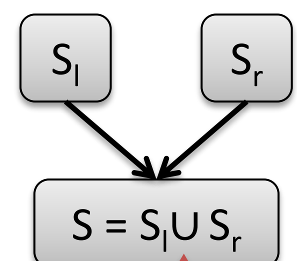
finite set of data-flow facts

distributive function

Example for: a=b;

$$\lambda S. \quad if(b \in S) \quad then\ S \cup \{a\} \quad else\ S \setminus \{a\}$$

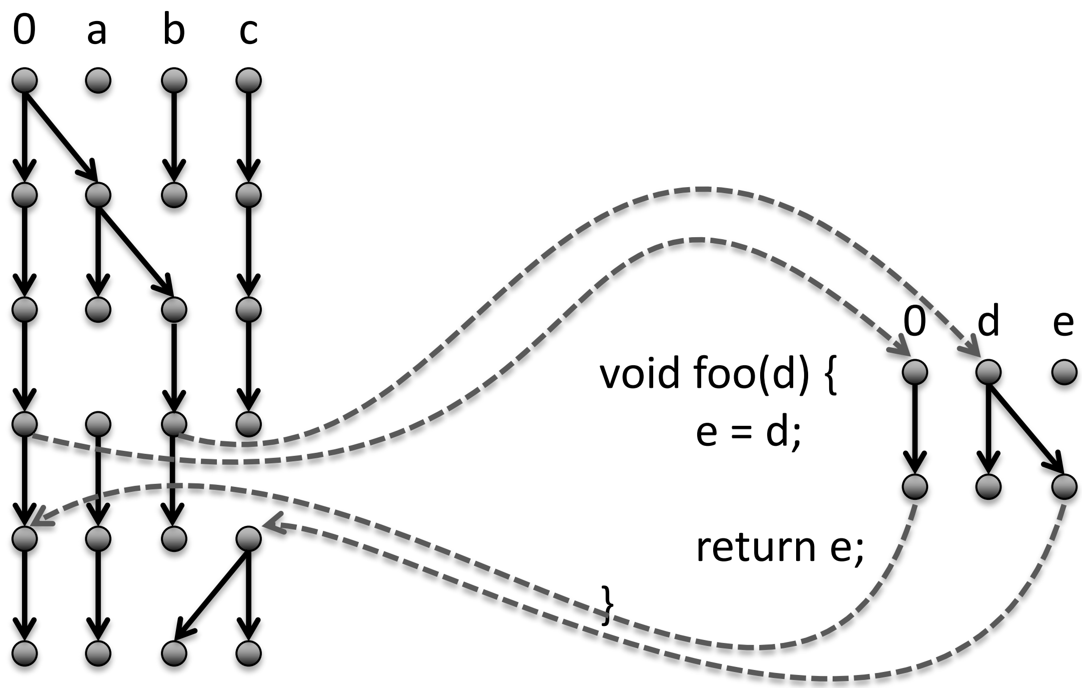




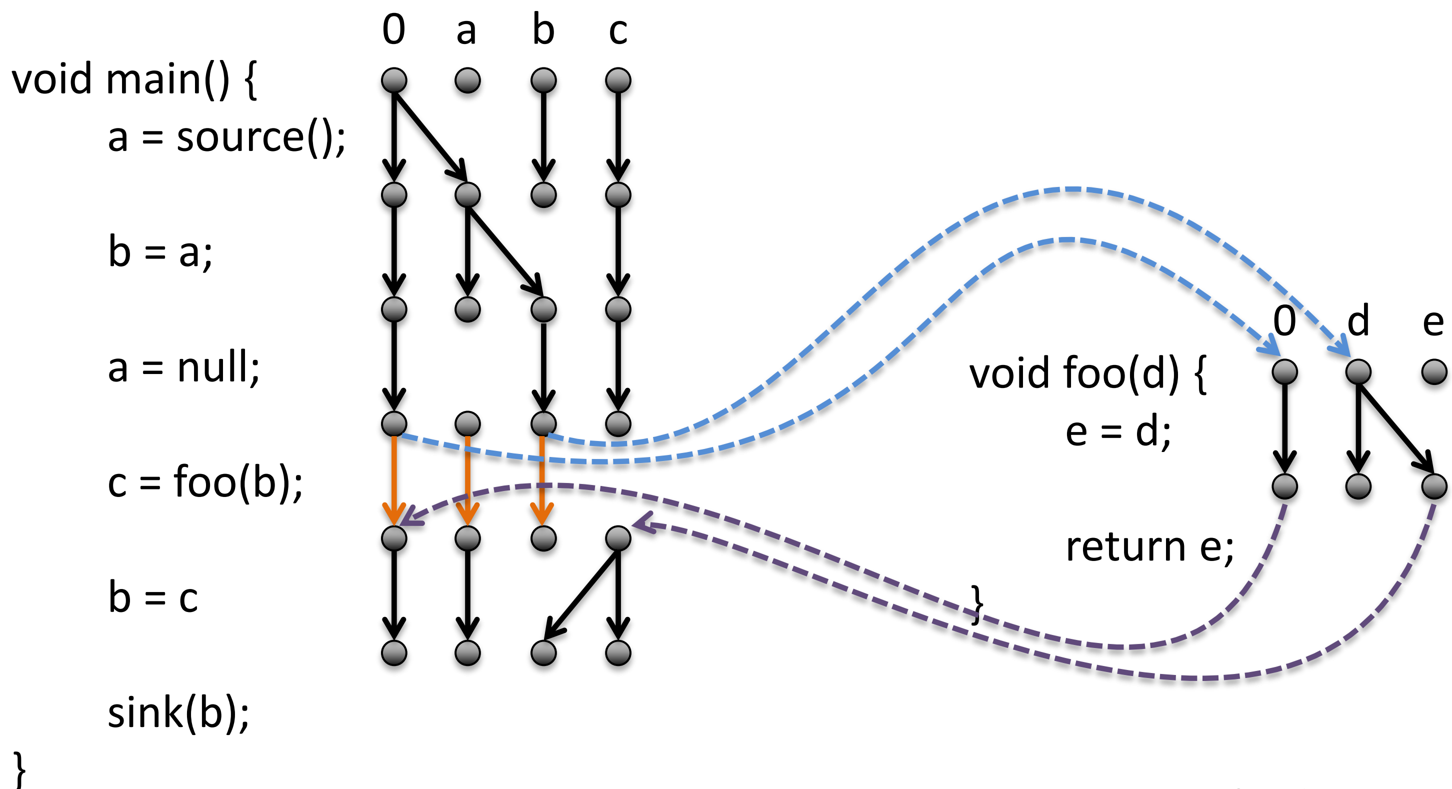
joins subsets

Graph Reachability

```
void main() {
        a = source();

        b = a;

        a = null;

        c = foo(b);

        b = c

        sink(b);
}
```



Four Types of Edges

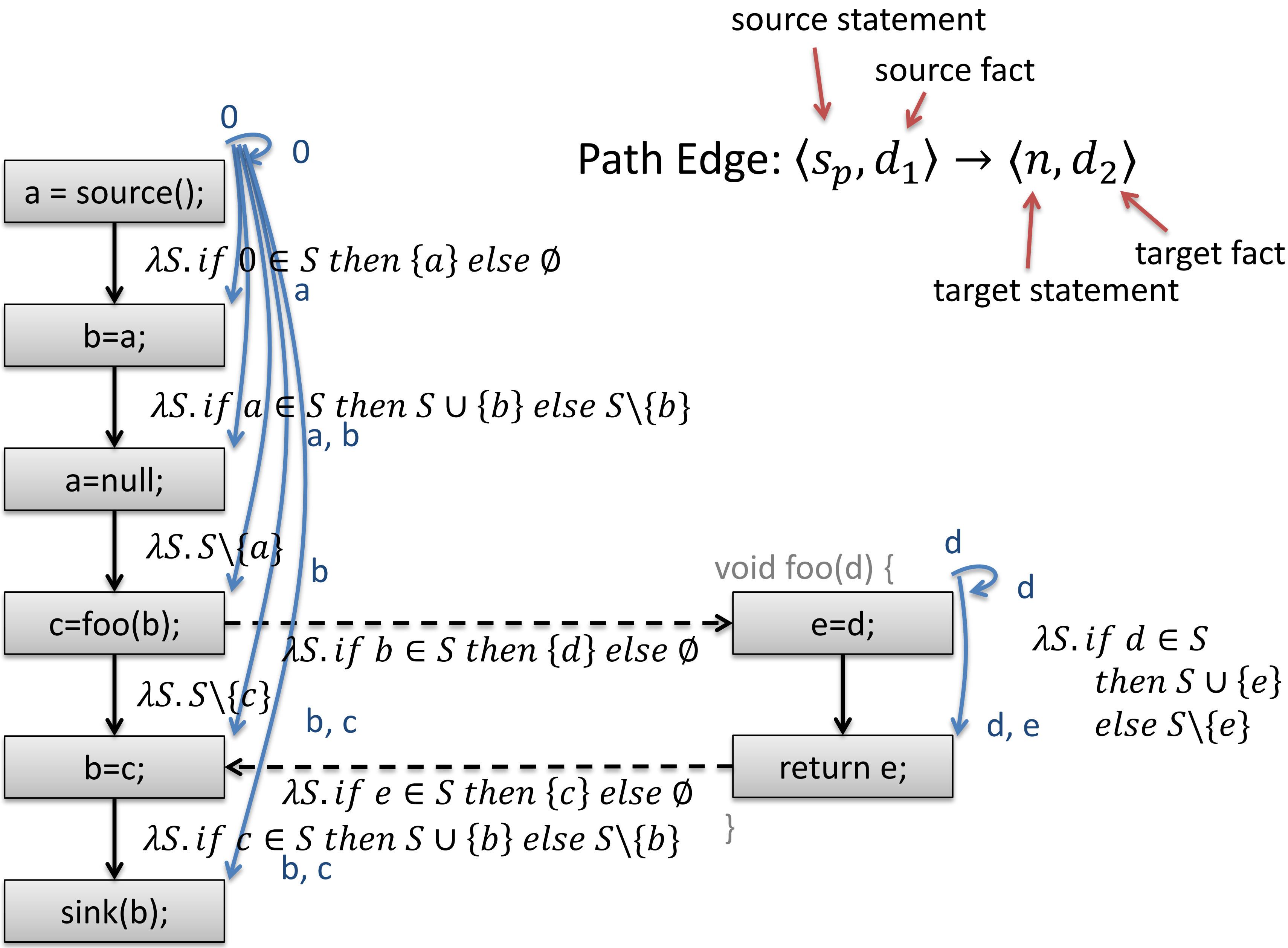


Normal Edge

Call Edge

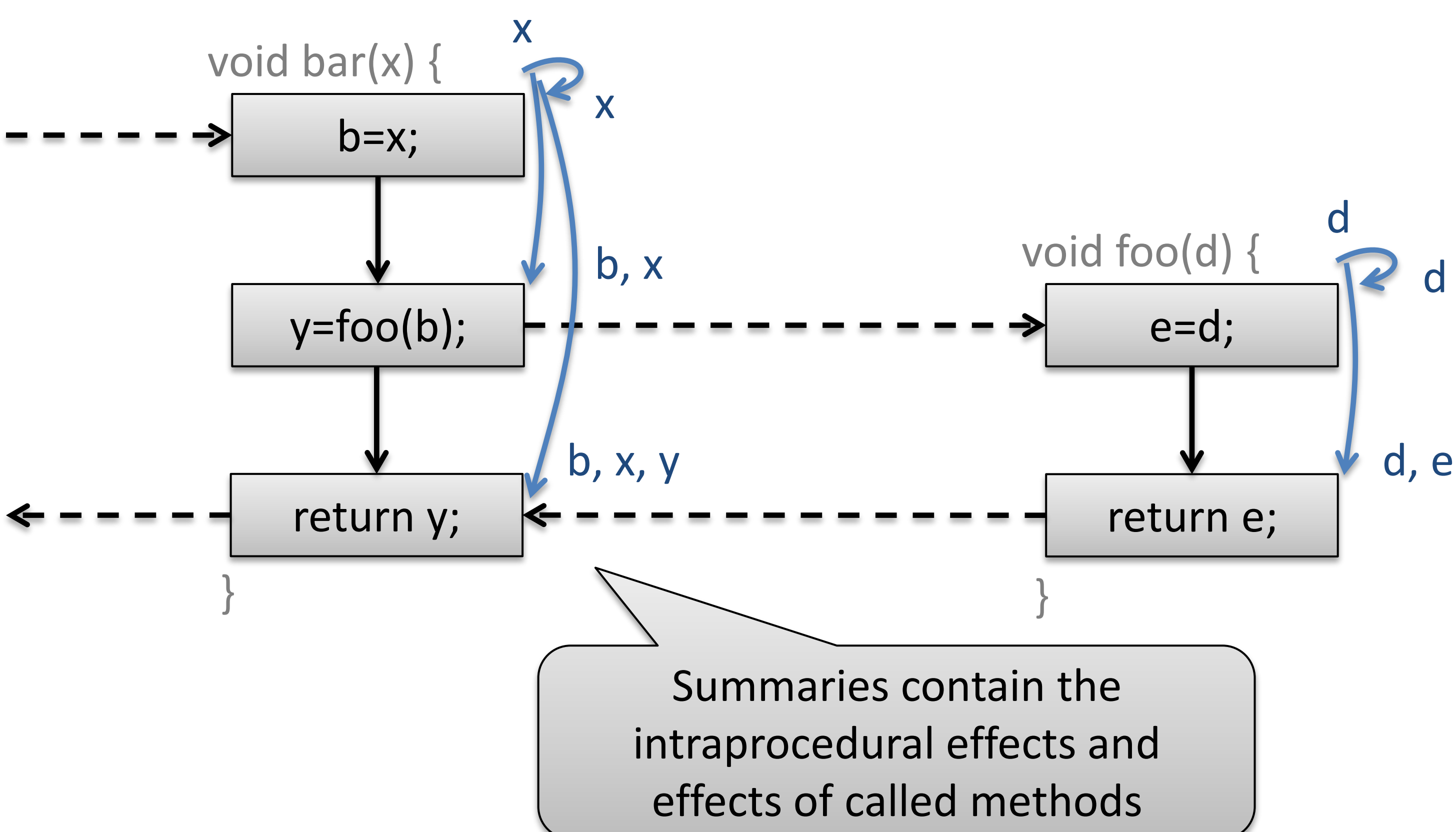
Return Edge

Call-to-Return Edge

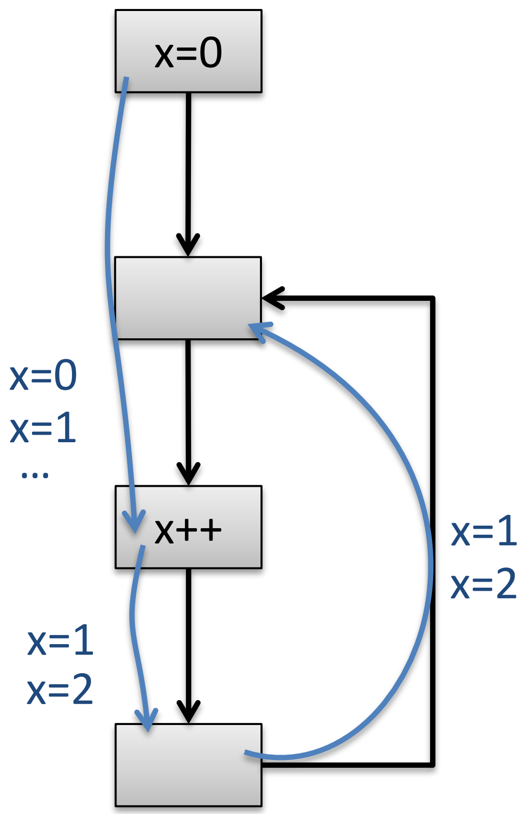


source statement
source fact
Path Edge: $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$
target statement
target fact
0
0
a = source();
$\lambda S. if\ 0 \in S\ then\ \{a\}\ else\ \emptyset$
a
b=a;
$\lambda S. if\ a \in S\ then\ S \cup \{b\}\ else\ S \backslash \{b\}$
a, b
a=null;
$\lambda S. S \backslash \{a\}$
b
c=foo(b);
$\lambda S. if\ b \in S\ then\ \{d\}\ else\ \emptyset$
$\lambda S. S \backslash \{c\}$
b, c
b=c;
$\lambda S. if\ e \in S\ then\ \{c\}\ else\ \emptyset$
$\lambda S. if\ c \in S\ then\ S \cup \{b\}\ else\ S \backslash \{b\}$
b, c
sink(b);
void foo(d) {
d
d
e=d;
$\lambda S. if\ d \in S\ then\ S \cup \{e\}\ else\ S \backslash \{e\}$
d, e
return e;
}

Interprocedural Analysis



Finite Domain of Data-Flow Facts

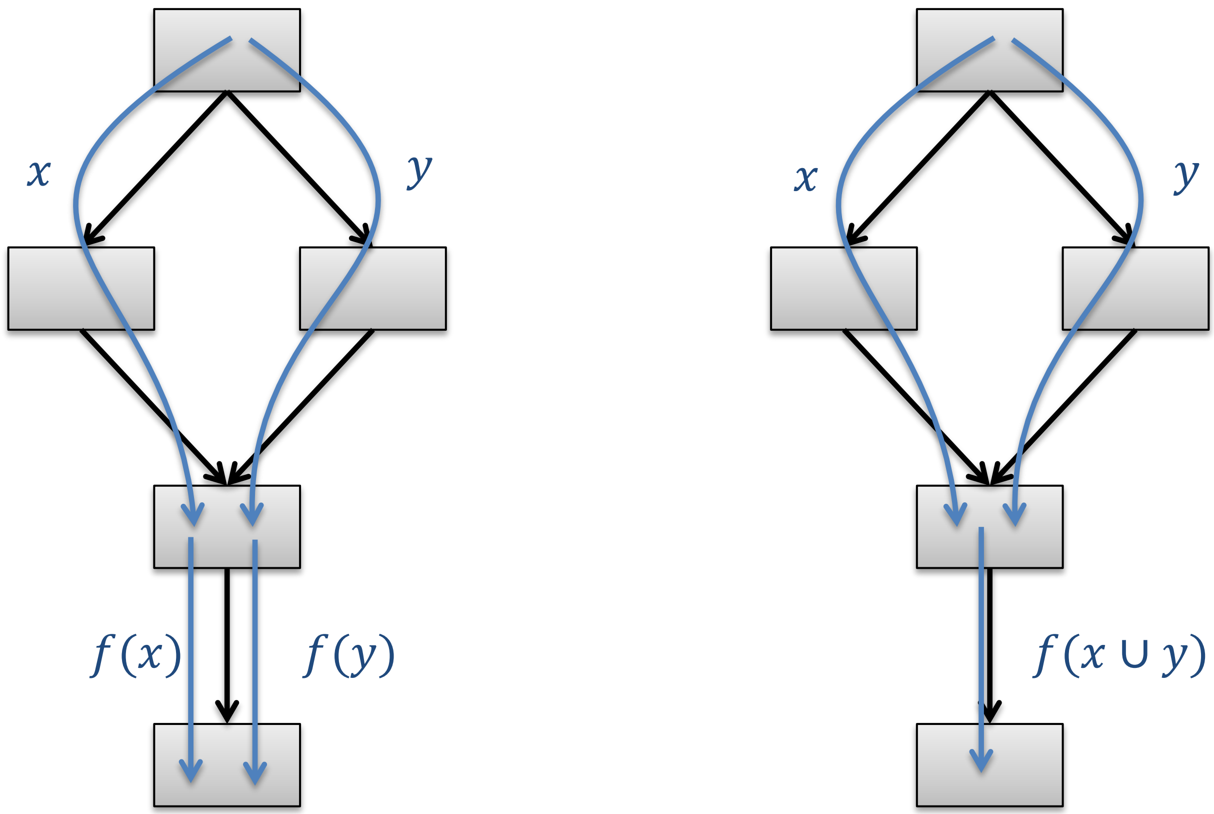


Range of Integer is bound, thus domain is theoretically finite

Eventually, no new path edge will be created and the algorithm terminates

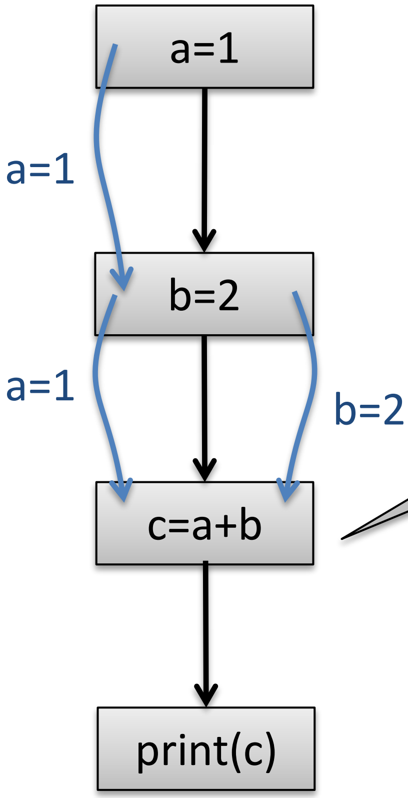
However, in practice this will run out of memory

Distributive Flow Functions



$$f(x) \cup f(y) \quad = \quad f(x \cup y)$$

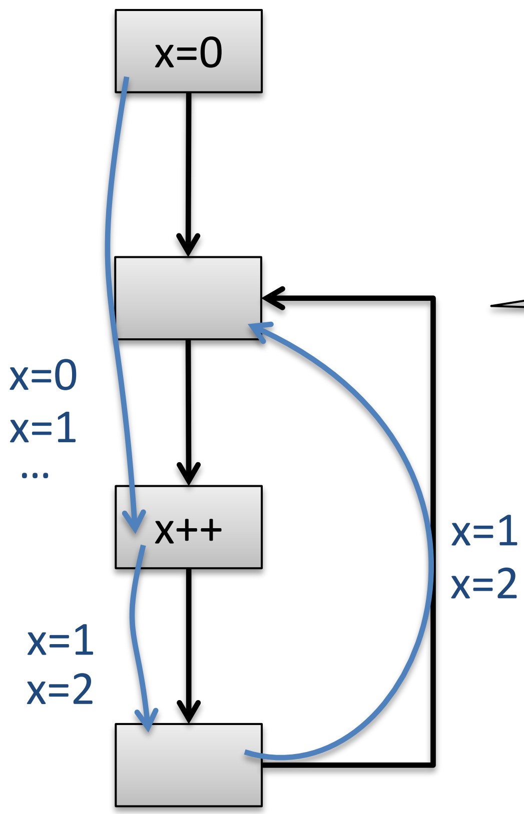
Subset Problem



IFDS cannot infer that c is 3,
if we track a and b independently

$$f : S \to \mathcal{P}(S)$$

Subset Problem (2)



Meet operator is limited to **union** or intersection

x=0 ⊓ x=1

x=[0,1]

x≥0

Impossible as outcome of meet

Implicitly possible, but tracked as two independent facts

```
procedure ForwardTabulateSLRPs()
begin
  while WorkList ≠ ∅ do
    Select and remove an edge ⟨s_p, d1⟩ -π-> ⟨n, d2⟩ from WorkList
    switch n
      case n ∈ Call_p :
        foreach d3 ∈ passArgs(⟨n, d2⟩) do
          Propagate(⟨s_calledProc(n), d3⟩ -0-> ⟨s_calledProc(n), d3⟩)
          Incoming [⟨s_calledProc(n), d3⟩] ∪= ⟨n, d2⟩
          foreach ⟨e_p, d4⟩ ∈ EndSummary [⟨s_calledProc(n), d3⟩] do
            foreach d5 ∈ returnVal(⟨e_p, d4⟩, ⟨n, d2⟩) do
              Insert ⟨n, d2⟩ → ⟨returnSite(n), d5⟩ into SummaryEdge
            od
          od
        od
        foreach d3 s.t. d3 ∈ callFlow(⟨n, d2⟩) or
                        ⟨n, d2⟩ → ⟨returnSite(n), d3⟩ ∈ SummaryEdge do
          Propagate(⟨s_p, d1⟩ -n-> ⟨returnSite(n), d3⟩)
        od
      end case
      case n ∈ e_p :
        EndSummary [⟨s_p, d1⟩] ∪= ⟨e_p, d2⟩
        foreach ⟨c, d4⟩ ∈ Incoming [⟨s_p, d1⟩] do
          foreach d5 ∈ returnVal(⟨e_p, d2⟩, ⟨c, d4⟩) do
            if ⟨c, d4⟩ → ⟨returnSite(c), d5⟩ ∉ SummaryEdge then
              Insert ⟨c, d4⟩ → ⟨returnSite(c), d5⟩ into SummaryEdge
              foreach d3 s.t. ⟨s_procOf(c), d3⟩ → ⟨c, d4⟩ ∈ PathEdge do
                Propagate(⟨s_procOf(c), d3⟩ -c-> ⟨returnSite(c), d5⟩)
              od
            fi
          od
        od
      end case
      case n ∈ (N_p − Call_p − {e_p}) :
        foreach m, d3 s.t. n → m ∈ CFG and d3 ∈ flow(⟨n, d2⟩, π) do
          Propagate(⟨s_p, d1⟩ -n-> ⟨m, d3⟩)
        od
      end case
    end switch
  od
end
```

For a complete view of the algorithm, check Figure 2 and Figure 4 of Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez: Practical Extensions to the IFDS Algorithm. C’10

Static-Analysis Buzzword Bingo

- Flow-Sensitive ✓
- Context-Sensitive ✓
- Field-Based / Field-Sensitive

depends on chosen domain

Field-Based Tracking

$$Domain = Locals \cup Fields$$

$0 \rightarrow 0$

a = source();

$0 \rightarrow a$

b = new DS();

$0 \rightarrow a$

c = new DS();

$0 \rightarrow a$

b.f = a;

$0 \rightarrow a \quad 0 \rightarrow DS.f$

d = c.f

$0 \rightarrow a \quad 0 \rightarrow DS.f \quad 0 \rightarrow d$

sink(d);

Field-Sensitive Tracking

$$Domain = \{l.f_1.f_2.\ldots.\mathrm{f_n}\}$$
$$l \in Locals,$$
$$f_i \in Fields,$$
$$\mathrm{n} \geq 0$$

$0 \rightarrow 0$

a = source();

$0 \rightarrow a$

b = new DS();

$0 \rightarrow a$

c = new DS();

$0 \rightarrow a$

b.f = a;

$0 \rightarrow a \quad 0 \rightarrow b.f$

d = c.f

$0 \rightarrow a \quad 0 \rightarrow b.f$

sink(d);

Field-Sensitive Tracking (2)

$$Domain = \{l.f_1.f_2.\ldots.\mathrm{f_n}\}$$

$l \in Locals,$

$f_i \in Fields,$

$\mathrm{n} \geq 0$

Domain is not finite

$0 \rightarrow 0$

a = source();

$0 \rightarrow a$

while(random()) {

$0 \rightarrow a$ $\quad 0 \rightarrow a.f$ $\quad 0 \rightarrow b.f$ $\quad \ldots$

    b = new DS();

$0 \rightarrow a$ $\quad 0 \rightarrow a.f$

    b.f = a;

$0 \rightarrow a$ $\quad 0 \rightarrow b.f$ $\quad 0 \rightarrow a.f$ $\quad 0 \rightarrow b.f.f$

    a = b;

} $\quad 0 \rightarrow a.f$ $\quad 0 \rightarrow b.f$ $\quad 0 \rightarrow a.f.f$ $\quad 0 \rightarrow b.f.f$

sink(a);

k-limiting

$$Domain = \{l.f_1.f_2.\ldots.\mathrm{f_n}\}$$
$$l \in Locals,$$
$$f_i \in Fields,$$
$$0 \le \mathrm{n} \le k$$

Domain is now finite

$0 \rightarrow 0$

```
a = source();
```

$0 \rightarrow a$

```
while(random()) {
```

$0 \rightarrow a$ $\quad 0 \rightarrow a.f$ $\quad 0 \rightarrow b.f$ $\quad \ldots$

```
    b = new DS();
```

$0 \rightarrow a$ $\quad 0 \rightarrow a.f$

```
    b.f = a;
```

$0 \rightarrow a$ $\quad 0 \rightarrow b.f$ $\quad 0 \rightarrow a.f$ $\quad 0 \rightarrow b.f.f$

```
    a = b;
}
```

$0 \rightarrow a.f$ $\quad 0 \rightarrow b.f$ $\quad 0 \rightarrow a.f.f$ $\quad 0 \rightarrow b.f.f$

```
sink(a);
```

k-limiting (2)

$$Domain = \{l.f_1.f_2.\dots.\mathrm{f_n}\}$$
$$l \in Locals,$$
$$f_i \in Fields,$$
$$0 \leq \mathrm{n} \leq k$$

Domain is not sound!

$0 \rightarrow 0$

a = source();

$0 \rightarrow a$

b = new DS();

$0 \rightarrow a$

c = new DS();

$0 \rightarrow a$

b.f = a;

…

c.f = b;

d = c.f;

e = d.f;

sink(e);

For $k \geq 2$:

$0 \rightarrow b.f$

$0 \rightarrow c.f.f$

$0 \rightarrow d.f$

$0 \rightarrow e$

For $k = 1$:

$0 \rightarrow b.f$

$0 \rightarrow c.f$

$0 \rightarrow d$

k-limiting (3)

$$Domain = \{l.f_1.f_2.\ldots.\mathrm{f_n}.\mathrm{w}\}$$
$$l \in Locals,$$
$$f_i \in Fields,$$
$$0 \leq \mathrm{n} \leq k,$$
$$w = [*]?$$

```
a = source();
b = new DS();
c = new DS();
b.f = a;
c.f = b;
d = c.f;
e = d.f;
sink(e);
```

$0 \to 0$

$0 \to a$

$0 \to a$

$0 \to a$

...

<u>For $\boldsymbol{k \geq 2}$:</u>	<u>For $\boldsymbol{k = 1}$:</u>
$0 \to b.f$	$0 \to b.f$
$0 \to c.f.f$	$0 \to c.f.*$
$0 \to d.f$	$0 \to d.*$
$0 \to e$	$0 \to e.*$

k-limiting (4)

$$Domain = \{l.f_1.f_2.\dots.\mathrm{f_n}.\mathrm{w}\}$$
$$l \in Locals,$$
$$f_i \in Fields,$$
$$0 \le \mathrm{n} \le k,$$
$$w = [*]?$$

$0 \rightarrow 0$

a = source();

$0 \rightarrow a$

b = new DS();

$0 \rightarrow a$

c = new DS();

$0 \rightarrow a$

b.f = a;

…

c.f = b;

d = c.f;

e = d.g;

sink(e);

For $k \ge 2$:

$0 \rightarrow b.f$

$0 \rightarrow c.f.f$

$0 \rightarrow d.f$

For $k = 1$:

$0 \rightarrow b.f$

$0 \rightarrow c.f.*$

$0 \rightarrow d.*$

$0 \rightarrow e.*$

over-approximation may yield false positives

IFDS-Exercise

Implement a simple Taint Analysis using Heros' IFDS-Solver and OPAL

```
public static foo() {
        Object a = source();
        Object b = a;
        sink(b);
}
```

Detect if values returned by **source()** flow as argument into **sink()**

Set Up

git clone https://bitbucket.org/delors/opal.git
git clone https://github.com/Sable/heros.git
git clone https://github.com/stg-tud/apsa.git
cd opal
git checkout develop
sbt publishLocal
cd ../heros
cp ant.settings.template ant.settings
mkdir javadoc
ant publish-Local
cd ../apsa/2016/ifds/ifds-exercise
sbt eclipse
Import projects IFDS-exercise and IFDS-testcases in Eclipse
Verify set-up: should compile without errors, some tests should succeed

From within Eclipse select Run As → Ant Build… on the build.xml file

Quickstart

- Heros implementation of IFDS (https://github.com/sable/heros)
  - Important classes:
    - IFDSSolver / IDESolver
      Implementation of the IFDS framework
    - IFDSTabulationProblem / IDETabulationProblem
      settings & input configuration: ICFG, flow functions
    - FlowFunctions
      provides FlowFunction implementations for edges of the ICFG
    - FlowFunction
      implementation of a flow function

FlowFunctions

N = Statement / Instruction
D = Data-Flow Fact
M = Method

```
public interface FlowFunctions<N, D, M> {

        FlowFunction<D> getNormalFlowFunction(N curr, N succ);

        FlowFunction<D> getCallFlowFunction(N callStmt, M destinationMethod);

        FlowFunction<D> getReturnFlowFunction(N callSite, M calleeMethod,
                N exitStmt, N returnSite);

        FlowFunction<D> getCallToReturnFlowFunction(N callSite, N returnSite);
}

public interface FlowFunction<D> {

        Set<D> computeTargets(D source);
}
```

The Domain

- For simplicity we used some intermediate representation in the previous slides
- Does not match Bytecode using an operand stack

```
public static foo() {
        Object a = source();
        Object b = a;
        sink(b);
}
```

```
public static void foo();
0      invokestatic source()
3      astore_0
4      aload_0
5      astore_1
6      aload_1
7      invokestatic sink(java.lang.Object)
10     return
```

Track registers containing tainted values

Track where on the operand stack the tainted value is

The Domain (2)

```
case class RegisterFact(
	registerIndex: Int,
	opStack: List[StackEntry]
)
```

Track registers containing tainted values

```
case class OperandStackFact(
	stackIndex: Int,
	opStack: List[StackEntry]
)
```

Track where on the operand stack the tainted value is

Track the Operand Stack

```
Object a = source();
DS b = new DS();
DS c = new DS();
b.f = a;
Object d = c.f;
sink(d);
```

```
aload_1 [b]
aload_0 [a]
putfield DS.f
aload_2 [c]
getfield DS.f
astore_3 [d]
```

To taint **b.f** you need to know register 1 [b] is on the operand stack

```
Object a = source();
Object[] arr = new Object[1];
arr[0] = a;
Object b = arr[0];
sink(b);
```

```
aload_1 [arr]
iconst_0
aload_0 [a]
aastore
aload_1 [arr]
iconst_0
aaload
astore_2 [b]
```

Not only required for field-sensitivity, but also for arrays

Some Instructions of Interest

(listed as types of OPAL)

- LoadLocalVariableInstruction(_, localVarIndex)
- StoreLocalVariableInstruction(_, localVarIndex)
- PUTFIELD(declaringClass, fieldName, _)
- PUTSTATIC(declaringClass, fieldName, _)
- GETFIELD(declaringClass, fieldName, _)
- GETSTATIC(declaringClass, fieldName, _)
- ArrayLoadInstruction
- ArrayStoreInstruction
- ReturnValueInstruction
- …