Static Program Analysis

Part 8 – distributive analysis frameworks

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

Anders Møller

Computer Science, Aarhus University

Agenda

- **Distributive analysis**
- IFDS
- IDE

Key ideas

the function summary effect in
interprocedural dataflow analysis

+

compact representations of distributive functions

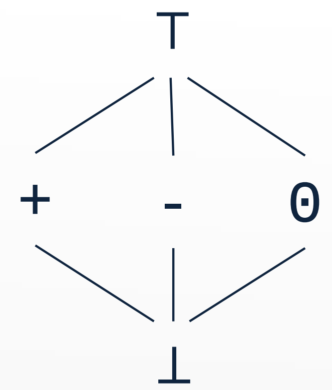
$\Downarrow$

efficient analysis algorithms

Context sensitive dataflow analysis

Recall our context-sensitive interprocedural sign analysis:

- Lattice for abstract values: Sign =



- Lattice for abstract states: States = Vars $\rightarrow$ Sign

- Analysis lattice: $(\text{Contexts} \rightarrow \text{lift(States)})^n$

For each CFG node v we have a map m_v from call contexts to abstract states (or *unreachable*)
"If the current function is called in context c, then the abstract state at v is $m_v(c)$"

Example, revisited:

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↦+]

Possibly-uninitialized variables analysis

(very similar to *taint analysis*)

- Let's make an analysis to detect possibly-uninitialized variables
  - remember the initialized variables analysis?*
- We want
  - flow-sensitivity
  - full context-sensitivity (with the functional approach)
- Lattice of abstract states: States = $\mathcal{P}$(Vars)
- Analysis lattice: (Contexts → lift(States))^n =

$$\boxed{(\mathcal{P}(\text{Vars}) \rightarrow \text{lift}(\mathcal{P}(\text{Vars})))^n}$$

  - as usual, n is the number of CFG nodes
  - recall that the full functional approach has Contexts = States
  - intuitively, the context is the set of possibly uninitialized variables at the entry of the current function

*) In this analysis, a variable is *possibly-uninitialized* if its value may be computed from an uninitialized variable

Possibly-uninitialized variables – example

```
main() {
  var x,y,z;
  x = input;
  z = p(x,y);
  return z;
}

p(a,b) {
  if (a > 0) {
    b = input;
    a = a - b;
    b = p(a,b);
    output(a);
    output(b);
  }
  return b;
}
```

- When `p` is called from `main`,
  `a` is initialized and `b` is uninitialized
- When `p` is called from `p`,
  `a` and `b` are both initialized

- A context-insensitive analysis concludes that `b` may be uninitialized at `output(b)` ☹
- A fully context-sensitive analysis concludes that `b` is definitely initialized at `output(b)` ☺

Possibly-uninitialized variables analysis

A forward, may analysis – context-insensitive version:

- variable declarations, `var` x : $[\![v]\!] = JOIN(v) \cup \{x\}$
- assignments, $x = E$:

  $$t_v(S) = \begin{cases} S \cup \{x\} & \text{if } vars(E) \cap S \neq \emptyset \\ S \setminus \{x\} & \text{otherwise} \end{cases}$$

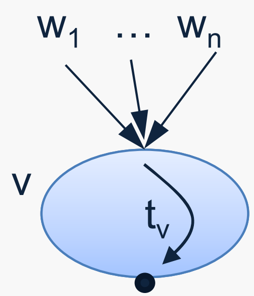
  $$[\![v]\!] = t_v(JOIN(v))$$

- function entries:
- after-call nodes:

  see SPA Section 8.1

- all others: $[\![v]\!] = JOIN(v)$

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



Possibly-uninitialized variables analysis

A forward, may analysis – context-sensitive version:

- variable declarations, `var` x : ...
- assignments, $x = E$:

$$t_v(S) = \begin{cases} S \cup \{x\} & \text{if } vars(E) \cap S \neq \emptyset \\ S \setminus \{x\} & \text{otherwise} \end{cases}$$

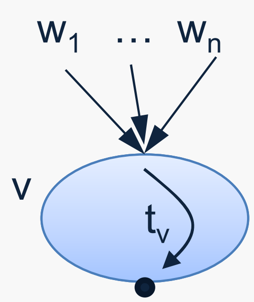
$$[\![v]\!](c) = \begin{cases} t_v(\mathit{JOIN}(v,c)) & \text{if } \mathit{JOIN}(v,c) \in \text{States} \\ \text{unreachable} & \text{if } \mathit{JOIN}(v,c) = \text{unreachable} \end{cases}$$

- program entry: $[\![v]\!](c) \neq$ unreachable
- other function entries:
- after-call nodes:

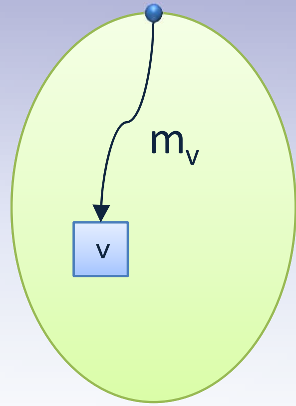
see SPA Section 8.4

- all others: $[\![v]\!](c) = \mathit{JOIN}(v,c)$

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



Pre-analysis



- The analysis lattice is $(\mathrm{lift}(\mathcal{P}(\mathrm{Vars}) \rightarrow \mathcal{P}(\mathrm{Vars})))^n$
- *Idea:* run a *context-insensitive*(!) analysis that computes, for each CFG node v, a map m_v: $\mathcal{P}(\mathrm{Vars}) \rightarrow \mathcal{P}(\mathrm{Vars})$ with the following property:

  If the function containing v is executed in an initial abstract state where $S \subseteq \mathrm{Vars}$ are the possibly-uninitialized variables at the entry, then $m_v(S)$ is the set of possibly-uninitialized variables at v

  The ‘unreachable’ element means that the function containing v is unreachable from the program entry
- If we have such an analysis, then we can easily compute the sets of possibly-uninitialized variables for all CFG nodes (without doing a full context-sensitive analysis)
- It suffices to compute m_v for CFG nodes in reachable functions

Distributive functions and analyses

Exercise 4.20: A function $f : L_1 \to L_2$ where L_1 and L_2 are lattices is *distributive* when $\forall x, y \in L_1 : f(x) \sqcup f(y) = f(x \sqcup y)$.

(a) Show that every distributive function is also monotone.

(b) Show that not every monotone function is also distributive.

Exercise 5.26: An analysis is distributive if all its constraint functions are distributive according to the definition from Exercise 4.20. Show that live variables analysis is distributive.

Is possibly-uninitialized variables analysis distributive?

Distributive functions and analyses

Exercise 5.34: Which among the following analyses are distributive, if any?

(a) Available expressions analysis.

(b) Very busy expressions analysis.

(c) Reaching definitions analysis.

(d) Sign analysis.

(e) Constant propagation analysis.

Exercise 10.6: Recall from Exercise 5.26 that an analysis is distributive if all its constraint functions are distributive. Show that Andersen's analysis is *not* distributive. (Hint: consider the constraint for the statement **x=*y** or ***x=y**.)

Agenda

- Distributive analysis
- **IFDS**
- IDE

IFDS (Interprocedural Finite Distributive Subset problems)

- *Precise Interprocedural Dataflow Analysis via Graph Reachability*, Reps, Horwitz, Sagiv, POPL 1995
- Setting:
  - lattice of abstract states: States = $\mathcal{P}$(D) where D is a finite set (i.e., a powerset lattice)
  - all transfer functions, f_v: States $\rightarrow$ States, are distributive
- Great idea #1:
  - such constraints can be represented compactly!
  - distributivity closed under composition and least upper bound, so function summaries can also be represented compactly and without loss of precision!
- Great idea #2:
  - tabulation solver (building the m_v maps)
- Bonus: can be made demand-driven

Compact representation

- Assume f: $\mathcal{P}(D) \to \mathcal{P}(D)$ where D is a finite set and f is distributive
- A naive representation of f would be a table with $2^{|D|}$ entries
  (if D is, for example, the set of program variables, then such a table is big!)
- f can be decomposed into a function g: $(D \cup \{\bullet\}) \to \mathcal{P}(D)$
  – Define $g(\bullet) = f(\emptyset)$ and $g(d) = f(\{d\}) \setminus f(\emptyset)$ for $d \in D$
  – Now $f(X) = g(\bullet) \cup \bigcup_{y \in X} g(y)$
- Can be represented compactly as a graph with 2(|D|+1) nodes
  – <u>Example:</u>

$$\begin{tikzcd} \bullet \arrow[d] \arrow[dr] & d_1 & d_2 \arrow[dr] & d_3 \arrow[d] \\ \bullet & d_1 & d_2 & d_3 \end{tikzcd}$$

  for $D=\{d_1, d_2, d_3\}$

  means that $g(\bullet) = \{d_1\}$, $g(d_1)=\emptyset$, $g(d_2)=\{d_3\}$, and $g(d_3)=\{d_3\}$
  (the edge from $\bullet$ to $\bullet$ is always present)
  so $f(S) = \{d_1, d_3\}$ if $d_2 \in S$ or $d_3 \in S$, and $f(S) = \{d_1\}$ otherwise

  – In general, the edges are:
  $\{\bullet \rightsquigarrow \bullet\} \cup \{\bullet \rightsquigarrow y \mid y \in f(\emptyset)\} \cup \{x \rightsquigarrow y \mid y \in f(\{x\}) \land y \notin f(\emptyset)\}$

Compact representation

<u>Exercise:</u>

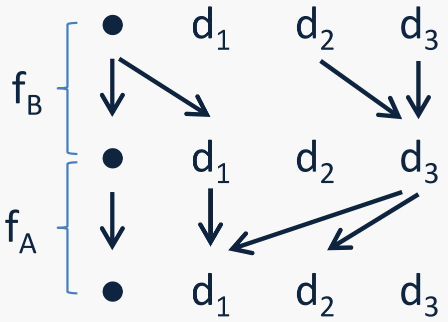
For uninitialized-variables analysis,
what is the IFDS graph representation of

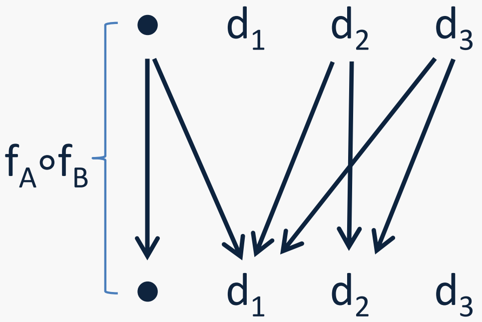
1) an assignment, `X = E`, or

2) a variable declaration, `var X` **?**

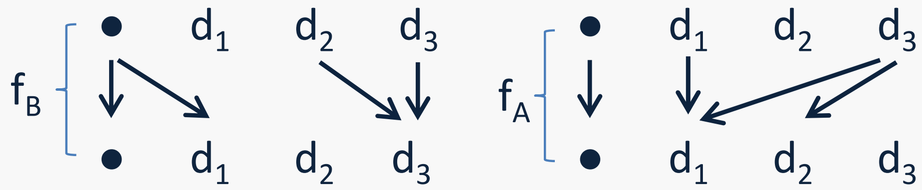
Composition and l.u.b.

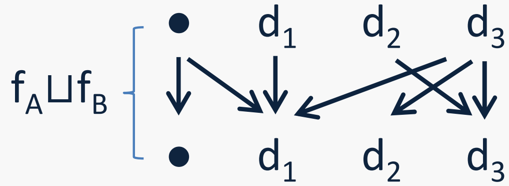
- Distributivity is closed under function composition and l.u.b. Assume f_A: $\mathcal{P}(D) \rightarrow \mathcal{P}(D)$ and f_B: $\mathcal{P}(D) \rightarrow \mathcal{P}(D)$ where D is a finite set and both f and are distributive
  - $f_A \circ f_B$: $\mathcal{P}(D) \rightarrow \mathcal{P}(D)$ is also distributive $(f_A \circ f_B)(S) = f_A(f_B(S))$
  - $f_A \sqcup f_B$: $\mathcal{P}(D) \rightarrow \mathcal{P}(D)$ is also distributive $(f_A \sqcup f_B)(S) = f_A(S) \sqcup f_B(S)$
- Proof? (exercise)
- With the graph representation:





(edges $d_2 \rightarrow d_1$ and $d_3 \rightarrow d_1$ could be omitted)





(edges $d_1 \rightarrow d_1$ and $d_3 \rightarrow d_1$ could be omitted)

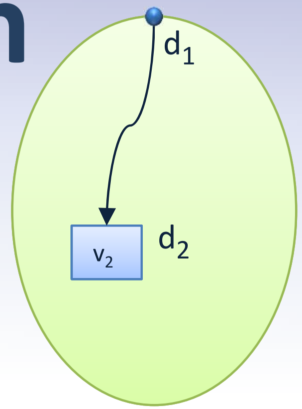
Possibly-uninitialized variables analysis

- The analysis lattice is $(\mathrm{lift}(\mathcal{P}(\mathrm{Vars}) \rightarrow \mathcal{P}(\mathrm{Vars})))^n$
- For each reachable CFG node, the analysis computes an element of

$$\underbrace{\mathcal{P}(\mathrm{Vars})}_{\substack{\text{assuming we have this set of} \\ \text{possibly-uninitialized variables} \\ \text{at the entry of the function...}}} \rightarrow \underbrace{\mathcal{P}(\mathrm{Vars})}_{\substack{\text{...we have this set of} \\ \text{possibly-uninitialized variables at v}}}$$

- With the graph representation, all such functions can be represented compactly and constructed efficiently!
- Using the ordinary worklist algorithm from monotone frameworks amounts to propagating sets of possibly-uninitialized variables for different contexts (Exercise: worst-case time complexity?)
- A smarter approach: ***the tabulation algorithm***

The IFDS Tabulation Algorithm



- The idea: with a worklist algorithm, incrementally build a set of *path edges* $\langle v_1,d_1\rangle \rightsquigarrow \langle v_2,d_2\rangle$ where
  - v_1 is a function entry node, v_2 is a CFG node in the same function as v_1, and $d_1, d_2 \in D \cup \{\bullet\}$
  - the edge means: **if dataflow fact d_1 holds at v_1 then d_2 holds at v_2**
- Only requires function composition and l.u.b.
- At each call node, use the path edges for the return nodes of the function being called as a function summary!
- See pseudo-code in [Reps et al., 1995]
- Worst-case time complexity: $O(|E|\cdot|D|^3)$ where $|E|$ is the number of CFG edges
- After the table is built, it is easy to compute the dataflow facts for any given CFG node

Example [Reps et al., 1995]

```
declare g: integer

program main
begin
  declare x: integer
  read(x)
  call P(x)
end


procedure P(value a : integer)
begin
  if (a > 0) then
    read(g)
    a := a − g
    call P(a)
    print(a, g)
  fi
end
```

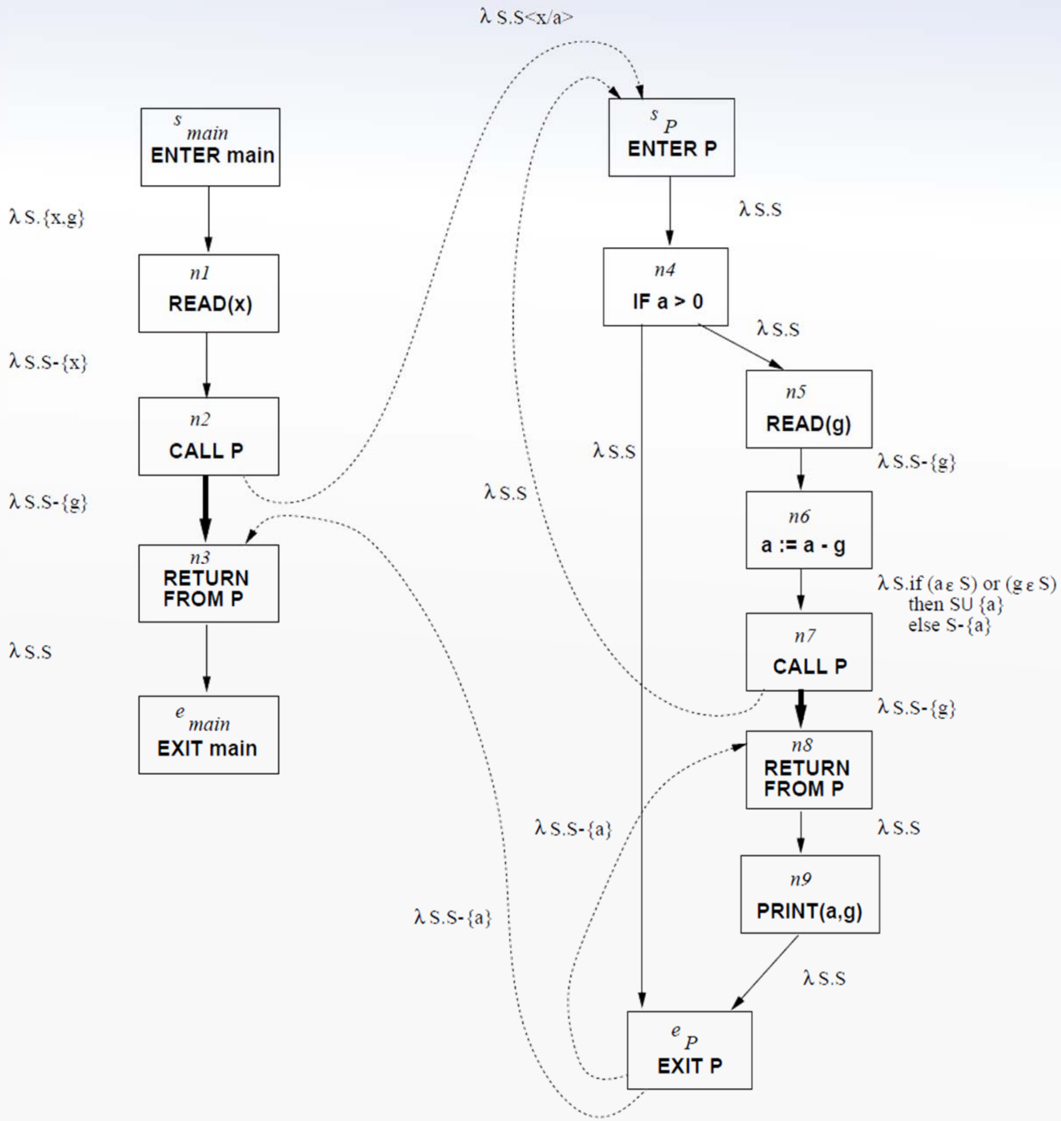
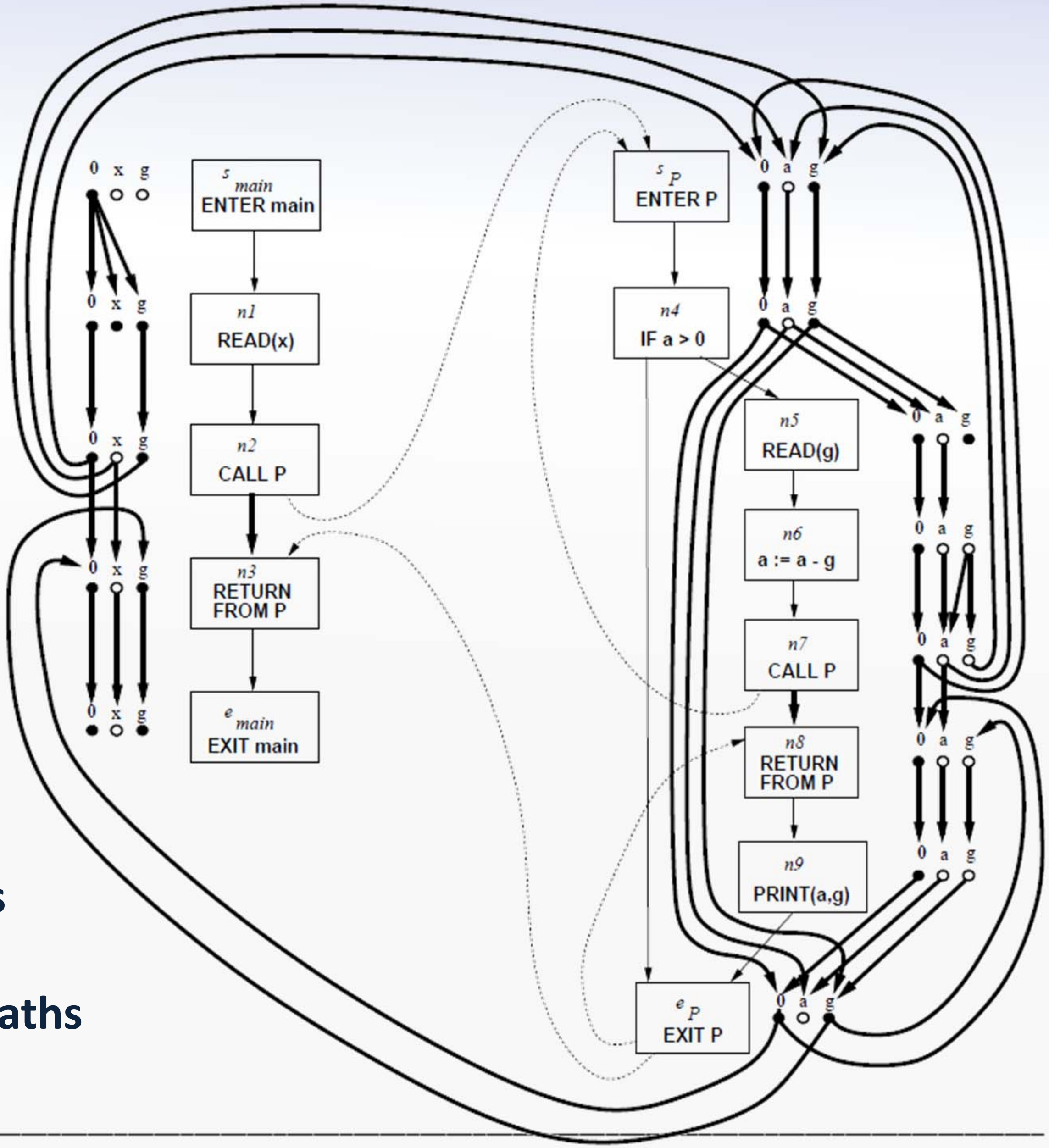


Figure 1. An example program and its supergraph G^*. The supergraph is annotated with the dataflow functions for the "possibly-uninitialized variables" problem. The notation S<x/a> denotes the set *S* with *x* renamed to *a*.

Example [Reps et al., 1995]

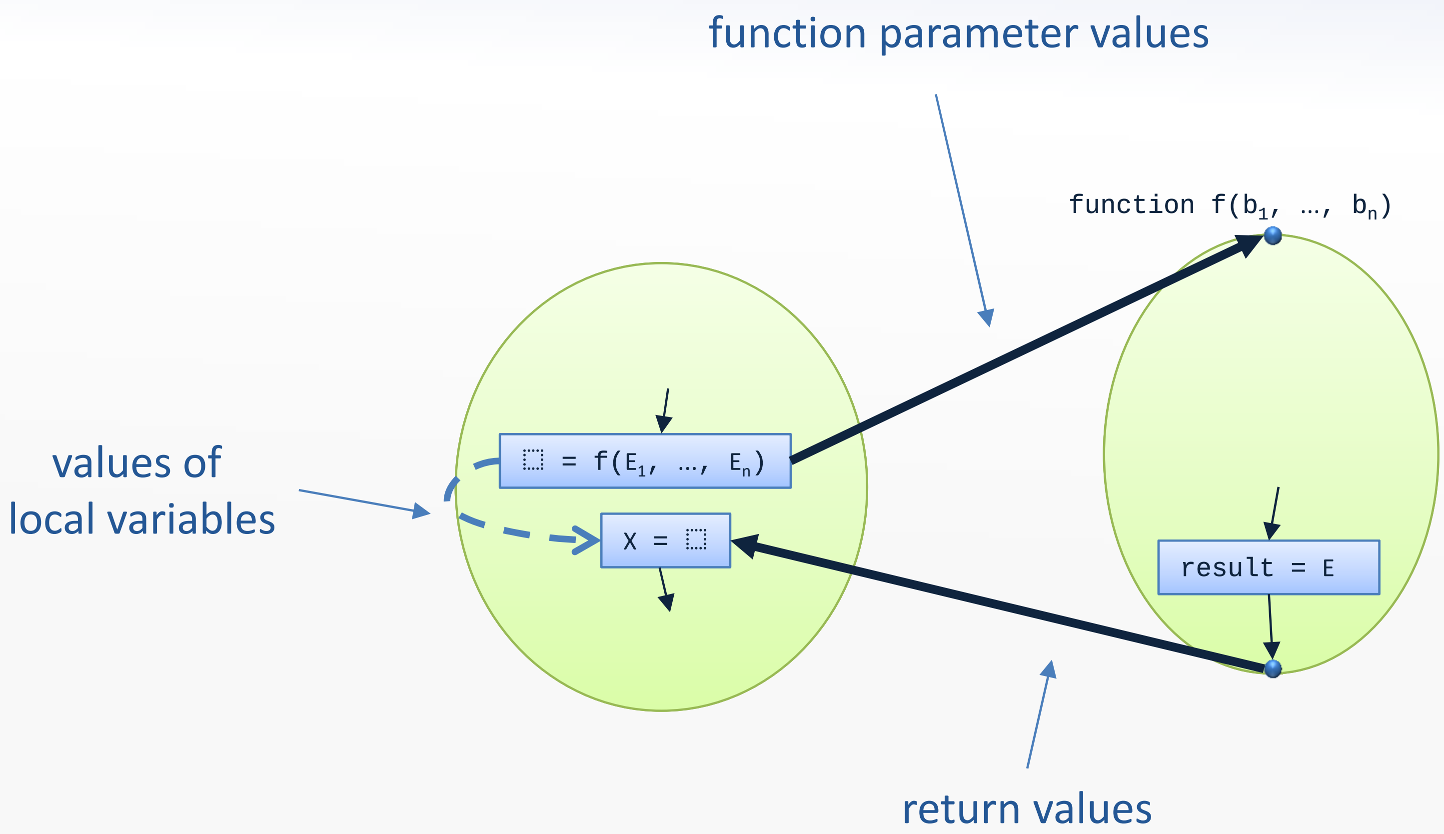


Computing the possibly-uninitialized variables amounts to finding realizable (i.e., interprocedurally valid) paths in this graph!

Figure 2. The exploded supergraph that corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 1. Closed circles represent nodes of $G^{\#}_{IP}$ that are reachable along realizable paths from $\langle s_{main}, 0\rangle$. Open circles represent nodes not reachable along such paths.

(the paper uses 0 instead of ●)

Dataflow at function calls



IFDS constraint-based specification
Phase 1

- **E** represents the program being analyzed:
  $\langle v_1,d_1\rangle \rightsquigarrow \langle v_2,d_2\rangle \in$ **E** means that $v_2 \in succ(v_1)$ and
  if dataflow fact d_1 holds at v_1 then d_2 holds at v_2
  (obtained from the graph representation of the transfer functions)
- **P** is the set of path edges (see slide 19)

IFDS constraint-based specification
Phase 1

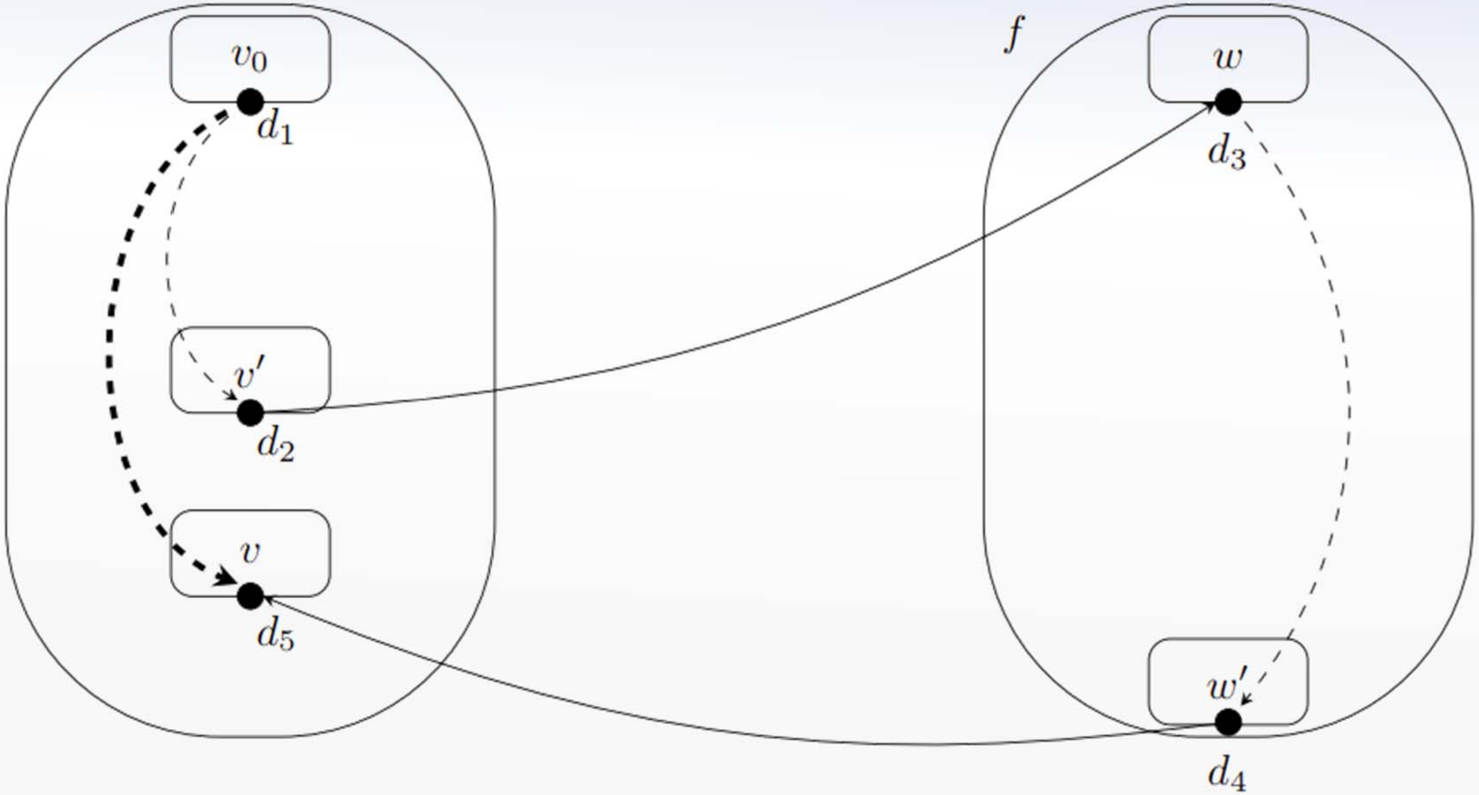
- v is a program entry node:

  $\langle v, \bullet \rangle \rightsquigarrow \langle v, \bullet \rangle \in \mathbf{P}$

- v is a function entry node, v_1 is a call node that calls the function containing v, and v_0 is the entry node of the function containing v_1:

  $\langle v_0, d_1 \rangle \rightsquigarrow \langle v_1, d_2 \rangle \in \mathbf{P} \wedge \langle v_1, d_2 \rangle \sim\!\!\rightarrow \langle v, d_3 \rangle \in \mathbf{E} \;\Rightarrow\; \langle v, d_3 \rangle \rightsquigarrow \langle v, d_3 \rangle \in \mathbf{P}$ for all d_1, d_2, d_3

IFDS constraint-based specification
Phase 1



- v is an after-call node belonging to a call node v', v_0 is the entry node of the function containing v and v', w is the entry node of the function being called, and w' is the exit node of that function:

$$\langle v_0, d_1\rangle \rightsquigarrow \langle v', d_2\rangle \in \mathbf{P} \wedge \langle v', d_2\rangle \leadsto \langle w, d_3\rangle \in \mathbf{E} \wedge \langle w, d_3\rangle \rightsquigarrow \langle w', d_4\rangle \in \mathbf{P} \wedge \langle w', d_4\rangle \leadsto \langle v, d_5\rangle \in \mathbf{E}$$
$$\Rightarrow \langle v_0, d_1\rangle \rightsquigarrow \langle v, d_5\rangle \in \mathbf{P} \quad \text{for all } d_1, d_2, d_3, d_4, d_5$$

IFDS constraint-based specification
Phase 1

- v is an after-call node belonging to a call node v'
  or v is another node with a predecessor v'∈pred(v)
  and v_0 is the entry node of the function containing v and v':
  $\langle v_0, d_1\rangle \rightsquigarrow\!\!\!\rightarrow \langle v', d_2\rangle \in \mathbf{P} \wedge \langle v', d_2\rangle \rightsquigarrow \langle v, d_3\rangle \in \mathbf{E} \Rightarrow \langle v_0, d_1\rangle \rightsquigarrow\!\!\!\rightarrow \langle v, d_3\rangle \in \mathbf{P}$ for all d_1, d_2, d_3

  Similar for any other node v with predecessor v' where
  v_0 is the entry node of the function containing v and v'

IFDS constraint-based specification
Phase 2

$$\langle v_0, d_1\rangle \rightsquigarrow \langle v, d_2\rangle \in \mathbf{P} \;\wedge\; d_2 \in D \;\Rightarrow\; d_2 \in [\![v]\!]$$

$[\![v]\!]$ now contains the set of dataflow facts that may hold at v

IFDS constraint-based specification

```
PathEdge(d1, m, d3) :-
    CFG(n, m),
    PathEdge(d1, n, d2),
    d3 <- eshIntra(n, d2).
PathEdge(d1, m, d3) :-
    CFG(n, m),
    PathEdge(d1, n, d2),
    SummaryEdge(n, d2, d3).
PathEdge(d3, start, d3) :-
    PathEdge(d1, call, d2),
    CallGraph(call, target),
    EshCallStart(call, d2, target, d3),
    StartNode(target, start).
SummaryEdge(call, d4, d5) :-
    CallGraph(call, target),
    StartNode(target, start),
    EndNode(target, end),
    EshCallStart(call, d4, target, d1),
    PathEdge(d1, end, d2),
    d5 <- eshEndReturn(target, d2, call).

EshCallStart(call, d, target, d2) :-
    PathEdge(_, call, d),
    CallGraph(call, target),
    d2 <- eshCallStart(call, d, target).

Result(n, d2) :-
    PathEdge(_, n, d2).
```

Figure 5. FLIX implementation of the IFDS analysis

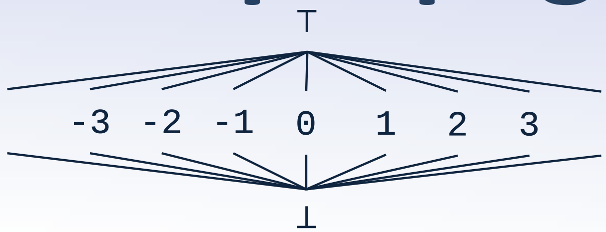
Agenda

- Distributive analysis
- IFDS
- **IDE**

IDE (Interprocedural Distributive Environment problems)

- *Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation,* Sagiv, Reps, Horwitz, TCS 1996
- Generalization of IFDS,
  in practice more efficient also for some IFDS problems!
- Setting:
  - lattice of abstract states: States = D → L where D is a finite set and L is a lattice (generalization of IFDS)
  - all transfer functions, f_v: States → States, are distributive (as with IFDS)
- Great idea #1:
  - also allows compact representation and summarization!
- Great idea #2:
  - the tabulation solver can easily be generalized…

Copy-constant propagation analysis



- Constant propagation analysis is not distributive
- ... but *copy-constant propagation analysis* is!
- Like constant propagation analysis, but only handles
  – constant assignments, e.g., x = 42
  – copy assignments, e.g., x = y
- All other assignments just give ⊤

- A variant: *linear-constant propagation analysis*
- Also handles linear expressions, e.g., x = 5*y+17

Exercise: prove that these two analyses are indeed distributive

A generalization of IFDS

- The powerset lattice $\mathcal{P}(D)$ is isomorphic to the map lattice $D \to \{T, F\}$ where $F \sqsubset T$ T="true", F="false"
- So $(\mathcal{P}(D) \to \mathcal{P}(D))^n$ is isomorphic to $((D \to \{T, F\}) \to (D \to \{T, F\}))^n$
- In IDE we have States = $D \to L$ where D is a finite set and L is a (finite-height) complete lattice
- IFDS thus corresponds to the special case $L = \{T, F\}$
- We have seen how to compactly represent distributive functions of the form $f: \mathcal{P}(D) \to \mathcal{P}(D)$
- How can we generalize that to distributive functions of the form $f: (D \to L) \to (D \to L)$ for arbitrary lattices**?**

Compact representation

- Assume f: (D → L) → (D → L) is distributive, D is a finite set, and L is a complete lattice
- Define g: (D ∪ {●}) × (D ∪ {●}) → (L → L) by

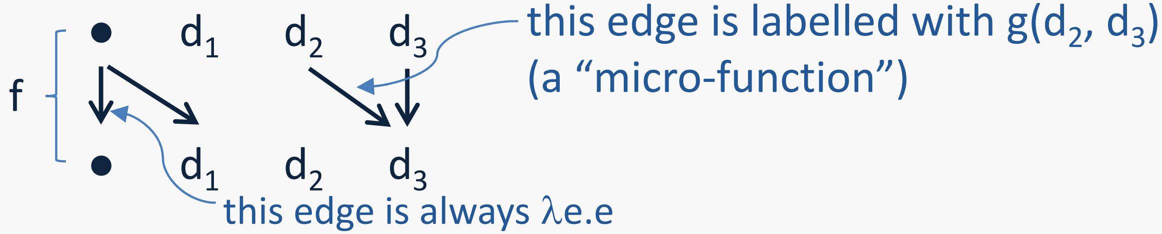
  g(a, b)(e) = f(⊥[a↦e])(b) for a,b∈D and e∈L

  g(●, b)(e) = f(⊥)(b) for b∈D and e∈L

  g(●, ●)(e) = e for e∈L

  g(a, ●)(e) = ⊥ for a∈D and e∈L

- Now $f(m)(b) = g(\bullet, b)\,(\bot) \sqcup \bigsqcup_{a\in D} g(a, b)(m(a))$
- Similar graph representation as in IFDS, but now each edge is a function L → L (an absent edge represents the function λe.⊥)



Compact representation

Exercise:

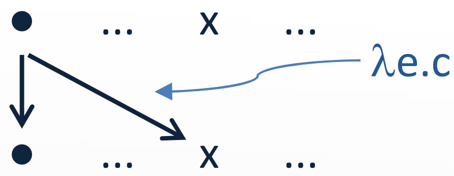
What is the graph representation of an assignment x=E for copy-constant propagation analysis?

Compact representation

Exercise:

What is the graph representation of an assignment x=E for copy-constant propagation analysis?

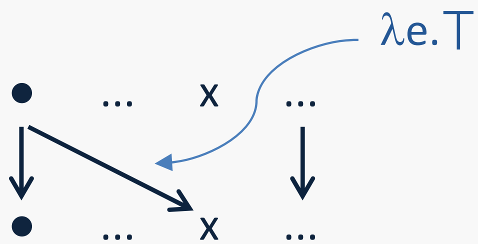
- If E is a constant c:



- If E is a variable y:

(default edge label: λe.e)

- Any other expression:



- How to also handle assignments like x = 5*y+1 ?
  (for linear-constant propagation analysis)

Composition and l.u.b.

- Function composition and least upper bound can be performed efficiently on the graph representation
  - here it is useful that $\bullet \rightsquigarrow \bullet$ is always labelled with $\lambda e.e$
- ...assuming efficiently representable lattice elements
  - for copy-constant propagation analysis we only need the identity function and constant functions, and those are trivially closed under composition and l.u.b.

Exercise: what about linear-constant propagation analysis?

Implementation: `TIP/src/tip/lattices/EdgeLattice`

Example [Sagiv et al., 1996]

```
declare x: integer
program main
begin
        call P(7)
        print (x) /* x is a constant here */
end

procedure P (value a : integer)
begin /* a is not a constant here */
        if a > 0 then
           a := a − 2
           call P (a)
           a := a + 2
        fi
        x := −2 ∗ a + 5
        /* x is not a constant here */
end
```

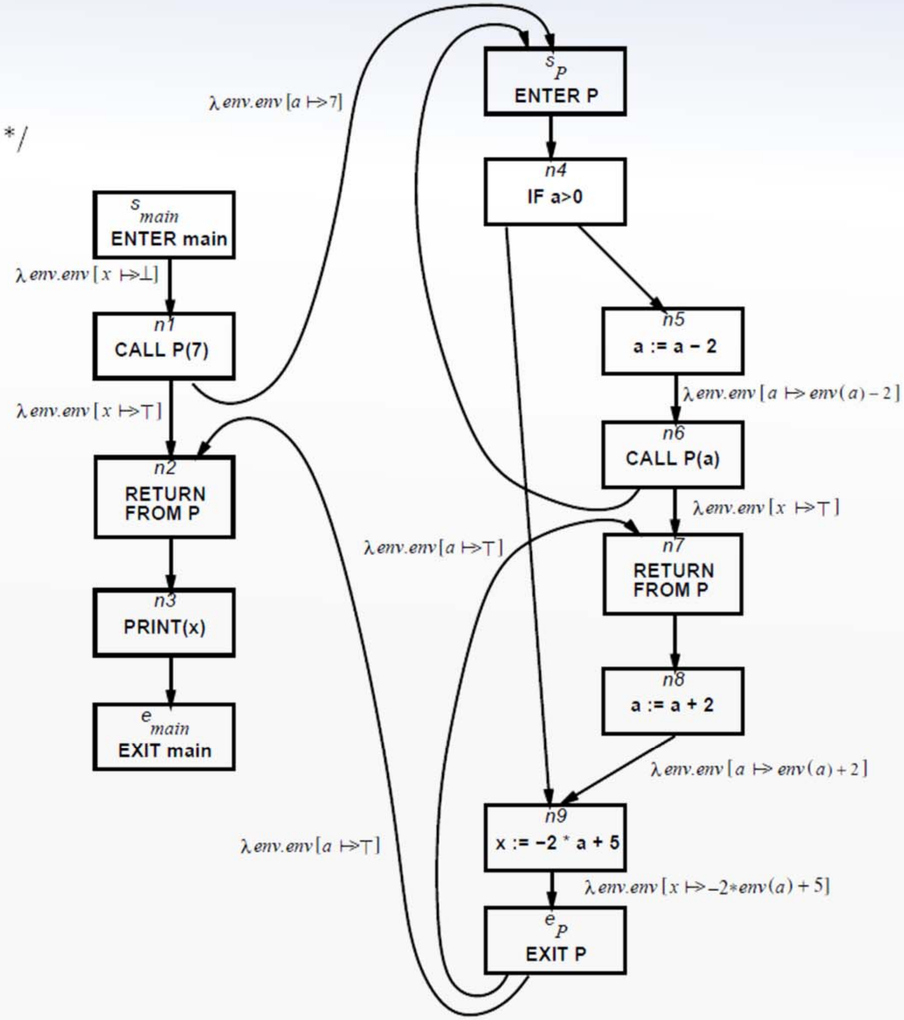


Figure 1: An example program and its labeled supergraph G^*. The environment transformer for all unlabeled edges is $\lambda env.env$.

(the paper uses lattices upside-down)

Example [Sagiv et al., 1996]

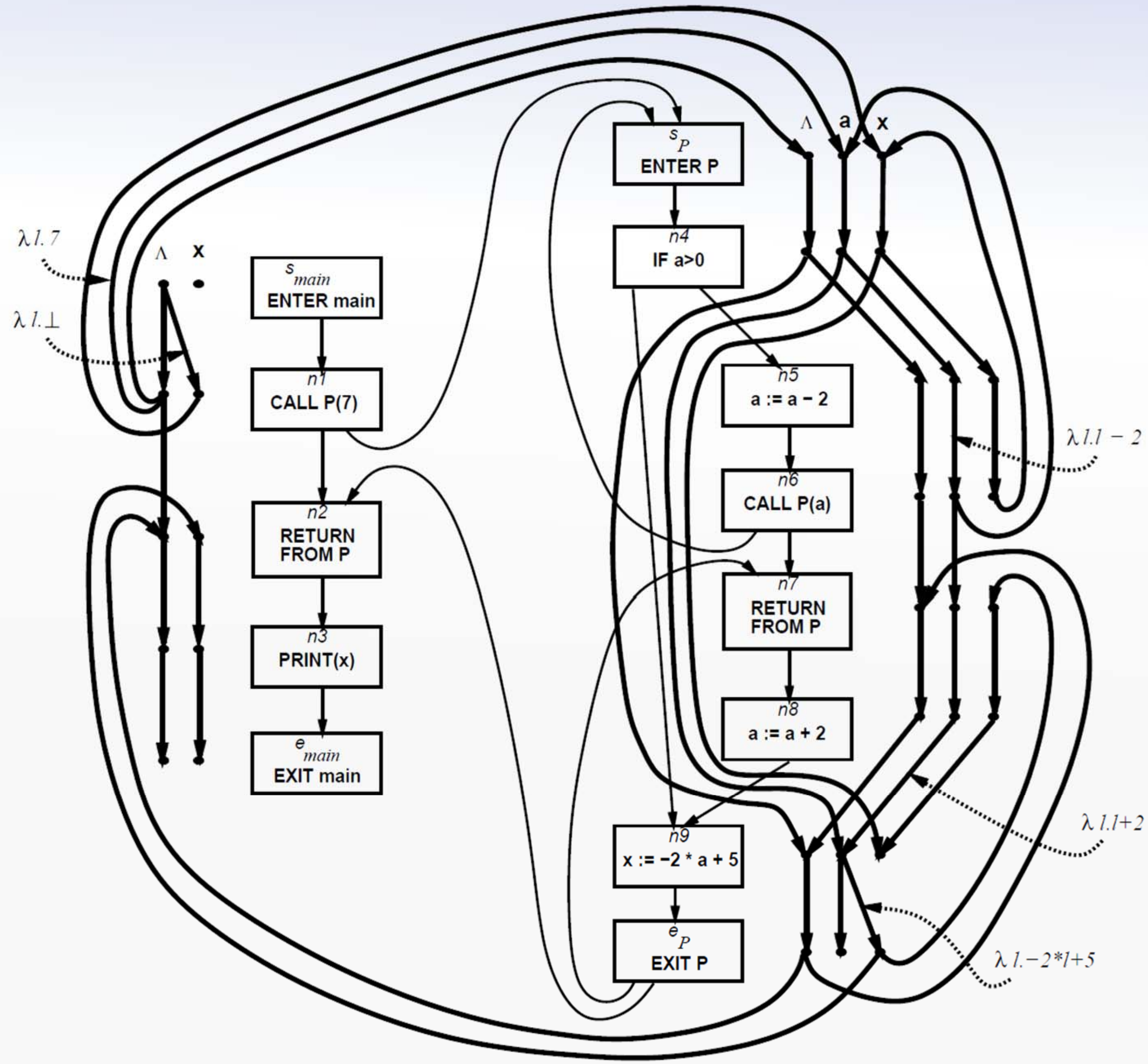


Figure 4: The labeled exploded supergraph for the running example program for the linear-constant-propagation problem. The edge functions are all $\lambda l.l$ except where indicated.

IDE constraint-based specification

- Edges in **E** and **P** are now labelled with L → L functions

- $[\![\langle v_1, d_1\rangle \rightsquigarrow \langle v_2, d_2\rangle]\!]_{\mathbf{P}} : L \to L$ denotes the label of the edge in **P** from $\langle v_1, d_1\rangle$ to $\langle v_2, d_2\rangle$

- $[\langle v_1, d_1\rangle \rightarrow \langle v_2, d_2\rangle]_{\mathbf{E}} : L \to L$ denotes the label of the edge in **E** from $\langle v_1, d_1\rangle$ to $\langle v_2, d_2\rangle$

IDE constraint-based specification
Phase 1

For the program entry:

$$id \sqsubseteq [\![\langle entry_{\mathtt{main}}, \bullet \rangle \rightsquigarrow \langle entry_{\mathtt{main}}, \bullet \rangle]\!]_{\mathbf{P}}$$

IDE constraint-based specification
Phase 1

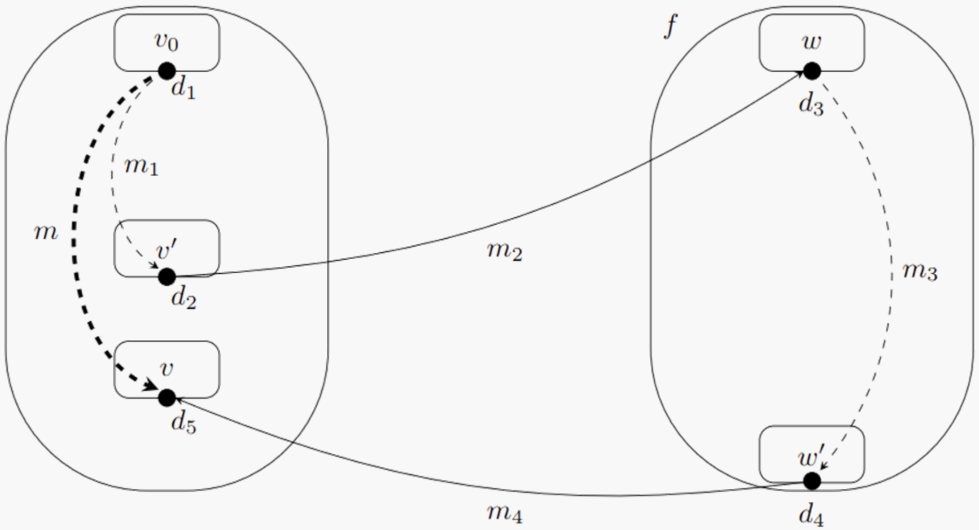
If v is a function entry node, v_1 is a call node that calls the function containing v, and v_0 is the entry node of the function containing v_1:

$$\forall d_1, d_2, d_3 : [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v_1, d_2\rangle]\!]_{\mathbf{P}} \neq \bot \;\wedge\; [\langle v_1, d_2\rangle \rightarrow \langle v, d_3\rangle]_{\mathbf{E}} \neq \bot$$
$$\Longrightarrow \; id \sqsubseteq [\![\langle v, d_3\rangle \rightsquigarrow \langle v, d_3\rangle]\!]_{\mathbf{P}}$$

IDE constraint-based specification
Phase 1

If v is an after-call node belonging to a call node v', v_0 is the entry node of the function containing v and v', w is the entry node of the function being called, and w' is the exit node of that function:

$$\forall d_1, d_2, d_3, d_4, d_5 :$$
$$m_1 = [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v', d_2\rangle]\!]_{\mathbf{P}} \neq \bot \;\wedge\; m_2 = [\langle v', d_2\rangle \rightarrow \langle w, d_3\rangle]_{\mathbf{E}} \neq \bot$$
$$\wedge\; m_3 = [\![\langle w, d_3\rangle \rightsquigarrow \langle w', d_4\rangle]\!]_{\mathbf{P}} \neq \bot \;\wedge\; m_4 = [\langle w', d_4\rangle \rightarrow \langle v, d_5\rangle]_{\mathbf{E}} \neq \bot$$
$$\Longrightarrow\; m_4 \circ m_3 \circ m_2 \circ m_1 \sqsubseteq [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v, d_5\rangle]\!]_{\mathbf{P}}$$



IDE constraint-based specification
Phase 1

If v is an after-call node belonging to a call node v’
or v is another node with a predecessor v’∈pred(v)
and v_0 is the entry node of the function containing v and v’:

$$\forall d_1, d_2, d_3 : m_1 = [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v', d_2\rangle]\!]_{\mathbf{P}} \neq \bot \;\wedge\; m_2 = [\langle v', d_2\rangle \rightarrow \langle v, d_3\rangle]_{\mathbf{E}} \neq \bot \\ \implies m_2 \circ m_1 \sqsubseteq [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v, d_3\rangle]\!]_{\mathbf{P}}$$

Similar for any other node v with predecessor v’ where
v_0 is the entry node of the function containing v and v’

IDE constraint-based specification
Phase 2

Computes abstract values: $[\![\langle v, d\rangle]\!] \in \mathit{lift}(L)$

Program entry: $\forall d\colon [\![\langle \mathit{entry}_{\mathtt{main}}, d\rangle]\!] \neq \text{unreachable}$

For any node v where v_0 is the entry of the function containing v:

$$\forall d_0, d\colon [\![\langle v_0, d_0\rangle]\!] \neq \text{unreachable} \;\wedge\; m = [\![\langle v_0, d_0\rangle \rightsquigarrow \langle v, d\rangle]\!]_{\mathbf{P}} \implies m([\![\langle v_0, d_0\rangle]\!]) \sqsubseteq [\![\langle v, d\rangle]\!]$$

If v is a function entry node and v_1 is a call node to v:

$$\forall d_1, d\colon [\![\langle v_1, d_1\rangle]\!] \neq \text{unreachable} \;\wedge\; m = [\langle v_1, d_1\rangle \rightarrow \langle v, d\rangle]_{\mathbf{E}} \implies m([\![\langle v_1, d_1\rangle]\!]) \sqsubseteq [\![\langle v, d\rangle]\!]$$

Combine into abstract states: $[\![v]\!]_2(d) = [\![\langle v, d\rangle]\!] \in L$ for $d \in D$

IDE constraint-based specification

```
JumpFn(d1, m, d3, comp(long, short)) :-
    CFG(n, m),
    JumpFn(d1, n, d2, long),
    (d3, short) <- eshIntra(n, d2).
JumpFn(d1, m, d3, comp(caller, summary)) :-
    CFG(n, m),
    JumpFn(d1, n, d2, caller),
    SummaryFn(n, d2, d3, summary).
JumpFn(d3, start, d3, identity()) :-
    JumpFn(d1, call, d2, _),
    CallGraph(call, target),
    EshCallStart(call, d2, target, d3, _),
    StartNode(target, start),
SummaryFn(call, d4, d5, comp(comp(cs, se), er)) :-
    CallGraph(call, target),
    StartNode(target, start),
    EndNode(target, end),
    EshCallStart(call, d4, target, d1, cs),
    JumpFn(d1, end, d2, se),
    (d5, er) <- eshEndReturn(target, d2, call).

EshCallStart(call, d, target, d2, cs) :-
    JumpFn(_, call, d, _),
    CallGraph(call, target),
    (d2, cs) <- eshCallStart(call, d, target).

InProc(p, start) :- StartNode(p, start).
InProc(p, m) :- InProc(p, n), CFG(n, m).

Result(n, d, apply(fn, vp)) :-
    ResultProc(proc, dp, vp),
    InProc(proc, n),
    JumpFn(dp, n, d, fn).

ResultProc(proc, dp, apply(cs, v)) :-
    Result(call, d, v),
    EshCallStart(call, d, proc, dp, cs).
```

Figure 6. FLIX implementation of the IDE analysis

Asymptotic running time

$$O(|E| \cdot |D|^3)$$

Same as IFDS!

[Sagiv et al., 1996]

Copy-constant propagation analysis with IDE

Implementation: `TIP/src/tip/analysis/CopyConstantPropagationAnalysis`

Copy-constant propagation – example

```
main() {
  var x,y;
  x = p(42);
  y = p(117);
  return x + y;
}

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

Context sensitive analysis with IDE concludes that `x` and `y` are constants at the exit of `main`

IFDS vs. IDE

- IDE is more general than IFDS
- ...and sometimes faster also for IFDS problems!

Example:

- Copy-constant propagation analysis fits into IFDS (the set of constants that appear as literals in the program is finite), but the set of dataflow facts is Vars $\times$ Literals (where Literals is the set of literals in the program)
- In contrast, IDE only needs one micro-function per CFG edge and program variable and a map Vars $\rightarrow$ Const for each CFG node (where Const is the constant propagation lattice)

Possibly-uninitialized variables analysis reformulated in IDE

- Lattice of abstract states: States = $\mathcal{P}$(Vars)
  which is isomorphic to: Vars $\to$ {T, F}
  ...and to: $\{\star\} \to \mathcal{P}$(Vars)
- The transfer function for assignments:

$$t_{x=E}(S) = \begin{cases} S \cup \{x\} & \text{if } vars(E) \cap S \neq \emptyset \\ S \setminus \{x\} & \text{otherwise} \end{cases}$$

- Exercise: How can such a transfer function be represented using micro-functions?
  - Hint: consider either of the two isomorphic lattice variants
- (Micro-functions for the other transfer functions are easy...)

Implementation: `TIP/src/tip/analysis/PossiblyUninitializedVarsAnalysis`

Demand-driven analysis

An alternative to exhaustive analysis

- IFDS: *“does dataflow fact d hold at program point v?”*
- IDE: *“what is the abstract value of x at program point v?”*

Use dynamic programming... [Reps et al., 1995], [Sagiv et al., 1996]

Implementations

- Soot: https://github.com/Sable/heros
- WALA: https://github.com/amaurremi/IDE
- TIP: https://github.com/cs-au-dk/TIP/blob/master/src/tip/solvers/IDESolver.scala

See also:

- Nomair A. Naeem, Ondrej Lhoták, Jonathan Rodriguez: *Practical Extensions to the IFDS Algorithm*. CC 2010
- Eric Bodden: *Inter-procedural Data-flow Analysis with IFDS/IDE and Soot.* SOAP@PLDI 2012
- Jonathan Rodriguez, Ondrej Lhoták: *Actor-Based Parallel Dataflow Analysis*. CC 2011
- Steven Arzt, Eric Bodden: *Reviser: Efficiently Updating IDE-/IFDS-based Data-Flow Analyses in Response to Incremental Program Changes*. ICSE 2014
- Magnus Madsen, Ming-Ho Yee, Ondrej Lhoták: *From Datalog to Flix: A Declarative Language for Fixed Points on Lattices*. PLDI 2016
- Johannes Späth, Karim Ali, Eric Bodden: *IDE^{al}: Efficient and Precise Alias-Aware Dataflow Analysis*. Proc. ACM Program. Lang. 1(OOPSLA): 99:1-99:27 (2017)