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:
  \[ \text{Sign} = + \quad - \quad 0 \]

- Lattice for abstract states:
  \[ \text{States} = \text{Vars} \rightarrow \text{Sign} \]

- 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: $\text{Contexts} \rightarrow \text{lift} (\text{Vars} \rightarrow \text{Sign})$

where $\text{Contexts} = \text{Vars} \rightarrow \text{Sign}$

\[
\begin{align*}
f(z) \{ & \\
\text{var t1, t2;} & \\
t1 &= z \ast 6; & \\
t2 &= t1 \ast 7; & \\
\text{return t2;} & \\
\}
\end{align*}
\]

The abstract state at the exit of $f$ can be used as a function summary

\[
\begin{align*}
[ & \perp [z \mapsto 0] \mapsto \perp [z \mapsto 0, t1 \mapsto 0, t2 \mapsto 0, \text{result} \mapsto 0], \\
& \perp [z \mapsto +] \mapsto \perp [z \mapsto +, t1 \mapsto +, t2 \mapsto +, \text{result} \mapsto +], \\
& \text{all other contexts} \mapsto \text{unreachable} ]
\end{align*}
\]

At this call, we can reuse the already computed exit abstract state of $f$ for the context $\perp [z \mapsto +]$
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 = \( \mathcal{P}(\text{Vars}) \)

• Analysis lattice: \( (\text{Contexts} \rightarrow \text{lift(States)})^n = \) \( (\mathcal{P}(\text{Vars}) \rightarrow \text{lift(\mathcal{P}(\text{Vars}))})^n \)
  – 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

```c
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, \( \text{var } x : \llbracket v \rrbracket = \text{JOIN}(v) \cup \{ x \} \)

– assignments, \( x = E : \)

\[
\llbracket v \rrbracket = t_v(\text{JOIN}(v))
\]

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

– function entries:

– after-call nodes:

– all others: \( \llbracket v \rrbracket = \text{JOIN}(v) \)

where \( \text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket \)
Possibly-uninitialized variables analysis

A forward, may analysis – context-sensitive version:

- variable declarations, \( \text{var} \ x : \ ? \)
- assignments, \( x = E \):
  \[
  t_v(S) = \begin{cases} 
  S \cup \{x\} & \text{if } \text{vars}(E) \cap S \neq \emptyset \\
  S \setminus \{x\} & \text{otherwise}
  \end{cases}
  \]
  \[
  \llbracket v \rrbracket(c) = \begin{cases} 
  t_v(JOIN(v,c)) & \text{if } JOIN(v,c) \in \text{States} \\
  \text{unreachable} & \text{if } JOIN(v,c) = \text{unreachable}
  \end{cases}
  \]
- program entry: \( \llbracket v \rrbracket(c) = \emptyset \)
- other function entries:
  - after-call nodes:
  - all others: \( \llbracket v \rrbracket(c) = \text{JOIN}(v,c) \)
  
  where \( \text{JOIN}(v,c) = \bigsqcup_{w \in \text{pred}(v)} \llbracket w \rrbracket(c) \)
Pre-analysis

• The analysis lattice is \((\text{lift}(\mathcal{P}(\text{Vars}) \rightarrow \mathcal{P}(\text{Vars})))^n\)

• *Idea:* run a *context-insensitive*(!) analysis that computes, for each CFG node \(v\), a map \(m_v: \mathcal{P}(\text{Vars}) \rightarrow \mathcal{P}(\text{Vars})\) with the following property:

  If the function containing \(v\) is executed in an initial abstract state where \(S \subseteq \text{Vars}\) 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 $f : L_1 \rightarrow L_2$ 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 $x=\ast y$ or $\ast x=y$.)
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 = \( P(D) \) where D is a finite set (i.e., a powerset lattice)
  - all transfer functions, \( f_v : \text{States} \to \text{States} \), 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 \( f: P(D) \rightarrow P(D) \) 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 \( g: (D \cup \{ ● \}) \rightarrow P(D) \)
  
  - Define \( g(●) = f(∅) \) and \( g(d)=f(\{d\}) \) for \( d \in D \)
  
  - Now \( f(X) = g(●) \cup \bigcup_{y \in X} g(y) \)

• Can be represented compactly as a graph with \( 2(|D|+1) \) nodes
  
  - Example: \( \bullet \) for \( D=\{d_1, d_2, d_3\} \)

\[
\begin{array}{c}
\bullet \\
\quad \downarrow \\
\bullet & d_1 & d_2 & d_3 \\
\quad \downarrow \\
\bullet & d_1 & d_2 & d_3
\end{array}
\]

means that \( g(●) = f(∅), g(d_1)=f(∅), g(d_2)=∅, \) and \( g(d_3)=f(\{d_2,d_3\}) \)

(the edge from \( ● \) to \( ● \) is always present)

so \( f(S) = (S \cup \{d_1\}) \setminus \{d_2\} \cup P \) where
\[
P = \begin{cases} 
\{d_3\} & \text{if } d_2 \in S \\
\emptyset & \text{otherwise}
\end{cases}
\]

- In general, the edges are:
  \[
\{●\rightarrow●\} \cup \{●\rightarrow y \mid y \in f(∅)\} \cup \{x\rightarrow y \mid y \in f(\{x\}) \land y \notin f(∅)\}
\]
Exercise:
For uninitialized-variables analysis, what is the IFDS graph representation of
1) an assignment, \( X = E \), or
2) a variable declaration, \( \text{var} \ X \) ?
Distributivity is closed under function composition and l.u.b. Assume $f_A : \mathcal{P}(D) \to \mathcal{P}(D)$ and $f_B : \mathcal{P}(D) \to \mathcal{P}(D)$ where $D$ is a finite set and both $f$ and $are distributive

- $f_A \circ f_B : \mathcal{P}(D) \to \mathcal{P}(D)$ is also distributive $(f_A \circ f_B)(S) = f_A(f_B(S))$
- $f_A \sqcup f_B : \mathcal{P}(D) \to \mathcal{P}(D)$ is also distributive $(f_A \sqcup f_B)(S) = f_A(S) \sqcup f_B(S)$

Proof? (exercise)

With the graph representation:
Possibly-uninitialized variables analysis

- The analysis lattice is \((\text{lift}(\mathcal{P}(\text{Vars}) \to \mathcal{P}(\text{Vars})))^n\)

- For each reachable CFG node, the analysis computes an element of \(\mathcal{P}(\text{Vars}) \to \mathcal{P}(\text{Vars})\)

- 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: \textit{the tabulation algorithm}
The IFDS Tabulation Algorithm

• The idea: with a worklist algorithm, incrementally build a set of path edges \( \langle v_1, d_1 \rangle \rightarrow \langle v_2, d_2 \rangle \) 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!
Dataflow at function calls

\[ \text{result} = f(b_1, ..., b_n) \]

function parameter values

values of local variables

return values

values of local variables
IFDS constraint-based specification
Phase 1

• $E$ represents the program being analyzed:
  $\langle v_1,d_1 \rangle \leadsto \langle v_2,d_2 \rangle \in E$ means that $v_2 \in \text{succ}(v_1)$ 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 \mapsto \langle v, \bullet \rangle \in 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 \mapsto \langle v_1, d_2 \rangle \in P \land \langle v_1, d_2 \rangle \leadsto \langle v, d_3 \rangle \in E \Rightarrow \langle v, d_3 \rangle \mapsto \langle v, d_3 \rangle \in P \quad \text{for all } d_1, d_2, d_3

• 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 \mapsto \langle v', d_2 \rangle \in P \land \langle v', d_2 \rangle \leadsto \langle w, d_3 \rangle \in E \land \langle w, d_3 \rangle \mapsto \langle w', d_4 \rangle \in P \land \langle w', d_4 \rangle \leadsto \langle v, d_5 \rangle \in E \Rightarrow \langle v_0, d_1 \rangle \mapsto \langle v, d_5 \rangle \in P \quad \text{for all } d_1, d_2, d_3, d_4, d_5

• v is an after-call node belonging to a call node \( v' \) or v is another node with a predecessor \( v' \in \text{pred}(v) \) and \( v_0 \) is the entry node of the function containing \( v \) and \( v' \):
  \langle v_0, d_1 \rangle \mapsto \langle v', d_2 \rangle \in P \land \langle v', d_2 \rangle \leadsto \langle v, d_3 \rangle \in E \Rightarrow \langle v_0, d_1 \rangle \mapsto \langle v, d_3 \rangle \in P \quad \text{for all } d_1, d_2, d_3
IFDS constraint-based specification

Phase 2

$$\langle v_0, d_1 \rangle \mapsto \langle v, d_2 \rangle \in P \land d_2 \in D \Rightarrow d_2 \in [v]$$

$$[v]$$ 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
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: $\text{States} = D \rightarrow L$ where $D$ is a finite set and $L$ is a lattice (generalization of IFDS)
  - all transfer functions, $f_v: \text{States} \rightarrow \text{States}$, 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 \( \top \)

• 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 $D \rightarrow \{T, F\}$ where $F \sqsubseteq T$. $T$=“true”, $F$=“false”
- So $(\mathcal{P}(D) \rightarrow \mathcal{P}(D))^n$ is isomorphic to $((D \rightarrow \{T, F\}) \rightarrow (D \rightarrow \{T, F\}))^n$
- In IDE we have States = $D \rightarrow L$ 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 $f$: $\mathcal{P}(D) \rightarrow \mathcal{P}(D)$
- How can we generalize that to distributive functions of the form $f$: $(D \rightarrow L) \rightarrow (D \rightarrow L)$ for arbitrary lattices?
Compact representation

• Assume \( f: (D \rightarrow L) \rightarrow (D \rightarrow L) \) is distributive, \( D \) is a finite set, and \( L \) is a complete lattice

• Define \( g: (D \cup \{\bullet\}) \times (D \cup \{\bullet\}) \rightarrow (L \rightarrow L) \) by

\[
g(a, b)(e) = f(\bot[a \mapsto e])(b) \text{ for } a, b \in D \text{ and } e \in L
\]

\[
g(\bullet, b)(e) = f(\bot)(b) \text{ for } b \in D \text{ and } e \in L
\]

\[
g(\bullet, \bullet)(e) = e \text{ for } e \in L
\]

\[
g(a, \bullet)(e) = \bot \text{ for } a \in D \text{ and } e \in L
\]

• 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 \( L \rightarrow L \) (an absent edge represents the function \( \lambda e.\bot \))
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 \( c \):
  - Diagram:

- If \( E \) is a variable \( y \):
  - Diagram:
  (default edge label: \( \lambda.e.e \))

- Any other expression:
  - Diagram:

- 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 $\bullet \sim \bullet$ 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]

```plaintext
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)
Figure 4: The labeled exploded supergraph for the running example program for the linear-constant-propagation problem. The edge functions are all \( \lambda I.I \) except where indicated.
IDE constraint-based specification

• Path edges are now labelled with $L \rightarrow L$ functions

• $[[\langle v_1, d_1 \rangle \rightsquigarrow \langle v_2, d_2 \rangle]]_P : L \rightarrow L$ denotes the label of the path edge from $\langle v_1, d_1 \rangle$ to $\langle v_2, d_2 \rangle$. 
IDE constraint-based specification
Phase 1

For the program entry:

\[ id \sqsubseteq \left[ \langle \text{entry}_{\text{main}}, \bullet \rangle \rightsquigarrow \langle \text{entry}_{\text{main}}, \bullet \rangle \right]_P \]
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 \( v_1 \):

\[
\forall d_1, d_2, d_3: [[v_0, d_1] \leadsto [v_1, d_2]]_P \land [v_1, d_2] \xrightarrow{m_2} [v, d_3] \in E
\implies id \subseteq [[v, d_3] \leadsto [v, d_3]]_P
\]
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:

\[
\forall d_1, d_2, d_3, d_4, d_5 : m_1 = [\langle v_0, d_1 \rangle \leadsto \langle v', d_2 \rangle]_P \land \langle v', d_2 \rangle \xrightarrow{m_2} \langle w, d_3 \rangle \in E \\
\land m_3 = [\langle w, d_3 \rangle \leadsto \langle w', d_4 \rangle]_P \land \langle w', d_4 \rangle \xrightarrow{m_4} \langle v, d_5 \rangle \in E \\
\Rightarrow m_4 \circ m_3 \circ m_2 \circ m_1 \sqsubseteq [\langle v_0, d_1 \rangle \leadsto \langle v, d_5 \rangle]_P
\]
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 $v' \in \text{pred}(v)$ and $v_0$ is the entry node of the function containing $v$ and $v'$:

$$\forall d_1, d_2, d_3: m_1 = [\langle v_0, d_1 \rangle \leadsto \langle v', d_2 \rangle]_P \land \langle v', d_2 \rangle \xrightarrow{m_2} \langle v, d_3 \rangle \in E$$

$$\implies m_2 \circ m_1 \sqsubseteq [\langle v_0, d_1 \rangle \leadsto \langle v, d_3 \rangle]_P$$
Computes abstract values: \([\langle v, d \rangle] \in \text{lift}(L)\)

Program entry: \(\forall d: [\langle entry_{\text{main}}, d \rangle] \neq \text{unreachable}\)

For any node \(v\) where \(v_0\) is the entry of the function containing \(v\):
\[
\forall d_0, d: [\langle v_0, d_0 \rangle] \neq \text{unreachable} \land m = \left[\langle v_0, d_0 \rangle \leadsto \langle v, d \rangle\right]_P \implies m([\langle v_0, d_0 \rangle]) \subseteq [\langle v, d \rangle]
\]

If \(v\) is a function entry node and \(v_1\) is a call node to \(v\):
\[
\forall d_1, d: [\langle v_1, d_1 \rangle] \neq \text{unreachable} \land \langle v_1, d_1 \rangle \overset{m}{\rightarrow} \langle v, d \rangle \in E \implies m([\langle v_1, d_1 \rangle]) \subseteq [\langle v, d \rangle]
\]

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. FLLIX 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

```javascript
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 $\text{Vars} \times \text{Literals}$ (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 $\text{Vars} \rightarrow \text{Const}$ for each CFG node (where Const is the constant propagation lattice)
Possibly-uninitialized variables analysis reformulated in IDE

- Lattice of abstract states: \( \text{States} = \mathcal{P}(\text{Vars}) \)
  which is isomorphic to: \( \text{Vars} \rightarrow \{\text{T}, \text{F}\} \)
  ...and to: \( \{\star\} \rightarrow \mathcal{P}(\text{Vars}) \)

- The transfer function for assignments:
  \[
  t_{x=E}(S) = \begin{cases} 
  S \cup \{x\} & \text{if } \text{vars}(E) \cap S \neq \emptyset \\ 
  S \setminus \{x\} & \text{otherwise} 
  \end{cases}
  \]

- 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
- Magnus Madsen, Ming-Ho Yee, Ondrej Lhoták: *From Datalog to Flix: A Declarative Language for Fixed Points on Lattices*. PLDI 2016