Static Program Analysis
Part 8 – distributive analysis frameworks

http://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

⇓

efficient analysis algorithms
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:  Contexts → lift(Vars → Sign)
where Contexts = Vars → Sign

\[ f(z) \{ \]
  \[ \text{var } t1, t2; \]
  \[ t1 = z \times 6; \]
  \[ t2 = t1 \times 7; \]
  \[ \text{return } t2; \]
\[ \} \]
\[ \ldots \]
\[ x = f(0); \]
\[ y = f(87); \]
\[ z = f(42); \]
\[ \ldots \]

The abstract state at the exit of \( f \) can be used as a function summary

\[ [ \bot[z \mapsto 0] \mapsto \bot[z \mapsto 0, t1 \mapsto 0, t2 \mapsto 0, \text{result} \mapsto 0], \]
\[ \bot[z \mapsto +] \mapsto \bot[z \mapsto +, t1 \mapsto +, t2 \mapsto +, \text{result} \mapsto +], \]
\[ \text{all other contexts } \mapsto \text{unreachable} ] \]

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

• Analysis lattice:  \((\text{Contexts} \rightarrow \text{States})^n = (\mathcal{P}(\text{Vars}) \rightarrow \mathcal{P}(\text{Vars}))^n\)
  – as usual, \( n \) is the number of CFG nodes
  – recall that the full functional approach has \( \text{Contexts} = \text{States} \)
  – intuitively, the context is the set of possibly uninitialized variables at the entry of the current function
  – (we don’t use a lifted lattice for reachability here)

*) In this analysis, a variable is possibly-uninitialized if its value may come from an uninitialized variable.
Possibly-uninitialized variables – example

```plaintext
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:

- $\bigcup = \bigcup$
- program entry, $v = \text{entry}$: $\llbracket v \rrbracket = \text{Vars}$
- assignments, $v = x = E$: $\llbracket v \rrbracket = \begin{cases} \text{JOIN}(v) \cup \{x\} & \text{if } \text{vars}(E) \cap \text{JOIN}(v) \neq \emptyset \\ \text{JOIN}(v) \setminus \{x\} & \text{otherwise} \end{cases}$
- other function entries:
- after-call nodes:
- all others: $\llbracket v \rrbracket = \text{JOIN}(v)$

The transfer function for assignments is $t_{x=E}(S) = \begin{cases} S \cup \{x\} & \text{if } \text{vars}(E) \cap S \neq \emptyset \\ S \setminus \{x\} & \text{otherwise} \end{cases}$ with $\llbracket v \rrbracket = t_v(\text{JOIN}(v))$
Possibly-uninitialized variables analysis

A forward, may analysis – context-sensitive version:

- \( \mathbb{U} = \mathbb{U} \)
- program entry, \( v = \text{entry} \): \( \llbracket v \rrbracket(\emptyset) = \text{Vars} \)
- assignments, \( v = x = E \): \( \llbracket v \rrbracket(c) = \begin{cases} \text{JOIN}(v,c) \cup \{x\} & \text{if } \text{vars}(E) \cap \text{JOIN}(v,c) \neq \emptyset \\ \text{JOIN}(v,c) \setminus \{x\} & \text{otherwise} \end{cases} \)
- other function entries:
- after-call nodes:
- all others:

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

where \( \text{JOIN}(v,c) = \bigsqcup_{w \in \text{pred}(v)} \llbracket w \rrbracket(c) \)

The transfer function for assignments is

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

with \( \llbracket v \rrbracket(c) = t_{v}(\text{JOIN}(v,c)) \)

\[ w_1 \ldots w_n \]

\[ v \rightarrow t_v \]
Pre-analysis

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

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

- If we have such a pre-analysis, then we can easily compute the sets of maybe-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
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?
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: $\text{States} = \mathcal{P}(D)$ where $D$ is a finite set (i.e., a powerset lattice)
  - all transfer functions, $f_v: \text{States} \rightarrow \text{States}$, 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 (pre-analysis + main analysis)
  - can be made demand driven
**Compact representation**

- Assume $f: \mathcal{P}(D) \rightarrow \mathcal{P}(D)$ 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 $g: (D \cup \{\bullet\}) \rightarrow \mathcal{P}(D)$
  - Define $g(\bullet) = f(\emptyset)$ and $g(d) = f\{d\}$ 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
  - Example: for $D=\{d_1, d_2, d_3\}$
    
    \[
    \begin{align*}
    &\text{Example:} \quad \bullet \quad d_1 \quad d_2 \quad d_3 \\
    &\downarrow \quad \downarrow \quad \downarrow \\
    &\bullet \quad d_1 \quad d_2 \quad d_3
    \end{align*}
    \]
    
    means that $g(\bullet) = f(\emptyset)$, $g(d_1) = f(\emptyset)$, $g(d_2) = \emptyset$, and $g(d_3) = f\{d_2, d_3\}$
    (the edge from $\bullet$ to $\bullet$ 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:
      \[
      \{\bullet \leadsto \bullet\} \cup \{\bullet \leadsto y \mid y \in f(\emptyset)\} \cup \{x \leadsto y \mid y \in f\{x\} \land y \notin f(\emptyset)\}
      \]
Exercise:
For uninitialized-variables analysis, what is the IFDS graph representation of
1) an assignment, $X = E$, or
2) a variable declaration, $\text{var } X$?
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:
Possibly-uninitialized variables analysis

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

• For each CFG node \(v\), the analysis computes an element of \(\mathcal{P}(\text{Vars}) \rightarrow \mathcal{P}(\text{Vars})\)

• 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: **the tabulation algorithm**
The IFDS Tabulation Algorithm

- The idea: with a worklist algorithm, incrementally build a set of path edges \((v, d_1, d_2)\) where
  - \(v\) is a CFG node, \(d_1, d_2 \in D \cup \{\bullet\}\)
  - the edge means: if dataflow fact \(d_1\) holds at the entry of the function containing \(v\) then \(d_2\) holds at \(v\)
- 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]

```plaintext
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 $G^R_{TP}$ that are reachable along realizable paths from $(v_{main}, 0)$. Open circles represent nodes not reachable along such paths.
Dataflow at function calls

\[ \square = f(E_1, \ldots, E_n) \]

\[ X = \square \]

\[ \text{function parameter values} \]

\[ \text{function } f(b_1, \ldots, b_n) \]

\[ \text{values of local variables} \]

\[ \text{return values} \]

\[ \text{result} = E \]
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
IFDS constraint-based specification
Phase 1

• $E$ represents the program being analyzed:
  $(v_1, d_1, v_2, d_2) \in E$ 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)

• $\text{PathEdge}$ is the set of path edges (see slide 19)

• $\text{SummaryEdge}$ is a set of summary edges:
  $(v, d_1, d_2) \in \text{SummaryEdge}$ means that $v$ is a call node and if $d_1$ holds at $v$ then $d_2$ holds at $v$’s after-call node
  (used for bookkeeping during phase 1)
IFDS constraint-based specification

Phase 1

• v is a program entry node:
  \((v, \bullet, \bullet) \in \text{PathEdge}\)

• v is a call node:
  \((v, d_1, d_2) \in \text{PathEdge} \land (v, d_2, v_3, d_3) \in E \land v_3 \in \text{callee}(v) \Rightarrow (v_3, d_3, d_3) \in \text{PathEdge}\)
  \((v, d_1, d_2) \in \text{PathEdge} \land (v, d_2, v_3, d_3) \in E \land v_3 = \text{aftercall}(v) \Rightarrow (v_3, d_1, d_3) \in \text{PathEdge}\)
  \((v, d_1, d_2) \in \text{PathEdge} \land (v, d_2, d_3) \in \text{SummaryEdge} E \land v_3 = \text{aftercall}(v) \Rightarrow (v_3, d_1, d_3) \in \text{PathEdge}\)

• v is a function exit node:
  \((v, d_1, d_2) \in \text{PathEdge} \land (v_0, d_0, v_1, d_1) \in E \land (v, d_2, v_3, d_3) \in E \land v_1 = \text{funentry}(v) \land v_1 \in \text{callee}(v_0) \land v_3 = \text{aftercall}(v_0) \Rightarrow (v_0, d_0, d_3) \in \text{SummaryEdge}\)
  \((v, d_1, d_2) \in \text{PathEdge} \land (v_0, d_0, v_1, d_1) \in E \land (v, d_2, v_3, d_3) \in E \land (v_0, d_4, d_0) \in \text{PathEdge} \land v_1 = \text{funentry}(v) \land v_1 \in \text{callee}(v_0) \land v_3 = \text{aftercall}(v_0) \Rightarrow (v_3, d_4, d_3) \in \text{PathEdge}\)

• v is another node:
  \((v, d_1, d_2) \in \text{PathEdge} \land v_3 \in \text{succ}(v) \land (v, d_2, v_3, d_3) \in E \Rightarrow (v_3, d_1, d_3) \in \text{PathEdge}\)
IFDS constraint-based specification

Phase 2

\[(v, d_1, d_2) \in \text{PathEdge} \Rightarrow d_2 \in \llbracket v \rrbracket\]

\llbracket v \rrbracket \text{ now contains the set of dataflow facts that may hold at } v
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, \( f_v : \text{States} \rightarrow \text{States} \), 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 $D \to \{T, F\}$ where $F \subseteq T$  
  \[ T=\text{“true”, } F=\text{“false”}\]

• 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 $f: (D \rightarrow L) \rightarrow (D \rightarrow L)$ 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
  
  $g(a, b)(e) = f(\bot_{[a\mapsto e]})(b)$ for $a,b \in D$ and $e \in L$
  
  $g(\bullet, b)(e) = f(\bot)(b)$ for $b \in D$ and $e \in L$
  
  $g(\bullet, \bullet)(e) = e$ for $b \in D$ and $e \in L$
  
  $g(a, \bullet)(e) = \bot$ for $a \in D$ 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 $\lambda e. \bot$)

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

\text{this edge is labelled with } g(d_2, d_3) \text{ (a “micro-function”)}

\text{this edge is always } \lambda e. e
Exercise:
What is the graph representation of an assignment $x=E$ for copy-constant propagation analysis?
Exercise:
What is the graph representation of an assignment $x=E$ for copy-constant propagation analysis?

- If $E$ is a constant $c$:
  \[
  \begin{array}{c}
  \bullet \\
  \downarrow \\
  \bullet \\
  \end{array}
  \quad \begin{array}{c}
  \ldots \\
  \downarrow \\
  x \\
  \downarrow \\
  \ldots \\
  \end{array}
  \quad \lambda_e.c
  \]

- If $E$ is a variable $y$:
  \[
  \begin{array}{c}
  \bullet \\
  \downarrow \\
  \bullet \\
  \end{array}
  \quad \begin{array}{c}
  \ldots \\
  \downarrow \\
  x \\
  \downarrow \\
  y
  \end{array}
  \quad \lambda_e.T
  \]
  (default edge label: $\lambda_e.e$)

- Any other expression:
  \[
  \begin{array}{c}
  \bullet \\
  \downarrow \\
  \bullet \\
  \end{array}
  \quad \begin{array}{c}
  \ldots \\
  \downarrow \\
  x \\
  \downarrow \\
  \ldots \\
  \end{array}
  \quad \lambda_e.T
  \]

- 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]

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 $\lambda l.l$ except where indicated.
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
The IDE Tabulation Solver

Phase 1: (the pre-analysis)

With a worklist algorithm, incrementally build a table of *jump functions*, \( jf(v, d_1, d_2) : L \to L \) where

- \( v \) is a CFG node, \( d_1, d_2 \in D \cup \{\bullet\} \)
- \( jf(v, d_1, d_2) \) expresses **how the abstract value of** \( d_2 \) **at** \( v \) **is affected by the abstract value of** \( d_1 \) **at the entry of the function containing** \( v \)

Phase 2: (the main analysis)

Compute the abstract state for any given CFG node by exploiting the jump functions

Implementation: TIP/src/tip/solvers/IDESolver
The IDE Tabulation Solver, Phase 1

initialize all jump functions \( jf \) and summary functions \( sf \) to bottom
initialize worklist = \{ initial \}, set \( jf_{\text{initial}} = \lambda e.e \) where initial = \((v_{\text{entry}}, \bullet, \bullet)\)

while worklist ≠ ∅ {
    select and remove an item \((v, d_1, d_2)\) from worklist
    switch \( v \) {
        case call node: ...
        case exit node: ...
        case other node: ...
    }
}

at the end, we have computed function summaries for all reachable functions in the program

iteratively explores all reachable functions

an item \((v, d_1, d_2)\) is in the worklist if
\(d_2\) at \(v\) may be affected by \(d_1\) at the function entry
and the successors of such execution paths may not yet be accounted for in the jump functions

propagate\((item, jf)\) {
    let \( jf' = jf_{\text{item}} \sqcup jf \)
    if \( jf' \neq jf \) {
        \( jf_{\text{item}} = jf' \)
        add item to worklist
    }
}
initialize all jump functions $jf$ and summary functions $sf$ to bottom
initialize worklist = \{ initial \}, set $jf_{initial} = \lambda e.e$ where initial = \( (v_{entry}, \bullet, \bullet) \)
while worklist \( \neq \emptyset \) {
    select and remove an item \((v, d_1, d_2)\) from worklist
    switch \( v \) {
        case call node: ...
        case exit node: ...
        case other node:
            for each edge from \((v, d_2)\) to \((v_3, d_3)\) with micro-function \( m \) for some \( v_3, d_3 \) {
                propagate( \((v_3, d_1, d_3)\), m \circ jf(v, d_1, d_2) \)
            }
    }
}

(The algorithm is formulated slightly differently in [Sagiv et al., 1996])
The IDE Tabulation Solver, Phase 1

initialize all jump functions $jf$ and summary functions $sf$ to bottom
initialize worklist = { initial }, set $jf_{initial} = \lambda.e.e$ where initial = $(v_{entry}, \bullet, \bullet)$
while worklist ≠ ∅ {
    select and remove an item $(v,d_1,d_2)$ from worklist
    switch v {
        case call node:
            let $v_3$ be the entry node of the function being called at $v$
            let $v_4$ be the exit node of the function being called at $v$
            let $v_5$ be the after-call node of $v$
            // trigger analysis of the function being called
            for each edge from $(v,d_2)$ to $(v_3,d_3)$ for some $d_3$
                propagate($v_3,d_3,d_3$, $\lambda.e.e$)
        }
        case exit node: ...
        case other node: ...
    }
    // use the existing call summary
    for each $d_5$ where $sf(v,d_2,d_5)$ is non-bottom {
        propagate($v_5,d_1,d_5$, $sf(v,d_2,d_5) \circ jf(v,d_1,d_2)$)
    }
}

// model flow of, for example, local variables at the call
for each edge from $(v,d_2)$ to $(v_5,d_5)$ with micro-function $m$ for some $d_5$
    propagate($v_5,d_1,d_5$, $m \circ jf(v,d_1,d_2)$)

// trigger analysis of the function being called
initialize all jump functions $jf_\_\_$ and summary functions $sf_\_\_$ to bottom
initialize worklist = \{ initial \}, set $jf_{initial} = \lambda e.e$ where initial = (v_{entry}, ●, ●)
while worklist ≠ ∅ {
    select and remove an item $(v,d_1,d_2)$ from worklist
    switch v {
        case call node: ...
        case exit node: ...
        case other node: ...
    }
    for each call node $v_0$ that calls the function containing $v$ {
        let $v_1$ be the entry node of the function containing $v$
        let $v_5$ be the after-call node of $v_0$
        for each edge from $(v_0,d_0)$ to $(v_1,d_1)$ with micro-function $m_1$ for some $d_0$ {
            for each edge from $(v,d_2)$ to $(v_5,d_5)$ with micro-function $m_5$ for some $d_5$ {
                // build call summary from the call node to the after-call node
                let $sf' = m_5 \circ jf(v,d_1,d_2) \circ m_1 \sqcup sf(v_0,d_0,d_5)$
                if $sf' \neq sf(v_0,d_0,d_5)$ {
                    $sf(v_0,d_0,d_5) = sf'$
                    // propagate to after-call
                    for each $d_6$ where $jf(v_0,d_6,d_0)$ is non-bottom {
                        propagate( $(v_5,d_6,d_5), sf' \circ jf(v_0,d_6,d_0)$ )
                    }
                }
            }
        }
    }
}
The IDE Tabulation Solver, Phase 2

- Computes an element of \((D \rightarrow L)^n\) represented by \(x_{v,d} \in L\) for each CFG node \(v\) and each \(d \in D\)
- Information for different contexts is conflated, but with the precision of fully context-sensitive analysis!

```
initialize \(x_{v,d} = \bot\) for each CFG node \(v\) and each \(d \in D\)

// phase 2a: compute information for function entry nodes and call nodes
initialize worklist = \{ initial \} and set \(x_{initial} = \top\) where initial = \((v_{entry}, \bullet)\)
while worklist \(\neq \emptyset\) {
    select and remove an item \((v,d)\) from worklist
    switch \(v\) {
        case entry node: ...
        case call node: ...
    }
}

// phase 2b: compute information for other CFG nodes
for each node \(v\) that is not a function entry node or a call node {
    for each \(d_1,d_2\) where \(j_f(v,d_1,d_2)\) is non-bottom {
        \(x_{v,d} = x_{v,d} \sqcup j_f(v,d_1,d_2)(x_{v_1,d_1})\) where \(v_1\) is the function entry of \(v\)
    }
}

propagate\((v, d, a)\) {
    let \(a' = x_{v,d} \sqcup a\)
    if \(a' \neq a\) {
        \(x_{v,d} = a'\)
        add \((v,d)\) to worklist
    }
}
```
The IDE Tabulation Solver, Phase 2

- Computes an element of \((D \rightarrow L)^n\) represented by \(x_{v,d} \in L\) for each CFG node \(v\) and each \(d \in D\)
- Information for different contexts is conflated, but with the precision of fully context-sensitive analysis!

\[
\text{initialize } x_{v,d} = \bot \text{ for each CFG node } v \text{ and each } d \in D
\]

// phase 2a: compute information for function entry nodes and call nodes

\[
\text{initialize worklist } = \{ \text{initial} \} \text{ and set } x_{\text{initial}} = \top \text{ where } \text{initial } = (v_{\text{entry}}, \bullet)
\]

while worklist \(\neq \emptyset\) {

  select and remove an item \((v,d)\) from worklist

  switch \(v\) {
    case entry node: ...
    case call node: ...
  }

// phase 2b: compute information for other CFG nodes

for each node \(v\) that is not a function entry node or a call node {

  for each \(d_1, d_2\) where \(j_f(v, d_1, d_2)\) is non-bottom {
    \[
    x_{v,d} = x_{v,d} \sqcup j_f(v, d_1, d_2)(x_{v_1, d_1}) \text{ where } v_1 \text{ is the function entry of } v
    \]
  }
}
The IDE Tabulation Solver, Phase 2

- Computes an element of \((D \rightarrow L)^n\) represented by \(x_{v,d} \in L\) for each CFG node \(v\) and each \(d \in D\)
- Information for different contexts is conflated, but with the precision of fully context-sensitive analysis!

initialize \(x_{v,d} = \bot\) for each CFG node \(v\) and each \(d \in D\)

// phase 2a: compute information for function entry nodes and call nodes
initialize worklist = \{ initial \} and set \(x_{initial} = \top\) where initial = \((v_{entry}, \bullet)\)

while worklist \(\neq \emptyset\) {
    select and remove an item \((v,d)\) from worklist
    switch \(v\) {
        case entry node: ...
        case call node:
    }
}

// phase 2b: compute information for other CFG nodes
for each node \(v\) that is not a function entry node or a call node {
    for each \(d_1,d_2\) where \(j_f(v,d_1,d_2)\) is non-bottom {
        \(x_{v,d} = x_{v,d} \sqcup j_f(v,d_1,d_2)(x_{v_1,d_1})\) where \(v_1\) is the function entry of \(v\)
    }
}
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: $\text{Vars} \rightarrow \{\text{T, F}\}$
  ...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](https://github.com/Sable/heros)
- **WALA:** [https://github.com/amaurremi/IDE](https://github.com/amaurremi/IDE)

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