Interprocedural, finite, distributive, subset (IDFS) problems [Reps-Horwitz-Sagiv-95]

K. V. Raghavan

IISc

Summary of Sharir-Pneuli approaches

- Requirements of all three approaches (call-strings, functional, iterative)
  - Transfer functions are monotonic
- Additional requirements of functional approach
  - Functions be represented as data-structures
  - The operations of join, composition, and equality-checking be defined on the representation
- Additional requirements of call-strings and iterative approach
  - L is finite
- What they all compute: LFP in general, JVP for distributive functions.

Efficiency of the known algorithms

- Lattice for available expressions analysis for a single expression: $AV \equiv \{\perp, 1, 0\}$.
- Lattice for k expressions?

Efficiency of the known algorithms

- Lattice for available expressions analysis for a single expression: $AV \equiv \{\perp, 1, 0\}$.
- Lattice for k expressions? AV^k

Efficiency of the known algorithms

- Lattice for available expressions analysis for a single expression: $AV \equiv \{\perp, 1, 0\}$.
- Lattice for k expressions? AV^k
  - Call-strings approach would need a very high bound. Iterative approach would need 2^k "columns".
  - Very expensive
- Main insight: If we analyze for each expression separately using AV lattice, we would get same results as analyzing all of them together using AV^k lattice.

Gen-kill problems

- Definition of gen-kill problem:
  - Lattice $L = 2^D$, where D is a finite set
  - Transfer functions are distributive
  - Join is union or intersection
  - For each transfer function f and for each $d \in D$, $f(\{d\}) = f(\emptyset)$ (i.e., f does not "transmit" d) or $f(\{d\}) = f(\emptyset) \cup \{d\}$ (i.e., f "transmits" d).

    (The elements of D that are in $f(\emptyset)$ are said to be "generated" by f. Any element of D that is neither transmitted nor generated by f is said to be "killed" by f.)
- Examples of gen/kill problems: reaching definitions, unitialized variables, live variables (backwards analysis).

More on gen-kill problems

- Another equivalent definition of gen-kill problem:
  - Lattice $L = 2^D$, where D is a finite set
  - Join is union or intersection
  - For each node n, there exist two sets gen_n and $kill_n$, both being subsets of D, such that

$$f_n = \lambda S.(S - kill_n) \cup gen_n$$

Strategy for solving gen-kill problems efficiently

1. Do abstract interpretation separately using lattice $2^{\{d\}}$, for each $d \in D$.
2. At any program point p, final JOP is union of JOP's using individual analyses.
3. Can do all the $|D|$ analyses together also
   - Does not change the order-complexity, but could be more efficient in practice
   - We would need $2|D|$ "columns" with iterative approach

Requirements of [RHS95] approach

- The abstract lattice $L = 2^D$, where D is a finite set
- Transfer functions are distributive (approach is undefined for non-distributive functions)
- Join (in the L lattice) is union or intersection.
  - Paper focuses on union problems. Every intersection problem has a complement problem whose join is union.
- Examples of IDFS problem that is not a gen/kill problem: Possibly uninitialized variables, copy-constant propagation, truly live variables (backward analysis).
- The RHS algorithm computes JVP for IDFS frameworks.

Representing dataflow functions

Definition 3.1. The ***representation relation of f***, $R_f \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$, is a binary relation (*i.e.*, graph) defined as follows:

$$R_f =_{df} \{(\mathbf{0}, \mathbf{0})\} \cup \{(\mathbf{0}, y) \mid y \in f(\emptyset)\} \cup \{(x, y) \mid y \in f(\{x\}) \text{ and } y \notin f(\emptyset)\}. \qquad \square$$

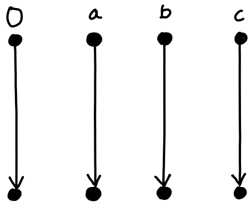
Representation relation can be thought of as a *Graph* with $2(D+1)$ nodes.

Representing dataflow functions

Definition 3.1. The ***representation relation of*** f, $R_f \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$, is a binary relation (*i.e.*, graph) defined as follows:

$$R_f =_{df} \{(\mathbf{0}, \mathbf{0})\} \cup \{(\mathbf{0}, y) \mid y \in f(\emptyset)\} \cup \{(x, y) \mid y \in f(\{x\}) \text{ and } y \notin f(\emptyset)\}. \qquad \square$$

Identity function
f $= \lambda S.S$



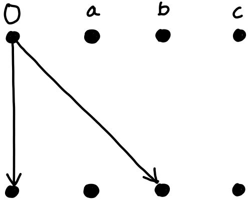
(Credit: Prof. Reps)

Representing dataflow functions

Definition 3.1. The ***representation relation of f***, $R_f \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$, is a binary relation (*i.e.*, graph) defined as follows:

$$
\begin{aligned}
R_f =_{df} \quad & \{ (\mathbf{0}, \mathbf{0}) \} \\
\cup \quad & \{ (\mathbf{0}, y) \mid y \in f(\emptyset) \} \\
\cup \quad & \{ (x, y) \mid y \in f(\{x\}) \text{ and } y \notin f(\emptyset) \}. \qquad \Box
\end{aligned}
$$

Constant function
$f = \lambda S.\{b\}$



(Credit: Prof. Reps)

Representing dataflow functions - II

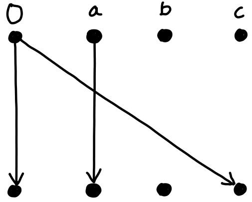
Try constructing the representation relation without the definition!

$$f = \lambda S.(S - \{b\}) \cup \{c\}$$

Representing dataflow functions - II

Try constructing the representation relation without the definition!

$$f = \lambda S.(S - \{b\}) \cup \{c\}$$



(Credit: Prof. Reps)

Representing dataflow functions - Exercise

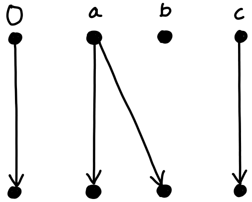
- $\text{f} = \lambda S.\text{if } a \in S$
  $\quad\text{then } S \cup \{b\}$
  $\quad\text{else } S - \{b\}$
- $\text{f} = \lambda S.\text{if } a \in S \text{ OR } b \in S$
  $\quad\text{then } S \cup \{c\}$
  $\quad\text{else } S - \{c\}$

Representing dataflow functions - Exercise

$$
\begin{aligned}
\mathrm{f} = \lambda S.\text{if } a \in S \\
&\text{then } S \cup \{b\} \\
&\text{else } S - \{b\}
\end{aligned}
$$

Representing dataflow functions - Exercise

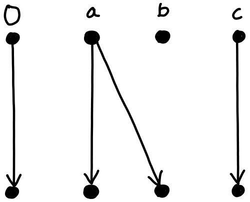
$$\mathrm{f} = \lambda S.\text{if } a \in S \text{ then } S \cup \{b\} \text{ else } S - \{b\}$$



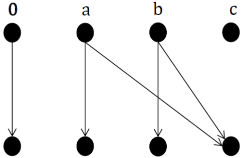
$$\mathrm{f} = \lambda S.\text{if } a \in S \text{ OR } b \in S \text{ then } S \cup \{c\} \text{ else } S - \{c\}$$

Representing dataflow functions - Exercise

$$f = \lambda S.\text{if } a \in S \text{ then } S \cup \{b\} \text{ else } S - \{b\}$$



$$f = \lambda S.\text{if } a \in S \text{ OR } b \in S \text{ then } S \cup \{c\} \text{ else } S - \{c\}$$



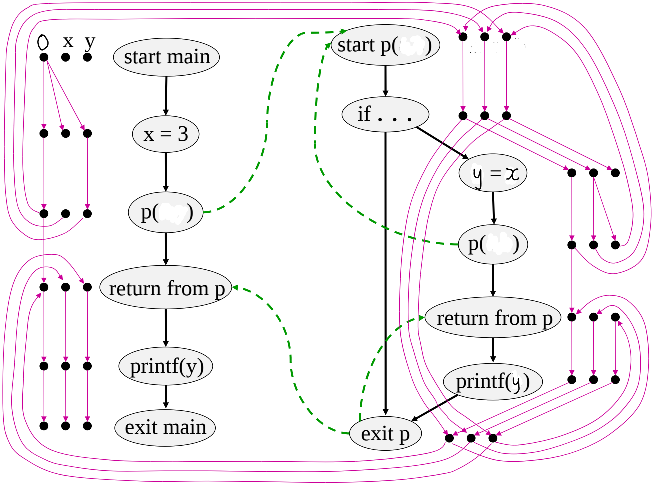
Properties of representation relations

- Interpretation $[[R]] : 2^D \rightarrow 2^D$ of a representation relation $R \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$:

$$[[R]] =_{df} \lambda X.(\{y \mid \exists x \in X.((x,y) \in R)\}) \cup \{y \mid (\mathbf{0}, y) \in R\}) - \{\mathbf{0}\}$$

- Given a collection of functions $f_i : 2^D \rightarrow 2^D$ for $1 \leq i \leq j$,
  $f_1 \circ f_2 \circ \ldots f_j = [[R_{f_j}; R_{f_{j-1}}; \ldots ; R_{f_1}]]$,
  where the composition operation ';' on representation relations is as in Definition 3.4 in the paper.

An example exploded graph for possibly-uninitialized variables using RHS approach



Initial fact is given as {x,y}. Hence, exploded edges are added from $\langle s_{main}, \mathbf{0}\rangle$ to $\langle$ "x=3", $\mathbf{0}\rangle$, $\langle$ "x=3", x $\rangle$, and $\langle$ "x=3", y$\rangle$.

Property of exploded graph

Theorem 3.8 : Given a super graph G^* for an IDFS problem *IP* with lattice D, and its corresponding exploded super graph $G^\#$, for any $d \in D, d \in JVP$ at node n in G^* *iff* there is an *interprocedurally valid* path from $\langle s_{main}, \mathbf{0} \rangle$ to $\langle n, d \rangle$ in $G^\#$. $\square$

Follows from the property that for $1 \leq i \leq j$,
$f_1 \circ f_2 \circ \ldots f_j = [[R_{f_j}; R_{f_{j-1}}; \ldots; R_{f_1}]]$.

In other words, they have reduced dataflow analysis to graph reachability (along inter-procedurally valid paths).

Algorithm

- Objective : to compute the nodes reachable from $\langle s_{main}, \mathbf{0} \rangle$ via interprocedurally valid paths in the exploded super graph $G^{\#}$.
- Worklist based algorithm
- Iteratively computes two sets : *PathEdge* and *SummaryEdge* using dynamic programming.
- The set *PathEdge* stores special edges called *path edges*. Every path edge begins at entry point of a procedure, and ends at some point in the procedure.
- First path edge added to the worklist: $< s_{main}, \mathbf{0} > \rightarrow < s_{main}, \mathbf{0} >$.
- In each iteration, a path edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ is removed from the worklist, and new path edge(s) and/or summary edge are added based on the type of n. (See Figures 3 and 4 in the paper)

Property of path edges

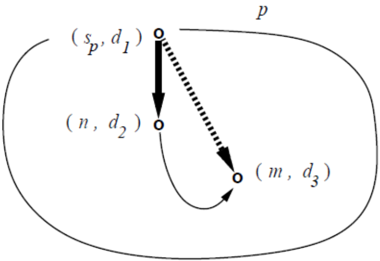
- Path edge $< s_{procOf(N)}, d_1 > \rightarrow < N, d_2 >$ iff
  - there is an inter-procedurally valid path from $< s_{main}, \mathbf{0} >$ to $< s_{procOf(N)}, d_1 >$, *and*
  - there is an inter-procedurally valid and complete path from $< s_{procOf(N)}, d_1 >$ to $< N, d_2 >$.

Property of path edges

- Path edge $< s_{procOf(N)}, d_1 > \rightarrow < N, d_2 >$ iff
  - there is an inter-procedurally valid path from $< s_{main}, \mathbf{0} >$ to $< s_{procOf(N)}, d_1 >$, *and*
  - there is an inter-procedurally valid and complete path from $< s_{procOf(N)}, d_1 >$ to $< N, d_2 >$.
- After the algorithm terminates, for any node N, the following set is equal to JVP$_N$: $\{d_2 \in D \mid \exists d_1 \in (D \cup \{\mathbf{0}\})$ such that $(\langle s_{procOf(N)}, d_1 \rangle, \langle N, d_2 \rangle) \in PathEdge\}$

Algorithm - n is not a call or exit node

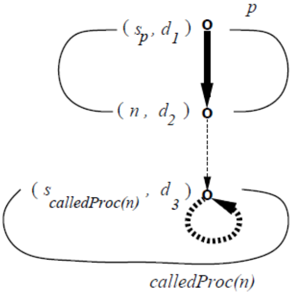
case $n \in (N_p - Call_p - \{ e_p \})$:
  for each $\langle m, d_3 \rangle$ such that $\langle n, d_2 \rangle \to \langle m, d_3 \rangle \in E^{\#}$ **do**
    Propagate($\langle s_p, d_1 \rangle \to \langle m, d_3 \rangle$)
  od
end case



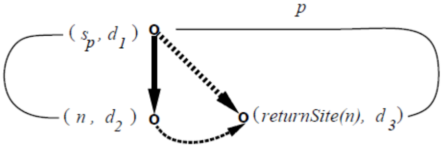
Lines [34]-[36]

Algorithm - n is a call node

case $n \in Call_p$:
  for each d_3 such that $\langle n, d_2 \rangle \rightarrow \langle s_{calledProc(n)}, d_3 \rangle \in E^\#$ **do**
    Propagate($\langle s_{calledProc(n)}, d_3 \rangle \rightarrow \langle s_{calledProc(n)}, d_3 \rangle$)
  od
  for each d_3 such that $\langle n, d_2 \rangle \rightarrow \langle returnSite(n), d_3 \rangle \in (E^\# \cup \text{SummaryEdge})$ **do**
    Propagate($\langle s_p, d_1 \rangle \rightarrow \langle returnSite(n), d_3 \rangle$)
  od
end case



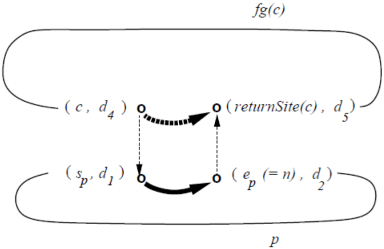
Lines [14]-[16]



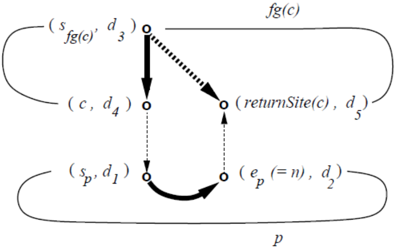
Lines [17]-[19]

Algorithm - n is an exit node

case $n = e_p$:
  for each $c \in callers(p)$ **do**
    for each d_4, d_5 such that $\langle c, d_4\rangle \rightarrow \langle s_p, d_1\rangle \in E^\#$ and $\langle e_p, d_2\rangle \rightarrow \langle returnSite(c), d_5\rangle \in E^\#$ **do**
      if $\langle c, d_4\rangle \rightarrow \langle returnSite(c), d_5\rangle \notin$ SummaryEdge **then**
        Insert $\langle c, d_4\rangle \rightarrow \langle returnSite(c), d_5\rangle$ into SummaryEdge
        for each d_3 such that $\langle s_{procOf(c)}, d_3\rangle \rightarrow \langle c, d_4\rangle \in$ PathEdge **do**
          Propagate($\langle s_{procOf(c)}, d_3\rangle \rightarrow \langle returnSite(c), d_5\rangle$)
        od
      fi
    od
  od
end case



Line [25]



Lines [26]-[28]

A note about summary edges

- Lines 17-19 and Lines 26-28 both add a path edge to a return site. Is there redundancy here? Answer is no.
- Say in Step k of the algorithm Lines 25-28 execute (see fourth rule in Fig. 4). Say in some previous step i ($i < k$), the path edge to the call-site node (c, d_4) was removed from the worklist. The summary edge from from (c, d_4) to $(returnSite(c),\ d_5)$ is being added just now, in Step k. Therefore, in Step i, this summary edge would not have existed. Therefore, in Step i, Lines 17-19 would not have added the path edge to $(returnSite(c),\ d_5)$. Therefore, in Step k, we need Lines 26-28 to add this path edge.
- The converse situation is also possible. Say in Step k a path edge to a call-site node (n, d_2) gets added (see second rule in Fig. 4). It is possible that in an earlier Step i ($i < k$), the summary edge from (n, d_2) to $(returnSite(n),\ d_3)$ was already added. At Step i Lines 26-28 would not have added the path edge to $(returnSite(n),\ d_3)$, because the path edge to (n, d_2) did not exist at that time. Therefore, Lines 17-19 need to add the path edge to $(returnSite(n),\ d_3)$ in Step k.

Algorithm Correctness

Theorem 4.1: Let $X_N = \{d_2 \in D \mid \exists d_1 \in (D \cup \{\mathbf{0}\})$ such that $(\langle s_{procOf(N)}, d_1\rangle, \langle N, d_2\rangle) \in PathEdge\}$ for the node N after the algorithm terminates. Then $X_N = JVP_N$.

Proof of Theorem 4.1: First direction

- Given: $\exists d_1 \in D \cup \{\mathbf{0}\}$ such that $(\langle s_{procOf(N)}, d_1\rangle, \langle N, d_2\rangle) \in \mathit{PathEdge}$ when the tabulation algorithm terminates.
- Invoking Theorem 3.8, it suffices to show that there is an inter-procedurally valid path in the exploded super graph from $\langle s_{main}, \mathbf{0}\rangle$ to $\langle N, d_2\rangle$.
- Argument is by induction on the number iterations of the main loop in the algorithm.
- Induction hypothesis:
  $\forall N, d_1, d_2, (\langle s_{procOf(N)}, d_1\rangle, \langle N, d_2\rangle) \in \mathit{PathEdge} \Rightarrow$
  - there is an inter-procedurally valid path from $< s_{main}, \mathbf{0} >$ to $< s_{procOf(N)}, d_1 >$, *and*
  - there is an inter-procedurally valid and complete path from $< s_{procOf(N)}, d_1 >$ to $< N, d_2 >$.

Proof of Theorem 4.1: The other direction

- Given there is an interprocedurally valid path from $\langle s_{main}, \mathbf{0} \rangle$ to $\langle N, d_2 \rangle)$
- To show that there exists $d_1 \in (D \cup \{\mathbf{0}\})$ such that $(\langle s_{procOf(N)}, d_1 \rangle, \langle N, d_2 \rangle) \in \mathit{PathEdge}$ when the tabulation algorithm terminates.
- Induction on the length of the path
- Induction hypothesis: $\forall N, d_2$, if there is an inter-procedurally valid path of length j from $\langle s_{main}, \mathbf{0} \rangle$ to $\langle N, d_2 \rangle$, then there exists $(\langle s_{procOf(N)}, d_1 \rangle, \langle N, d_2 \rangle) \in \mathit{PathEdge}$ when the tabulation algorithm terminates.

Efficiency

Complexity

- $O(ED^3)$ for general IDFS problems
- $O(ED)$ for gen-kill problems (a special class of problems that includes available expressions, reaching definitions and live variables); comparable to best known algorithm specialized for gen-kill problems.

How it compares to iterative approach

- Plain iterative would need time proportional to $|L|$, which is $2^{|D|}$.
- It's possible to produce a variant of iterative approach that works with values from D, instead of values from L. Still, its complexity is likely to be worse than $O(ED^3)$.