Practical Extensions to the IFDS Algorithm

Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez

University of Waterloo, Canada
{nanaeem,olhotak,j2rodrig}@uwaterloo.ca

Abstract. This paper presents four extensions to the Interprocedural Finite Distributive Subset (IFDS) algorithm that make it applicable to a wider class of analysis problems. IFDS is a dynamic programming algorithm that implements context-sensitive flow-sensitive interprocedural dataflow analysis. The first extension constructs the nodes of the supergraph on demand as the analysis requires them, eliminating the need to build a full supergraph before the analysis. The second extension provides the procedure-return flow function with additional information about the program state before the procedure was called. The third extension improves the precision with which ϕ instructions are modelled when analyzing a program in SSA form. The fourth extension speeds up the algorithm on domains in which some of the dataflow facts subsume each other. These extensions are often necessary when applying the IFDS algorithm to non-separable (i.e. non-bit-vector) problems. We have found them necessary for alias set analysis and multi-object typestate analysis. In this paper, we illustrate and evaluate the extensions on a simpler problem, a variation of variable type analysis.

1 Introduction

The Interprocedural Finite Distributive Subset (IFDS) algorithm [15] is an efficient and precise, context-sensitive and flow-sensitive dataflow analysis algorithm for the class of problems that satisfy its restrictions. Although this class includes the classic bit-vector dataflow problems, the original IFDS algorithm is not directly suitable for more interesting problems for which context- and flow-sensitivity would be useful, particularly problems involving objects and pointers. The algorithm can be extended to solve this larger class of problems, however, and in this paper, we present four such extensions.

The IFDS algorithm is an efficient dynamic programming instantiation of the functional approach to interprocedural analysis [19]. The fundamental restrictions of the algorithm, which we do not seek to eliminate in this paper, are that the analysis domain must be a powerset of some finite set D, and that the dataflow functions must be distributive. We present a detailed overview of the IFDS algorithm in Section 2, and further illustrate the algorithm with a running example variable type analysis in Section 3.

A more practical restriction is that the set D must be small, because the algorithm requires as input a so-called exploded supergraph, and the number of nodes in this supergraph is approximately the product of the size of D and the number of instructions in the program. Our first extension, presented in Section 4, removes the restriction on the size of D by enabling the algorithm to compute only those parts of the supergraph that are actually reached in the analysis. This allows the algorithm to be used for problems in which D is theoretically large, but only a small subset of D is encountered during the analysis, which is typical of analyses modelling objects and pointers.

A second practical restriction of the original IFDS algorithm is that it provides limited information to flow functions modelling return flow from a procedure. For many analyses, mapping dataflow facts from the callee back to the caller requires information about the state before the procedure was called. In Section 5, we extend the IFDS algorithm to provide this information to the return flow function.

A third limitation of many standard dataflow analysis algorithms, IFDS included, is that they can be less precise on a program in Static Single Assignment (SSA) form [2] than on the original non-SSA form of the program. When an instruction has multiple control flow predecessors, incoming dataflow facts are merged before the flow function is applied; this imprecisely models the semantics of ϕ instructions in SSA form. In Section 6, we present an example that exhibits this imprecision, and we extend the IFDS algorithm to avoid it, so that it is equally precise on SSA form as on non-SSA form programs. SSA form is not only a convenience; in prior work, we showed that SSA form can be used to improve running time and space requirements of analyses such as alias set analysis [13].

Finally, the IFDS algorithm does not take advantage of any structure in the set D. In many analyses of objects and pointers, some elements of D subsume others. In Section 7, we present an extension that exploits such structure to reduce analysis time.

We have implemented the IFDS algorithm with all four of these extensions, as well as the running example variable type analysis. In Section 8, we report on an empirical evaluation of the benefits of the extensions. We survey related work in Section 9 and conclude in Section 10.

2 Background: The Original IFDS Algorithm

The IFDS algorithm of Reps et al. [15] is a dynamic programming algorithm that computes a merge-over-all-valid paths solution to interprocedural, finite, distributive, subset problems. The merge is over *valid* paths in that procedure calls and returns are correctly matched (i.e. the analysis is context sensitive). The algorithm requires that the domain of dataflow facts be the powerset of a finite set D, with set union as the merge operator. The data flow functions must be distributive over set union: $f(a) \cup f(b) = f(a \cup b)$.

The algorithm follows the summary function approach to context-sensitive interprocedural analysis [19], in that it computes functions in $\mathcal{P}(D) \to \mathcal{P}(D)$ that summarize the effect of ever-longer sections of code on any given subset of D. The key to the efficiency of the algorithm is the compact representation of these functions, made possible by their distributivity. For example, suppose the set $S = \{a, b, c\}$ is a subset of D. By distributivity, $f(S)$ can be computed as $f(S) = f(\{\}) \cup f(\{a\}) \cup f(\{b\}) \cup f(\{c\})$. Thus every distributive function in $\mathcal{P}(D) \to \mathcal{P}(D)$ is uniquely defined by its value on the empty set and on every singleton subset of D. Equivalently, the function can be defined by a bipartite graph $\langle D \cup \{\mathbf{0}\}, D, E\rangle$, where E is a set of edges from elements of $D \cup \{\mathbf{0}\}$ to elements of (a second copy of) D. The graph contains an edge from d_1 to d_2 if and only if $d_2 \in f(\{d_1\})$. The special **0** vertex represents the empty set: the edge $\mathbf{0} \to d$ indicates that $d \in f(\{\})$. The function represented by the graph is defined to be $f(S) = \{b : (a, b) \in E \wedge (a = \mathbf{0} \vee a \in S)\}$. For example, the graph in Figure 1(a) represents the function $g(S) = \{x : x \in \{b, c\} \vee (x = d \wedge d \in S)\}$, which can be written more simply as $g(S) = (S \setminus \{a\}) \cup \{b, c\}$.

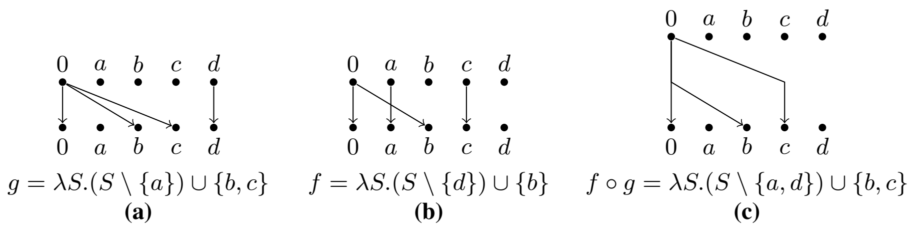

0 a b c d
0 a b c d
$g = \lambda S.(S \setminus \{a\}) \cup \{b, c\}$
(a)
$f = \lambda S.(S \setminus \{d\}) \cup \{b\}$
(b)
$f \circ g = \lambda S.(S \setminus \{a, d\}) \cup \{b, c\}$
(c)

Fig. 1. Compact representation of functions and their composition

The composition $f \circ g$ of two functions can be computed by combining their graphs, merging the nodes of the range of g with the corresponding nodes of the domain of f, then computing reachability from the nodes of the domain of g to the nodes of the range of f. That is, a relational product of the sets of edges representing the two functions gives a set of edges representing their composition. An example is shown in Figure 1. The graph in Figure 1(c), representing $f \circ g$, contains an edge from x to y whenever there is an edge from x to some z in the representation of g in Figure 1(a) and an edge from the same z to y in the representation of f in Figure 1(b).

We have reproduced the original IFDS algorithm [15] in Figure 2. The input to the algorithm is a so-called exploded supergraph that represents both the program being analyzed and the dataflow functions. The supergraph is constructed from the interprocedural control flow graph (ICFG) of the program by replacing each instruction with the graph representation of its flow function. Thus the vertices of the supergraph are pairs $\langle l, d\rangle$, where l is a label in the program and $d \in D \cup \{\mathbf{0}\}$. The supergraph contains an edge $\langle l, d\rangle \rightarrow \langle l', d'\rangle$ if the ICFG contains an edge $l \rightarrow l'$ and $d' \in f(\{d\})$ (or $d' \in f(\{\})$ when $d = \mathbf{0}$), where f is the flow function of the instruction at l. For each interprocedural call or return edge in the ICFG, the supergraph contains a set of edges representing the flow function associated with the call or return. The flow function on the call edge typically maps facts about actuals in the caller to facts about formals in the callee. The merge-over-all-valid paths solution at label l contains exactly the elements d of D for which there exists a valid path from $\langle s, \mathbf{0}\rangle$ to $\langle l, d\rangle$ in the supergraph. The dataflow analysis therefore reduces to valid-path reachability on the supergraph.

The IFDS algorithm works by incrementally constructing two tables, PathEdge and SummaryEdge, representing the flow functions of ever longer sequences of code. The PathEdge table contains triples $\langle d, l, d'\rangle$, indicating that there is a path from $\langle s_p, d\rangle$ to $\langle l, d'\rangle$, where s_p is the start node of the procedure containing l. These triples are often written in the form $\langle s_p, d\rangle \rightarrow \langle l, d'\rangle$ for clarity, but the start node s_p is uniquely determined by l, so it is not stored in an actual implementation. The SummaryEdge table contains triples $\langle c, d, d'\rangle$, where c is the label of a call site. Such a triple indicates that $d' \in f(\{d\})$, where f is a flow function summarizing the effect of the procedure called at c. These triples are often written $\langle c, d\rangle \rightarrow \langle r, d'\rangle$, where r is the instruction following c. For convenience, Reps's presentation of the IFDS algorithm [15] assumes that in the ICFG, every call site c has a single successor, a no-op "return site" node r.

The PathEdge and SummaryEdge tables are interdependent. Consider the edge $\langle s_p, d_1\rangle \rightarrow \langle e_p, d_2\rangle$ added to PathEdge, in which e_p is the exit node of some procedure p. This edge means that $d_2 \in f_p(\{d_1\})$, where f_p is the flow function representing the effect of the entire procedure p. As a result, for every call site c calling procedure p, a corresponding

```
    declare PathEdge, WorkList, SummaryEdge: global edge set
    algorithm Tabulate($G^\sharp_{IP}$)
    begin
1     Let ($N^\sharp$, $E^\sharp$) = $G^\sharp_{IP}$
2     PathEdge:={ $\langle s_{main}, \mathbf{0}\rangle \rightarrow \langle s_{main}, \mathbf{0}\rangle$ }
3     WorkList:={ $\langle s_{main}, \mathbf{0}\rangle \rightarrow \langle s_{main}, \mathbf{0}\rangle$ }
4     SummaryEdge:= $\emptyset$
5     ForwardTabulateSLRPs()
6     foreach $n \in N^\sharp$ do
7       $X_n$ := { $d_2 \in D | \exists d_1 \in (D \cup \{\mathbf{0}\})$ s.t. $\langle s_{procOf(n)}, d_1\rangle \rightarrow \langle n, d_2\rangle \in$ PathEdge
8     od
    end
    procedure Propagate(e)
    begin
9     if $e \notin$ PathEdge then Insert e into PathEdge; Insert e into WorkList; fi
    end
    procedure ForwardTabulateSLRPs()
    begin
10    while WorkList $\neq \emptyset$ do
11      Select and remove an edge $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ from WorkList
12      switch $n$
13        case $n \in Call_p$ :
14          foreach $d_3$ s.t. $\langle n, d_2\rangle \rightarrow \langle s_{calledProc(n)}, d_3\rangle \in E^\sharp$ do
15            Propagate($\langle s_{calledProc(n)}, d_3\rangle \rightarrow \langle s_{calledProc(n)}, d_3\rangle$)
16          od
17          foreach $d_3$ s.t. $\langle n, d_2\rangle \rightarrow \langle returnSite(n), d_3\rangle \in (E^\sharp \cup$ SummaryEdge$)$ do
18            Propagate($\langle s_p, d_1\rangle \rightarrow \langle returnSite(n), d_3\rangle$)
19          od
20        end case
21        case $n \in e_p$ :
22          foreach $c \in callers(p)$ do
23            foreach $d_4$,$d_5$ s.t. $\langle c, d_4\rangle \rightarrow \langle s_p, d_1\rangle \in E^\sharp$ and
                                    $\langle e_p, d_2\rangle \rightarrow \langle returnSite(c), d_5\rangle \in E^\sharp$ do
24              if $\langle c, d_4\rangle \rightarrow \langle returnSite(c), d_5\rangle \notin$ SummaryEdge then
25                Insert $\langle c, d_4\rangle \rightarrow \langle returnSite(c), d_5\rangle$ into SummaryEdge
26                foreach $d_3$ s.t. $\langle s_{procOf(c)}, d_3\rangle \rightarrow \langle c, d_4\rangle \in$ PathEdge do
27                  Propagate($\langle s_{procOf(c)}, d_3\rangle \rightarrow \langle returnSite(c), d_5\rangle$)
28                od
29              fi
30            od
31          od
32        end case
33        case $n \in (N_p - Call_p - \{e_p\})$ :
34          foreach $\langle m, d_3\rangle$ s.t. $\langle n, d_2\rangle \rightarrow \langle m, d_3\rangle \in E^\sharp$ do
35            Propagate($\langle s_p, d_1\rangle \rightarrow \langle m, d_3\rangle$)
36          od
37        end case
38      end switch
39    od
    end
```

Fig. 2. Original IFDS Algorithm reproduced from [15]

triple must be added to SummaryEdge indicating the newly-discovered effect at that call site. In fact, several such triples may be needed for a single edge added to PathEdge, since the effect of a procedure at c is represented not just by f_p, but by the composition $f_r \circ f_p \circ f_c$, where f_c and f_r are the flow functions representing the function call and return. This composition is computed by combining the graphs representing f_c and f_r from the supergraph with the newly discovered edge $\langle d_1, d_2 \rangle$ of f_p. That is, for each d_4 and d_5 such that $\langle d_4, d_1 \rangle \in f_c$ and $\langle d_2, d_5 \rangle \in f_r$, $\langle c, d_4, d_5 \rangle$ is added to SummaryEdge. This is performed in lines 23 to 25 of the algorithm.

Conversely, consider a triple $\langle c, d_4, d_5 \rangle$ added to SummaryEdge, indicating a new effect of the call at c. As a result, for each d_3 such that there is a path from $\langle s, d_3 \rangle$ to $\langle c, d_4 \rangle$, where s is the start node of the procedure containing c, there is now a valid path from $\langle s, d_3 \rangle$ to $\langle r, d_5 \rangle$, where r is the successor of c. Thus $\langle s, d_3 \rangle \rightarrow \langle r, d_5 \rangle$ must be added to PathEdge. This is performed in lines 26 to 28 of the algorithm.

3 Running Example: Type Analysis

The extensions to the IFDS algorithm presented in this paper were originally motivated by context-sensitive alias set analysis [13] and multi-object typestate analysis [12]. The same extensions are applicable to many other kinds of analyses. In this paper, we will use a much simpler analysis as a running example to illustrate the IFDS extensions.

The example analysis is a variation of Variable Type Analysis (VTA) [21] for Java. The analysis computes the set of possible types for each variable. This information can be used to construct a call graph or to check the validity of casts. At each program point p, the analysis computes a subset of D, where D is defined as the set of all pairs $\langle v, t \rangle$, where v is a variable in the program and t is a class in the program. The presence of the pair $\langle v, t \rangle$ in the subset indicates that the variable v may point to an object of type t.

For the sake of the example, we would like the analysis to analyze only the application code and not the large standard library. The analysis therefore makes conservative assumptions about the unanalyzed code based on statically declared types. For example, if `m()` is in the library, the analysis assumes that `m()` could return an object of the declared return type of `m()` or any of its subtypes. To this end we amend the meaning of a pair $\langle v, t \rangle$ to indicate that v may point to an object of type t or any of its subtypes.

The unanalyzed code could write to fields in the heap, either directly or by calling back into application code. To keep the analysis sound yet simple, we make the conservative assumption that a field can point to any object whose type is consistent with its declared type. We model a field read `x = y.f` with the pair $\langle \mathtt{x}, t \rangle$, where t is the declared type of `f`. We make these simplifications because the analysis is intended to illustrate the extensions to the IFDS algorithm, not necessarily as a practical analysis.

When the declared type of a field is an interface, the object read from it could be of any class that implements the interface. For a read from such a field, we generate multiple pairs $\langle \mathtt{x}, t_i \rangle$, where the t_i are all classes that implement the interface. If class A extends B and both implement the interface, it is redundant to include $\langle \mathtt{x}, B \rangle$ since $\langle \mathtt{x}, A \rangle$ already includes all subclasses of A, including B. For efficiency, we generate only those pairs $\langle \mathtt{x}, t_i \rangle$ where t_i implements the interface and its superclass does not.

The analysis is performed on an intermediate representation comprising the following kinds of instructions, in addition to procedure calls and returns: $s ::= x \leftarrow y \mid y.f \leftarrow x \mid x \leftarrow y.f \mid x \leftarrow \textbf{null} \mid x \leftarrow \textbf{new}\ T \mid x \leftarrow (T)y$. The instructions copy pointers between variables, store and load objects to and from fields, assign **null** to variables, create new objects and cast objects to a given type, respectively. We use $[\![s]\!]_P : \mathcal{P}(D) \to \mathcal{P}(D)$ to denote the transfer function for the type analysis. The IFDS algorithm requires the transfer function to be decomposed into its effect on each individual element of D and on the empty set. We decompose it as $[\![s]\!] : D \cup \{\mathbf{0}\} \to \mathcal{P}(D)$ and define $[\![s]\!]_P(P) \triangleq [\![s]\!](\mathbf{0}) \cup \bigcup_{d \in P} [\![s]\!](d)$. The decomposed transfer function $[\![s]\!]$ is defined in Figure 3.

$$
\begin{aligned}
[\![x \leftarrow y]\!](\langle v,t\rangle) &\triangleq \begin{cases} \{\langle x,t\rangle, \langle y,t\rangle\} & \text{if } v = y \\ \{\langle v,t\rangle\} & \text{if } v \neq y \text{ and } v \neq x \\ \emptyset & \text{if } v \neq y \text{ and } v = x \end{cases} \\
[\![y.f \leftarrow x]\!](\langle v,t\rangle) &\triangleq \{\langle v,t\rangle\} \\
[\![x \leftarrow \mathbf{null} | \mathbf{new}\ T | y.f]\!](\langle v,t\rangle) &\triangleq \begin{cases} \{\langle v,t\rangle\} & \text{if } v \neq x \\ \emptyset & \text{otherwise} \end{cases} \\
[\![x \leftarrow \mathbf{new}\ T]\!](\mathbf{0}) &\triangleq \{\langle x,T\rangle\} \\
[\![x \leftarrow y.f]\!](\mathbf{0}) &\triangleq \{\langle x,c\rangle : c \in \text{implClasses}(\mathit{type}(f))\} \\
[\![x \leftarrow (T)y]\!](\langle v,t\rangle) &\triangleq \bigcup_{c \in \text{implClasses}(T)} \mathit{cast}(x,y,c)(\langle v,t\rangle) \\
\mathit{cast}(x,y,t_2)(\langle v,t_1\rangle) &\triangleq \begin{cases} \{\langle v,t_1\rangle\} & \text{if } v \neq x \text{ and } v \neq y \\ \emptyset & \text{if } v = x \text{ and } v \neq y \\ \{\langle x,t_1\rangle, \langle y,t_1\rangle\} & \text{if } v = y \text{ and } t_1 <: t_2 \\ \{\langle x,t_2\rangle, \langle y,t_2\rangle\} & \text{if } v = y \text{ and } t_2 <: t_1 \\ \emptyset & \text{if } v = y \text{ and } t_1 \text{ and } t_2 \text{ are unrelated} \end{cases} \\
[\![s]\!](\mathbf{0}) &\triangleq \emptyset \text{ if } s \neq x \leftarrow y.f \text{ and } s \neq x \leftarrow \mathbf{new}
\end{aligned}
$$

Fig. 3. Intraprocedural flow functions for the running example type analysis

The flow function for a copy instruction ($x \leftarrow y$) applied to a pair $\langle v,t\rangle$ requires three cases. When v is the same as y, the pair $\langle v,t\rangle$ is preserved and, since the value of y is copied to x, a new pair $\langle x,t\rangle$ is created. If v is neither x nor y, the value of v is unaffected by the copy and the pair is therefore preserved. If v is x, and x and y are distinct, then since the existing value of x is overwritten by the new value, the existing pair $\langle v,t\rangle$ describing the old value of v is discarded, and the result is the empty set.

The store instruction ($v.f \leftarrow x$) has no effect on the values of local variables, and its flow function is therefore the identity. The flow function for an assignment to x via a load, **new** or **null** does not affect $\langle v,t\rangle$, unless v is x, in which case the existing value of x is overwritten, so the pair is dropped from the set. An allocation instruction $x \leftarrow$ **new** T generates the new pair $\langle x,T\rangle$. A load instruction $x \leftarrow y.f$ creates the pair $\langle x,t\rangle$, if the type of the field f is a class t, or the set of pairs $\langle x,t_i\rangle$, if the type of the field f is an interface, where the t_i are all of the classes implementing the interface, as explained earlier. The helper function implClasses(t) computes this set of classes.

The most interesting case is the cast instruction ($x \leftarrow (T)y$). The first complication is that T could be an interface. Such a cast is treated as casts to all classes implementing T. The flow function is the union of the flow functions modelling casts to these classes, reflecting the fact that the cast to the interface type succeeds if the cast to at least one of the implementing classes succeeds. For the simpler case of a cast to a type t_2 that is a class, not an interface, there are still several cases. The cast instruction has no effect on $\langle v, t_1 \rangle$ when v is neither x nor y. When v is x, the pair is dropped because the cast overwrites the existing value of x. When v is y and $t_1 <: t_2$, indicating that we already know that y points to an object whose type is a subtype of t_2, the cast acts as a copy and the new pair $\langle x, t_1 \rangle$ is generated. When v is y and $t_2 <: t_1$, indicating that y is being cast to a more restrictive type than the type it is already known to point to, we generate the new pair $\langle x, t_2 \rangle$, indicating that x must point to a subtype of the more restrictive cast type. The original pair $\langle y, t_1 \rangle$ can also be changed to the more precise pair $\langle y, t_2 \rangle$, since if control flow proceeds after the cast, the cast must have succeeded, and therefore y must point to an object whose type is a subtype of the cast type. For the purposes of the example, we assume that a failing cast terminates the program rather than being caught by an exception handler; catching class cast exceptions is rare in practice.

4 Demand Construction of the Supergraph

The number of nodes in the exploded supergraph $G^\sharp$ is $|\text{Inst}| \times (|D| + 1)$, where $|\text{Inst}|$ is the number of instructions in the program and $|D|$ is the size of D. In many analyses, D, though finite, is very large. For example, in an alias set analysis [13], D is a union of the powersets of the sets of variables of all procedures, and therefore exponential in the number of variables in a procedure. In our example variable type analysis, $D = |\text{Var}| \times |\text{Class}|$, where Var is the set of all variables in the program and Class is the set of all classes in the program, so $|D|$ is over one million even for a moderate program with a thousand variables and a thousand classes. Constructing and storing a graph that is a million times larger than the ICFG is not practical. In practice, only a small subgraph of $G^\sharp$ is reachable by valid paths from $\langle s_{main}, \mathbf{0} \rangle$ and therefore explored by the algorithm. Unfortunately, we cannot know exactly which subgraph this is before running the IFDS algorithm, since determining which nodes are reachable is exactly what the IFDS algorithm does. Therefore, our first extension to the IFDS algorithm modifies it to request only those parts of the supergraph that it encounters, instead of requiring the whole supergraph as input.

The extended IFDS algorithm with all four of our extensions is shown in Figure 4. Parts of the algorithm that were changed from the original or added are underlined.

The input to the extended algorithm is a function that, given a supergraph node $n^\sharp$, computes all of the edges leaving that node (i.e. the flow function of the desired analysis). For clarity of presentation, we have split this function into four separate functions:

- flow$(n^\sharp)$ computes all intraprocedural edges.[1]
- passArgs$(n^\sharp)$ computes call-to-start edges when $n^\sharp$ is at a call site.
- returnVal$(n^\sharp)$ computes exit-to-return-site edges when $n^\sharp$ is at the exit of a procedure.[2]
- callFlow$(n^\sharp)$ computes call-to-return-site edges when $n^\sharp$ is at a call site. These edges model procedure-local information that is not affected by the called procedure.

The original IFDS algorithm queries the edges of the supergraph $E^\sharp$ in five places. The queries on lines 14, 17 and 34, and the second query on line 23 can simply be replaced by calls to passArgs, callFlow, flow, and returnVal, respectively.

[1] In Figure 4, flow has a second parameter π, which will be explained in Section 6.

[2] In Figure 4, returnVal has a second node parameter, which will be explained in Section 5.

```
declare PathEdge, WorkList, SummaryEdge, Incoming, EndSummary: global
algorithm Tabulate(flow, passArgs, returnVal, callFlow)
     ⋮
procedure ForwardTabulateSLRPs()
begin
10     while WorkList $\neq \emptyset$ do
11       Select and remove an edge $\langle s_p, d_1\rangle \xrightarrow{\pi} \langle n, d_2\rangle$ from WorkList
12       switch $n$
13         case $n \in Call_p$ :
14           foreach $d_3 \in$ passArgs($\langle n, d_2\rangle$) do
15             Propagate$\left(\left\langle s_{calledProc(n)}, d_3\right\rangle \xrightarrow{0} \left\langle s_{calledProc(n)}, d_3\right\rangle\right)$
15.1           Incoming $\left[\left\langle s_{calledProc(n)}, d_3\right\rangle\right] \cup= \langle n, d_2\rangle$
15.2           foreach $\langle e_p, d_4\rangle \in$ EndSummary $\left[\left\langle s_{calledProc(n)}, d3\right\rangle\right]$ do
15.3             foreach $d_5 \in$ returnVal($\langle e_p, d_4\rangle$ , $\langle n, d_2\rangle$) do
15.4               Insert $\langle n, d_2\rangle \rightarrow \langle returnSite(n), d_5\rangle$ into SummaryEdge
15.5             od
15.6           od
16           od
17           foreach $d_3$ s.t. $d_3 \in$ callFlow($\langle n, d2\rangle$) or
                                $\langle n, d_2\rangle \rightarrow \langle returnSite(n), d_3\rangle \in$ SummaryEdge do
18             Propagate$\left(\langle s_p, d_1\rangle \xrightarrow{n} \langle returnSite(n), d_3\rangle\right)$
19           od
20         end case
21         case $n \in e_p$ :
21.1         EndSummary $[\langle s_p, d_1\rangle] \cup= \langle e_p, d_2\rangle$
22           foreach $\langle c, d_4\rangle \in$ Incoming $[\langle s_p, d_1\rangle]$ do
23             foreach $d_5 \in$ returnVal($\langle e_p, d_2\rangle$ , $\langle c, d_4\rangle$) do
24               if $\langle c, d_4\rangle \rightarrow \langle returnSite(c), d_5\rangle \notin$ SummaryEdge then
25                 Insert $\langle c, d_4\rangle \rightarrow \langle returnSite(c), d_5\rangle$ into SummaryEdge
26                 foreach $d_3$ s.t. $\left\langle s_{procOf(c)}, d_3\right\rangle \rightarrow \langle c, d_4\rangle \in$ PathEdge do
27                   Propagate$\left(\left\langle s_{procOf(c)}, d_3\right\rangle \xrightarrow{c} \langle returnSite(c), d_5\rangle\right)$
28                 od
29               fi
30             od
31           od
32         end case
33         case $n \in (N_p - Call_p - \{e_p\})$ :
34           foreach $m, d_3$ s.t. $n \rightarrow m \in$ CFG and $d_3 \in$ flow($\langle n, d_2\rangle$ , $\pi$) do
35             Propagate$\left(\langle s_p, d_1\rangle \xrightarrow{n} \langle m, d_3\rangle\right)$
36           od
37         end case
38       end switch
39     od
end
```

Fig. 4. Extended IFDS Algorithm

However, the first query on line 23 asks to evaluate the inverse of the flow function: find all call nodes $\langle c, d_4 \rangle$ from which an edge leads to the procedure start node $\langle s_p, d_1 \rangle$. This would require computing the inverse of the flow function, which can be difficult for many analyses. Moreover, even though $\langle s_p, d_1 \rangle$ is reachable in $G^\sharp$, many of its predecessors in $E^\sharp$ may not be, and enumerating them may be intractable. For example, for an alias set analysis, the number of predecessors for most nodes is $2^{|\text{Var}|-1}$, where $|\text{Var}|$ is the number of variables in the calling procedure. The extended algorithm therefore maintains a set Incoming$[\langle s_p, d_1 \rangle]$ that records nodes that the analysis has observed to be reachable and predecessors of $\langle s_p, d_1 \rangle$. Whenever the call to passArgs$(\langle n, d_2 \rangle)$ in line 14 returns $\langle s_p, d_3 \rangle$, $\langle n, d_2 \rangle$ is added in line 15.1 to Incoming$(\langle s_p, d_3 \rangle)$.

An obvious issue with querying the set of nodes already observed to be predecessors of $\langle s_p, d_1 \rangle$ is what must be done when a new predecessor is observed later. The solution is to keep track of exit nodes for which a given value of Incoming has been queried (line 21.1). Then, whenever a new predecessor is observed, those exit nodes are reprocessed to reflect the new predecessor. A simple way to reprocess the exit nodes correctly is to add them to the worklist. However, this approach is very inefficient, because whenever a new predecessor is added at one call site, the effect of the procedure is reprocessed for all predecessors at all call sites of the procedure. This intuitively poor performance was confirmed by our experience with the initial implementation of the algorithm.

A better way to reprocess the exit node is to recognize that when a new predecessor of $\langle s_p, d_1 \rangle$ is observed, the predecessor tells us the relevant call site. Instead of adding the corresponding exit node to the worklist, we can immediately process that exit node, but do only the work necessary for that one predecessor. Concretely, we duplicate the effect of lines 24 through 29 after line 15.1. The effect of lines 24, 25 and 29, adding the appropriate edge to SummaryEdge, is done in lines 15.3 through 15.5. The effect of lines 26 through 28 is already done by lines 17 through 19 of the original algorithm.

5 Return Flow Functions

```
void ensureCircle(Shape y){
    Shape z = y;
    (Circle) z;
}
Shape x = ...;
ensureCircle(x);
```

In the original IFDS algorithm, the return flow function is modelled by interprocedural edges in the exploded supergraph that lead from the exit of a procedure to the call site that called the procedure. In the callee, each flow fact is represented in terms of the local scope of the callee. For many analyses, it is necessary to map information in the callee back to the caller. For example, in the code on the right, the cast inside `ensureCircle` succeeds only if the object pointed to by `z`, which is also pointed to by `x` and `y`, is of type `Circle` or its subtype. Therefore, if `ensureCircle` returns normally, we know that `x` cannot point to an arbitrary `Shape`, but only to a `Circle`. However, the original IFDS algorithm cannot discover this fact: although it determines that at the exit of `ensureCircle`, `z` points to an object of type `Circle`, there is no way in the supergraph to associate `z` in the callee with `x` in the caller.

Yet with a small extension, this reverse mapping can be recovered. The fact that `z` points to a subtype of `Circle` is expressed by the edge $\langle s_{\texttt{ensureCircle}}, \langle \texttt{y}, \texttt{Shape} \rangle \rangle \to \langle e_{\texttt{ensureCircle}}, \langle \texttt{z}, \texttt{Circle} \rangle \rangle$ in PathEdge. This edge means that at the beginning of the procedure, there was an object pointed to by `y`, and at the exit of the procedure, the same object is pointed to by `z` and we know it is of type `Circle`. In addition, Incoming$[\langle s_{\texttt{ensureCircle}}, \langle \texttt{y}, \texttt{Shape}\rangle\rangle]$ contains $\langle c, \langle \texttt{x}, \texttt{Shape}\rangle\rangle$. This means that the object passed in through `y` from the call site c was pointed to by `x` in the caller scope. We can combine the context information provided by Incoming with the intraprocedural information computed in PathEdge to determine that the object pointed to by `x` at the call site is known to be of type `Circle` after the call.

This extension appears in the extended algorithm in Figure 4 on line 23. The returnVal function takes, in addition to the node d_2 at the exit instruction e_p, a second node d_4 at the call site c. These arguments indicate not only that the node d_2 is reachable at e_p, but that it is reachable from some node d_1 at the start instruction s_p of the procedure, and that a passArgs edge leads to the latter node from node d_4 at the call site c. Thus the returnVal function can use the caller-side state from the time the procedure was invoked to map the callee-side state at the exit of the procedure back to the caller-side context.

This extension is not merely an extension of the IFDS algorithm, but an extension of the exploded supergraph abstraction that the algorithm is based on. In the supergraph, for every pair of nodes d_2 at an exit node and d_5 at a return site, there either is or is not an edge from d_2 to d_5; if there is such an edge, the algorithm adds a SummaryEdge from $\langle c, d_4\rangle$ to $\langle \textit{returnSite}(c), d_5\rangle$ for every call site c calling the procedure and for every reachable node d_4 at c. However, the extended algorithm gives the analysis designer more flexbility, in that the decision to add the SummaryEdge is additionally dependent on the specific call-site node $\langle c, d_4\rangle$ being considered. It is as if the supergraph edge $\langle e_p, d_2\rangle \rightarrow \langle \textit{returnSite}(c), d_5\rangle$ can both exist and not exist, depending on which call site node $\langle c, d_4\rangle$ is being taken on the path used to reach $\langle e_p, d_2\rangle$.

6 Static Single Assignment (SSA) Form

Static Single Assignment (SSA) form [2] is a popular intermediate representation that makes many program analyses simpler and more efficient. Standard dataflow analysis algorithms such as the original IFDS can be applied unchanged to programs in SSA form, but without appropriate extensions, such an analysis may be less precise than when the same analysis is done on the original, non-SSA version of the program. In this section, we discuss the reasons for the precision loss and propose an extension to the IFDS algorithm that fully restores the lost precision. The extended algorithm analyzes a program in SSA form as precisely as in its original, non-SSA form.

The defining feature of SSA form is that every variable is written to in only one instruction in the program. To convert a program to SSA form, every variable is renamed at each of its definitions, so each definition writes to a fresh, unique variable. Every use of a variable must also be renamed to match the reaching definition. A problem arises when multiple definitions reach a use: to which of the new names should the variable at the use be renamed? The solution is to add ϕ pseudo-instructions to select the reaching definition based on the control flow path taken. A ϕ instruction at a control flow merge point defines a new variable whose value is selected from among the reaching definitions depending on the edge taken into the merge point. Thus only the ϕ definition of the variable reaches the instructions following the merge.

The ϕ pseudo-instruction differs from normal instructions in two ways. First, if multiple variables require ϕ assigments at a given merge point, the ϕ assignments are performed simultaneously, in parallel. The set of ϕ instructions at the merge point defines, for each incoming control-flow edge, a permutation of the variables. Thus it is clearer to group all of the ϕ instructions at a given merge point into a single multi-variable ϕ instruction. Multiple instructions in sequence would suggest that the operations are performed one after the other, which is an incorrect semantics for ϕ instructions.

Second, unlike other instructions, the effect of a ϕ depends on the control-flow edge taken to reach the instruction. This causes many dataflow analysis algorithms, including the original IFDS, to lose precision when analyzing a program in SSA form, compared to analyzing the same program in its original non-SSA form. We will present an example program that exhibits such precision loss in Section 6.1. In most dataflow analyses, at a control flow merge point, the analysis first merges the dataflow facts from the incoming edges, then passes the merged value to the flow function of the instruction after the merge (i.e. $\mathit{out}[s] = f_s(\bigcup_{p \in \mathit{pred}(s)} \mathit{out}[p])$). Merging before applying the flow function reflects the structure of the control flow graph, and is appropriate when the merge is followed by a non-ϕ instruction. When the merge is followed by a ϕ instruction, however, the merge preceding the flow function application makes it impossible for the flow function f_s to depend on the control flow predecessor that its input came from, since the inputs from all the predecessors have been merged into a single dataflow value. Most dataflow analyses treat a ϕ instruction such as $x_3 = \phi(x_1, x_2)$ as an assignment from both x_1 and x_2 to x_3, ignoring the control flow edges on which those values of x_1 and x_2 arrived.

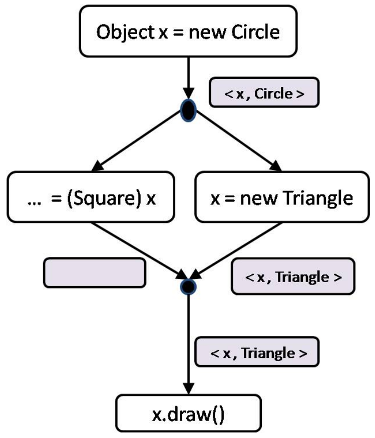
To analyze SSA-form code as precisely as non-SSA-form code, the merge must be delayed until *after* the ϕ instruction. That is, the ϕ flow function is applied separately to the dataflow value on each incoming control flow path, and the merge is performed on the *outputs* of the ϕ flow function, not on its input. As a result, the incoming control flow edge associated with each dataflow value can be made available to the flow function f_ϕ modelling the ϕ instruction. Formally, $\mathit{out}[\phi] = \bigcup_{p \in \mathit{pred}(\phi)} f_\phi(p, \mathit{out}[p])$.

Extending the IFDS algorithm to perform dataflow merges after ϕ instructions instead of before them requires two modifications. First, every edge added to PathEdge is annotated with a control flow predecessor. The edge $\langle s_p, d_1 \rangle \xrightarrow{n} \langle m, d_2 \rangle$ indicates that there is a path in the supergraph starting at the dataflow fact d_1 at the start node s_p, leading to the dataflow fact d_2 at node m, and that the second-last node on the path is at node n. In other words, the dataflow fact d_2 reaches m along the incoming control flow edge from n. Two PathEdge edges that differ only in the control flow predecessor are considered to be distinct. The PathEdge edges created in lines 18, 27, and 35 of the algorithm are annotated with the control flow predecessor, shown above the arrow. The PathEdge edge created in line 15 corresponds to the empty path from $\langle s, d_3 \rangle$ to itself, so there is no control flow predecessor to record. We therefore use a dummy predecessor, which we write as 0. However, the target of this edge is the start node of the procedure, which is never a ϕ instruction, so the predecessor will never be needed for this node.

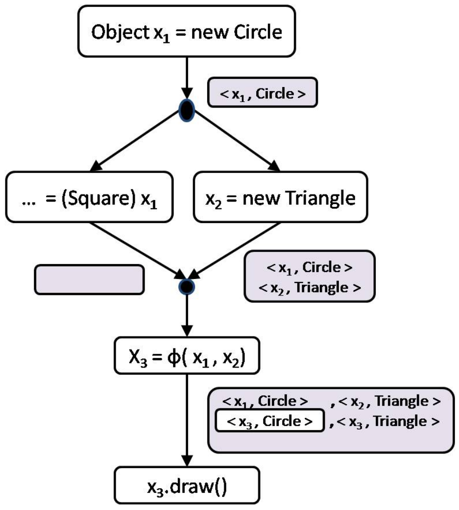
Second, the flow function is extended with a second parameter, and when the function is called in line 34, the control-flow predecessor π of the PathEdge edge currently being processed is passed in. Thus the flow function for the ϕ instruction can depend on the control-flow predecessor π associated with the dataflow value d_2 reaching n.

```
Object x = new Circle
if (cond) ... = (Square) x;
else x = new Triangle;
x.draw();
```

(a) Original Source Code



(b) Non-SSA Type Results


Object x_1 = new Circle
< x_1, Circle >
... = (Square) x_1
x_2 = new Triangle
< x_1, Circle > < x_2, Triangle >
X_3 = ϕ(x_1, x_2)
< x_1, Circle > , < x_2, Triangle >
< x_3, Circle > , < x_3, Triangle >
x_3.draw()

(c) SSA Type Results

Fig. 5. The effect on precision due to the choice of merge strategy at ϕ nodes.

An obvious optimization is to annotate only those edges $\langle s_p, d_1 \rangle \rightarrow \langle m, d_2 \rangle$ in which m is a ϕ instruction, and leave all other edges unannotated. We do this in our implementation, but have not shown it in Figure 4 to avoid cluttering the algorithm.

6.1 Example of precision loss

An example of how merging dataflow information before rather than after a ϕ instruction reduces precision is shown in Figure 5. The original non-SSA source code of the example program is in Figure 5(a). A variable x is initialized as a `Circle`. In the left branch of the conditional, x is cast to `Square`. In the right branch, x is redefined as a `Triangle`. Figure 5(b) shows the results of running the type analysis on the code. The flow function for the cast operation kills the flow fact $\langle x, \texttt{Circle} \rangle$, since a `Circle` cannot be successfully cast to a `Square`. Therefore, the type analysis indicates that the only possible type for receiver x at instruction `x.draw()` is `Triangle`. This is sound since the cast operation can never succeed and therefore a program executing the left branch can never reach the `draw` call. Conversely, if the program reaches the `draw` call it must have taken the right branch and the receiver must be a `Triangle`.

Figure 5(c) shows the same code after SSA conversion. The receiver x_3 for the call x_3`.draw()` is x_1 when the path follows the left branch and x_2 when the path follows the right branch, as reflected in the ϕ function. The left predecessor of the ϕ function has no flow facts because the cast kills $\langle x_1, \texttt{Circle} \rangle$ as before. The right predecessor has the facts $\langle x_1, \texttt{Circle} \rangle$ and $\langle x_2, \texttt{Triangle} \rangle$. The original IFDS algorithm would first merge the incoming flow facts from the two branches, then apply the flow function that models the ϕ as a copy from both x_1 and x_2. At the call to x_3`.draw()`, the analysis would compute the facts $\langle x_3, \texttt{Circle}\rangle$ and $\langle x_3, \texttt{Triangle}\rangle$, which is less precise than the non-SSA version of the analysis that was able to rule out x being a `Circle`.

In the extended IFDS algorithm, the merge is not performed before the flow function of the ϕ instruction, so the flow function has information about the control flow edge on which each dataflow fact arrives. For facts coming in from the left edge, it models a copy from x_1 to x_3; for facts coming in from the right edge, it models a copy from x_2 to x_3. Thus only the fact $\langle x_2, \texttt{Triangle}\rangle$ coming from the right edge leads to a new fact $\langle x_3, \texttt{Triangle}\rangle$. The fact $\langle x_1, \texttt{Circle}\rangle$ does not give rise to $\langle x_3, \texttt{Circle}\rangle$, as it did before, because it comes in from the right edge, which is not associated with a copy from x_1 to x_3. Thus the extended IFDS algorithm achieves the same precision on the SSA-form version of the program as on the original non-SSA-form version.

7 Exploiting Structure in the Set D

The IFDS algorithm requires that the dataflow domain be the powerset of a finite set D. The elements of D are treated independently and equally. The algorithm does not assume or take advantage of any relationships between the elements of D. This is appropriate for bit-vector dataflow problems. For example, the liveness of variable x at some program point implies nothing about the liveness of a different variable y.

However, some domains have more structure in the form of subsumption relationships between elements. In the example type analysis, the fact $\langle x, \texttt{Circle}\rangle$ subsumes the fact $\langle x, \texttt{Shape}\rangle$, since knowing that x points to an object whose type is some subtype of `Circle` implies that its type is also a subtype of `Shape`. Therefore, if the analysis computes, for some program point, the set $\{\langle x, \texttt{Circle}\rangle, \langle x, \texttt{Shape}\rangle\}$, which means that x points to a subtype of `Circle` or that x points to a subtype of `Shape`, then this set provides no additional information compared to the smaller set $\{\langle x, \texttt{Shape}\rangle\}$ that could have been computed; the two sets are equivalent.

Formally, we can define for an arbitrary analysis the partial order $a \leq b$, meaning that a subsumes b (for example, $\langle x, \texttt{Circle}\rangle \leq \langle x, \texttt{Shape}\rangle$). We require all of the dataflow functions to be monotone in the partial order: $a \leq b \implies \text{flow}(a) \leq \text{flow}(b)$. We consider two sets computed by the analysis to be equivalent, written $D_1 \sim D_2$, if every element of each set is subsumed by some element of the other set:

$$\begin{aligned} D_1 \leq D_2 &\iff \forall d_1 \in D_1 \exists d_2 \in D_2 \text{ s.t. } d_1 \leq d_2 \\ D_1 \sim D_2 &\iff D_1 \leq D_2 \wedge D_2 \leq D_1 \end{aligned}$$

The original IFDS algorithm handles analyses in which D has structure correctly but not as efficiently as possible. Because it ignores the subsumption relationship, it compute $\{\langle x, \texttt{Circle}\rangle, \langle x, \texttt{Shape}\rangle\}$ instead of the equivalent smaller set $\{\langle x, \texttt{Shape}\rangle\}$. We have extended the algorithm to use subsumption relationships in D to find smaller equivalent sets. The extension reduces the size not only of the final result, but of the intermediate sets during execution of the algorithm. The performance improvement is cumulative since smaller intermediate sets require less further processing.

The extended algorithm is as precise as the original IFDS algorithm in the sense that if the algorithms compute dataflow facts D_{ext} and D_{orig}, respectively, for a given program point, then $D_{ext} \sim D_{orig}$.

Extending the algorithm to exploit subsumption requires two steps. First, the Propagate function is changed to only add an edge to PathEdge if it does not subsume any already existing edge, as shown in Figure 6.[3] Any edges in PathEdge and in the WorkList subsuming the newly-added edge are redundant and can be removed in line 9.1.

```
    procedure Propagate($\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$)
    begin
9     if $\underline{\nexists \langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle \in \text{PathEdge s.t. } d_2 \le d_3}$ then
          Insert e into PathEdge; Insert e into WorkList; fi
9.1   Remove all edges $\langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle$ s.t. $d_3 \le d_2$ from PathEdge and from WorkList
    end
```

Fig. 6. Extended Propagate Procedure

Second, the worklist is modified so that subsumed elements are processed before subsuming ones. Without an appropriate worklist ordering, the algorithm might do the work of constructing the full sets and only afterwards discover an element that the existing elements subsume, making the existing elements unnecessary. Thus only after all of the work was done would the algorithm discover that the work was not necessary.

To define a suitable worklist ordering, we define an estimate function mapping each element of D to an integer with the property that $d_1 \le d_2 \implies \text{estimate}(d_1) \le \text{estimate}(d_2)$. For all analyses we have encountered, we have found it easy to define such an estimate. For the example type analysis, we use the following estimate: the class `Object` has estimate 0, and the estimate of each other class is one less than the estimate of its superclass. For a given estimate function, the worklist is implemented as a priority queue that makes the algorithm process edges with the highest estimate first.

This ordering heuristic does not completely guarantee that the algorithm will never call Propagate with an edge that makes a previous edge unnecessary, but it does ensure this property in most cases, and works well in practice. Recall that each flow function is monotonic, so that $a \le b \implies \text{flow}(a) \le \text{flow}(b)$. We can be sure to compute flow(b) and flow(a) in the correct order (that is, flow(b) first) by following the ordering heuristic to remove b from the worklist before a. However, at a control flow merge point, it is possible that a and b appear at two different control flow predecessors p, p', which are modelled by different flow functions. There is no guarantee that $a \le b \implies \text{flow}_p(a) \le \text{flow}_{p'}(b)$, so we cannot guarantee that it is more efficient to compute $\text{flow}_{p'}(b)$ before $\text{flow}_p(a)$.

8 Empirical Evaluation

We have performed experiments on the variable type analysis to measure the following:

- How large is the supergraph, and what fraction of it is reachable along valid paths?
- How does taking advantage of subsumption relationships in D reduce the number of dataflow facts that must be processed and the running time of the IFDS algorithm?

We implemented the extended IFDS algorithm and the example type analysis in Scala [14] using Soot [22] as a front-end to parse and convert Java classes into 3-address code and construct a control flow graph (CFG). Both normal Java control flow and control flow due to exceptions was represented by edges in the CFG. We ran the extended

[3] Though it may seem counterintuitive, it is correct to only add elements that do not subsume an existing element, rather than elements not themselves subsumed by an existing element. The interpretation of the PathEdge set is a disjunction of the possible types for each variable: any element in the set is a possible abstraction of runtime behaviour. If a subsumes b, then adding a to a disjunction already containing b does not change the meaning of the disjunction.

Benchmark	Methods	Variables	Instructions	Possible Types
antlr	949	10839	16621	257
bloat	3142	33727	46550	623
chart	9419	91280	129850	2292
fop	13556	131901	185129	3400
hsqldb	768	8004	11552	443
jython	5487	56090	74031	1079
luindex	1306	12519	18131	617
lusearch	1633	14850	21368	676
pmd	3643	33945	49640	998
xalan	786	7708	11084	451

Table 1. Benchmark Characteristics

algorithm on the DaCapo Benchmark Suite, version 2006-10-MR2 [1]. Since most of the benchmarks use reflection, we provided Soot with summaries of uses of reflection obtained by instrumenting the benchmarks using ProBe [11] and *J [5].[4] Statistics about the benchmarks are presented in Table 1. The Methods column shows the number of methods in the part of the call graph analyzed by the IFDS analysis; since the type analysis does not analyze the library, we cut off the call graph at any call into the library. Not analyzing the library is a characteristic of our example analysis, and not a limitation of the IFDS algorithm in general. In earlier work [12], we evaluated two IFDS analyses that successfully analyze the whole program including the standard library. The Variables column shows the number of SSA variables in the analyzed methods. The Instructions column shows the number of instructions after conversion to the intermediate representation presented in Section 3. The Possible Types column shows the number of concrete classes in the benchmark. These are the classes that could appear as the type associated with a variable in the analysis results.

We first measured the size of the complete exploded supergraph. In general, the number of nodes in the exploded graph is given by $|\text{Inst}| \times (|D| + 1)$ where $D = \text{Var} \times \text{Class}$, Var is the set of all variables in the program and Class is the set of all classes. However, when analyzing a given method, only the local variables of that method need to be tracked. Thus a much smaller exploded supergraph can be constructed of size $\sum_{m \in \text{Methods}} |\text{Var}_m| \times |\text{Class}| \times |\text{Inst}_m|$, where $|\text{Var}_m|$ and $|\text{Inst}_m|$ are the numbers of variables and instructions in method m. We measured the size of this smaller, more reasonable exploded supergraph. In addition to the number of nodes, we computed the number of edges in the exploded supergraph. To do this, we applied the flow function to every node of the exploded supergraph and counted the number of outgoing edges. The sizes of the exploded supergraph are shown using diamond shapes in Figure 7. The sizes range from 138 million to 21 billion nodes. On average (geometric mean), each exploded supergraph has 1.16 times as many edges as nodes. The largest exploded supergraphs took over 24 hours to enumerate.

We also measured the sizes of the reachable part of the supergraph that is explored when the IFDS algorithm has been extended with demand supergraph construction. These sizes are shown as horizontal lines in Figure 7. The number of edges in the reach-

[4] We excluded the Eclipse benchmark because it makes such heavy use of reflection that Soot is unable to process it.

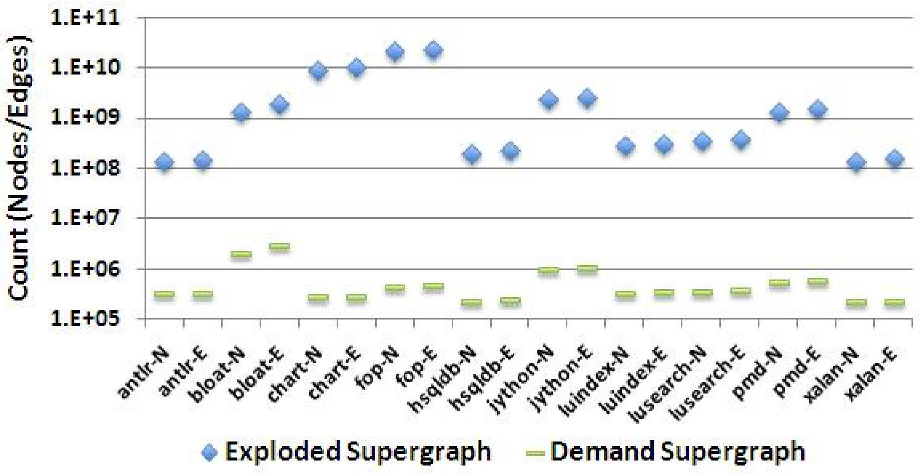


Fig. 7. Number of nodes and edges in the exploded supergraph and its reachable subgraph. The letters N and E after each benchmark name designate nodes and edges, respectively.

able part of the supergraph is 1.09 times the number of nodes. On average (geometric mean), the complete supergraph contains 2081 times as many nodes as the reachable part of the supergraph. Constructing the supergraph on demand rather than exhaustively is key to analyzing benchmarks of this size in reasonable time and memory bounds.

Next, we measured the effect of using subsumption relationships in D to avoid propagating dataflow facts that subsume existing facts. We ran the type analysis three times. In the first run, the subsumption extension from Section 7 was turned off, so all dataflow facts were propagated regardless of their subsumption relationships. In the second run, the subsumption extension was turned on, but the original first-in first-out (FIFO) worklist was used. In the third run, both the subsumption extension and the subsumption-aware worklist ordering from Section 7 were used. For each case, we measured the running time of the analysis and the total number of pairs $\langle v, t \rangle$ computed (i.e. the sum over all instructions of the number of $\langle v, t \rangle$ pairs for that instruction). The results are shown in Table 2. Empty cells in the table indicate that the analysis did not

	Facts (x10^3)		Time (s)		
Benchmark	w/o subs.	w subs.	w/o subs.	w subs., w/o PQ	w subs., w PQ
antlr	546	309	179	44	45
bloat		2037		1544	1518
chart		2817		3377	3197
fop		4408		3247	2847
hsqldb	1758	224	4720	60	60
jython		1015		1225	697
luindex	2900	326	9860	75	70
lusearch	3432	356	9776	78	68
pmd		556		241	211
xalan	1809	218	4813	61	60

Table 2. Effect of taking advantage of subsumption relationships in D.

complete within 10000 seconds of CPU time and 10 GB of memory. The subsumption-extended analysis completed on all of the benchmarks, but the unextended analysis completed on only five benchmarks within these time and memory limits. Columns 2 and 3 show the number of $\langle v, t\rangle$ pairs without and with the subsumption extension (this number is independent of the worklist ordering). Columns 4, 5, and 6 show the running time of the three runs of the analysis. On the five benchmarks on which all algorithms ran to completion, the unextended analysis had to compute 6.3 times as many pairs as the extended analysis, so the unextended analysis took 55 times as long as the extended analysis (geometric mean). In the extended analysis, the subsumption-aware priority queue worklist reduced the running time by 10% (geometric mean over all benchmarks). Extremes were jython, where the reduction was 43%, and antlr, where the running time increased by 2% due to the higher cost of maintaining a priority queue compared to a FIFO list. The subsumption extension presented in Section 7 is very important for the speed of the analysis and for its ability to analyze programs of significant size.

9 Related Work

Sharir and Puneli [19] extended Kildall's framework of intraprocedural dataflow analysis [9, 10] to two frameworks of context-sensitive interprocedural dataflow analysis, which they called the *call-strings* approach and the *functional* approach. The two frameworks compute a merge-over-all-valid-paths solution, where a valid path is one in which procedure calls and returns are correctly matched. The call-strings approach treats calls and returns from a procedure like all other control flow but restricts propagation to valid paths by tagging propagated dataflow facts with a call string (an abstraction of the active call stack). In the functional approach, the effects of each procedure are summarized by a summary function $f_p : \mathcal{D} \to \mathcal{D}$, where $\mathcal{D}$ is the dataflow analysis domain. The summary function is then used at each call site of the procedure to model the effect of the call. The key operation in the functional approach is function composition. For example, to compute the summary function f_r of a caller procedure that contains a call site to a callee procedure, the summary function f_e of the callee procedure must be composed with functions representing the intraprocedural effects of the caller procedure. Although the functional approach has the potential to be more precise and more efficient than the call strings approach, a key challenge is devising efficient representations of the summary functions that are amenable to function composition.

The IFDS framework [15] provides such an efficient representation of summary functions for the functional approach, as discussed in Section 2. When the dataflow domain is $\mathcal{P}(D)$ for a finite set D and all of the dataflow functions are distributive, they can be compactly represented using bipartite graphs with $O(D)$ nodes. Function composition can be computed efficiently in this representation, and the composition of distributive functions is also distributive. Thus the IFDS algorithm makes the functional approach practical for the class of dataflow analyses satisfying these restrictions. The IFDS algorithm has been used to solve both locally separable problems such as reaching definitions, available expressions and live variables, and non-locally-separable problems such as uninitialized variables and copy-constant propagation.

The IDE [18] algorithm generalizes IFDS to a wider class of dataflow analyses. Whereas in IFDS, the dataflow facts are elements of $\mathcal{P}(D)$, the IDE algorithm allows dataflow facts that are maps drawn from $D \to L$, where D is a finite set and L is a finite-height semi-lattice.[5] The IDE algorithm has been used to express copy-constant propagation and linear constant propagation [18]. The IDE literature calls elements of $D \rightarrow L$ environments, so the flow functions that are composed in the algorithm are environment transformers drawn from $(D \rightarrow L) \rightarrow (D \rightarrow L)$. Provided these transformers are distributive, they can be represented efficiently using graphs similar to those used in the IFDS algorithm, with additional labels on the edges of the graph describing the effect of the edge on elements of L. Whereas the IFDS problem computes reachability along valid paths, the IDE algorithm additionally evaluates functions $L \rightarrow L$ along those paths. The overall structure of both algorithms is very similar, however. All of the extensions presented in this paper are equally applicable to the IDE algorithm as well as to the IFDS algorithm. We have implemented the extensions in both algorithms.

Demand-driven variations of the IFDS and IDE algorithms have been thoroughly studied [3, 4, 8, 16, 18]. These algorithms differ from the exhaustive algorithms in that rather than computing all nodes reachable from the start node, they determine whether a given node n is reachable. These algorithms can be faster when only a small number of nodes are queried. The algorithms work by exploring reverse paths along the supergraph from the given node n, by evaluating inverses of the dataflow functions. The demand-driven computation of reachability implemented by these algorithms is distinct from and complementary to the demand-driven exploded supergraph construction that we presented in Section 4. The purpose of demand supergraph construction is to avoid constructing the whole supergraph, which may be much larger than its reachable subgraph; the demand-driven reachability algorithms do require the whole exploded supergraph to be constructed ahead of time. Although our extended IFDS algorithm constructs the exploded supergraph on demand, it then exhaustively computes all nodes reachable along valid paths, rather than answering reachability queries for specific notes. An interesting direction for future work would be to combine demand supergraph construction with demand-driven reachability queries. Such an algorithm appears to be challenging to design and to tune, however. The key difficulty that we had to overcome in constructing the exploded supergraph on demand was the need, on line 23 of the original IFDS algorithm, to evaluate the inverse of the dataflow function. The demand-driven supergraph reachability algorithms require much more evaluation of inverse dataflow functions.

Others have noticed limitations of the original IFDS algorithm, and mention implementing extensions similar to some of those that we have presented here [6,7,17,20,23]. Fink et al. [6, 7] used the IFDS algorithm to verify typestate properties of objects. To verify that an object respects a temporal property, they build precise abstractions of the objects in the program and aliasing between them. The analysis computes an object abstraction containing sets of access paths that must or must-not reference an object. This abstraction is computed using the IFDS algorithm with extensions for exceptional control flow and polymorphic dispatch. Though their presentation focuses on the typestate analysis rather than specifics of their extensions to the IFDS algorithm, their implementation depends on constructing the exploded supergraph on demand, providing call-site information to return flow functions, and exploiting subsumption between elements of D. Shoham et al. [20] apply the infrastructure of Fink et al. [6, 7], along with its IFDS extensions, to statically extract finite-state automata of sequences of API calls.

[5] The domain $\mathcal{P}(D)$ is isomorphic to $D \rightarrow L$ if L is chosen to be the two-point lattice.

Some shape analyses that have been implemented as instances of the IFDS algorithm construct the supergraph on demand for scalability. Rinetzky et al. [17] present an efficient shape analysis for the class of cutpoint-free programs, in which at each procedure call, the subgraph of the heap reachable in the callee can only be reached in the caller through arguments of the call. Yang et al. [23] present a different shape analysis that works for general programs. Both of these analyses are instances of the IFDS algorithm, and both implementations construct only the reachable part of the supergraph.

Several of the analyses just mentioned [6, 7, 17, 20, 23] use partial joins, an extension similar to subsumption in the analysis domain D that we discussed in Section 7. Whereas a partial join enables the analysis designer to sacrifice precision for efficiency, exploiting subsumption does not change analysis precision. A partial join may make the analysis output depend on the order of exploration; exploiting subsumption does not. A partial join operator $\overset{\dagger}{\sqcup}$ is a partial function $\overset{\dagger}{\sqcup} : D \times D \not\twoheadrightarrow D$ with the property that if $a \overset{\dagger}{\sqcup} b = d$, then each of a and b subsume d. Whenever the partial join IFDS algorithm encounters both a and b in a given set, it replaces them with d, reducing the size of the set. This operation is sound, since if each of a and b subsume d, then so does their disjunction. However, it may reduce precision. For example, if we also define $a \overset{\dagger}{\sqcup} c = d$, it becomes impossible for the analysis to distinguish $\{a, b\}$ from $\{a, c\}$, even though neither set subsumes the other (i.e. $\{a, b\} \not\succ \{a, c\}$). Our subsumption extension can be implemented using the following definition of a partial join: if $a \leq b$, then $a \overset{\dagger}{\sqcup} b = b \overset{\dagger}{\sqcup} a = b$, else $a \overset{\dagger}{\sqcup} b$ is undefined.

Our previous work [12] on verifying temporal properties of groups of interacting objects also uses the IFDS and IDE algorithms. Verifying typestate-like properties of multiple objects requires two separate abstractions and analyses: an alias-set abstraction to track the objects in the program and a second abstraction of the typestate of groups of objects. We used the IFDS algorithm to compute these abstractions. We used the IDE algorithm to compute the set of events that might trigger a violation of a temporal property. In later work [13] we improved the alias-set analysis using properties of programs in SSA form. Our extension to the IFDS algorithm to precisely handle ϕ instructions, as presented in Section 6, was essential to obtaining precise alias set information.

10 Conclusions

We presented four extensions to the IFDS algorithm that make it applicable to a wider class of interprocedural dataflow analysis problems, in particular analyses of objects and pointers. The extended algorithm does not require an exploded supergraph as input, but builds it on demand, only for those dataflow facts for which it is actually needed. The extended algorithm provides caller-side context information from before a procedure call to the flow function that maps callee-side state back to the caller after the call. The extended algorithm analyzes programs in SSA form as precisely as programs not in SSA form. The extended algorithm takes advantage of structure in the dataflow analysis domain to significantly speed up analyses exhibiting such structure. We illustrated our extensions on a variation of variable type analysis, and we have applied them to more complicated analyses including alias set analysis [13] and multi-object typestate analysis [12]. The extensions apply not only to the IFDS algorithm but also to the more general IDE algorithm.

References

1. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *OOPSLA '06*, pages 169–190, 2006.
2. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. *POPL '89*, pages 25–35, 1989.
3. E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. *POPL '95*, pages 37–48, 1995.
4. E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.*, 19(6):992–1030, 1997.
5. B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, June 2004.
6. S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ISSTA'06*, pages 133–144, 2006.
7. S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–34, 2008.
8. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. *SIGSOFT FSE '95*, pages 104–115, 1995.
9. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.
10. G. A. Kildall. A unified approach to global program optimization. *POPL '73*, pages 194–206, 1973.
11. O. Lhoták. Comparing call graphs. *PASTE '07*, pages 37–42, 2007.
12. N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. *OOPSLA '08*, pages 347–366, 2008.
13. N. A. Naeem and O. Lhoták. Efficient alias set analysis using SSA form. *ISMM '09*, pages 79–88, 2009.
14. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, 2008.
15. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. *POPL '95*, pages 49–61, 1995.
16. T. W. Reps. Solving demand versions of interprocedural analysis problems. *CC'94*, pages 389–403, 1994.
17. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. *SAS 2005*, pages 284–302, 2005.
18. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.
19. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.
20. S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *ISSTA '07*, pages 174–184, 2007.
21. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. *OOPSLA'00*, pages 264–280, 2000.
22. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? *CC'00*, pages 18–34, 2000.
23. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. *CAV '08*, pages 385–398, 2008.