A Correspondence between Two Approaches to Interprocedural Analysis in the Presence of Join

Ravi Mangal[1], Mayur Naik[1], and Hongseok Yang[2]

[1] Georgia Institute of Technology
[2] University of Oxford

Abstract. Many interprocedural static analyses perform a lossy join for reasons of termination or efficiency. We study the relationship between two predominant approaches to interprocedural analysis, the summary-based (or functional) approach and the call-strings (or k-CFA) approach, in the presence of a lossy join. Despite the use of radically different ways to distinguish procedure contexts by these two approaches, we prove that post-processing their results using a form of garbage collection renders them equivalent. Our result extends the classic result by Sharir and Pnueli that showed the equivalence between these two approaches in the setting of distributive analysis, wherein the join is lossless.

We also empirically compare these two approaches by applying them to a pointer analysis that performs a lossy join. Our experiments on ten Java programs of size 400K–900K bytecodes show that the summary-based approach outperforms an optimized implementation of the k-CFA approach: the k-CFA implementation does not scale beyond k=2, while the summary-based approach proves up to 46% more pointer analysis client queries than 2-CFA. The summary-based approach thus enables, via our equivalence result, to measure the precision of k-CFA with unbounded k, for the class of interprocedural analyses that perform a lossy join.

1 Introduction

Two dominant approaches to interprocedural static analysis are the summary-based approach and the call-strings approach. Both approaches aim to analyze each procedure precisely by distinguishing calling contexts of a certain kind. But they differ radically in the kind of contexts used: the summary-based (or functional) approach uses input abstract states whereas the call-strings (or k-CFA) approach uses sequences of calls that represent call stacks.

Sharir and Pnueli [SP81] showed that, in the case of a finite, distributive analysis, the summary-based approach is equivalent to the unbounded call-strings approach (hereafter called ∞-CFA). In this case, both these approaches maintain at most one abstract state at each program point under a given context of its containing procedure, applying a join operation to combine different abstract states at each program point into a single state. The distributivity condition ensures that this join is *lossless*. As a result, both approaches compute the precise meet-over-all-valid-paths (MVP) solution, and are thus equivalent.

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 513–533, 2014.

Many useful static analyses using the summary-based approach, however, lack distributivity. They too use a join, in order to maintain at most one abstract state at each program point under a given context, and thereby scale to large programs (e.g., [FYD$^+$08]). But in this non-distributive case, the join is *lossy*, leading such analyses to compute a solution less precise than the MVP solution.

We study the relationship between the summary-based and call-strings approaches, in the presence of a lossy join. Our main result is that these two approaches are equivalent in precision despite their use of radically different ways to distinguish procedure contexts. This result yields both theoretical and practical insights. The theoretical insight includes two new proof techniques. The first is a form of garbage collection on the results computed by the non-distributive summary-based approach. This garbage collection removes entries of procedure summaries that are used during analysis but not in the final analysis results. It provides a natural way for connecting the results of the summary-based approach with those of ∞-CFA. The other is a new technique for proving that a fixpoint of a non-monotone function is approximated by a pre-fixpoint of the function. Standard proof techniques do not apply because of non-monotonicity, but such an approximation result is needed in our case because non-distributive summary-based analyses use non-monotone transfer functions.

On the practical side, our equivalence result provides, for the class of non-distributive interprocedural analyses, a feasible approach to determine how precise k-CFA can get using arbitrary k. This feasible approach is the summary-based one, which scales much better than k-CFA. State-of-the-art algorithms for k-CFA do not scale to beyond small values of k, as the number of call-string contexts in which they analyze procedures grows exponentially with k. As a concrete example, we compare the performance of the summary-based approach to an optimized BDD-based implementation of k-CFA for a non-distributive pointer analysis for object-oriented programs. On ten Java programs each of size 400K-900K bytecodes from the DaCapo benchmark suite, we find that the k-CFA implementation does not scale beyond $k=2$, and even for $k=2$, it computes 4X–7X more contexts per benchmark than the summary-based approach. Furthermore, for three clients of the pointer analysis—downcast safety, call graph reachability, and monomorphic call inference—the summary-based approach proves up to 46% more client queries per benchmark than 2-CFA, providing an upper bound on the number of queries that is provable by k-CFA using arbitrary k.

2 Example

We illustrate various interprocedural approaches by means of a pointer analysis on the Java program in Figure 1. All the approaches infer points-to information—aliasing relationships among program variables and heap objects—but differ in their treatment of methods. We illustrate five key aspects of these approaches: (i) 0-CFA produces imprecise results; (ii) using k-CFA with $k > 0$ helps to address this imprecision but hurts scalability; (iii) summary-based analysis (hereafter called SBA) causes no loss in precision compared to k-CFA; (iv) the lossy join

```
class A {}

class B {}

class Container {
    Object holder;
    Container() { holder = null; }
    void add(Object x) {
      if (x.equals(holder)) return;
      holder = x;
    }
    bool isEmpty() {
      return (holder==null);
    }
}
```

```
class C {
    static Container foo() {
h1:   Container s1 = new Container();
h2:   A a = new A();
i1:   s1.add(a);
      return s1;
    }
    static Container bar() {
h3:   Container s2 = new Container();
h4:   B b = new B();
i2:   s2.add(b);
      return s2;
    }
    static void taz(Container s){...}
    static void main() {
      Container s=(*) ? foo() : bar();
j1:   // join point
i3:   s.isEmpty();
i4:   s.isEmpty();
i5:   taz(s);
    }
}
```

Fig. 1. Example Java program

operation in SBA allows analyzing methods in fewer contexts and thereby improves scalability; and (v) SBA can merge multiple k-CFA contexts of a method into a single SBA context which also improves scalability.

We start with 0-CFA which treats method calls in a context insensitive manner. This means that the analysis does not differentiate different call sites to a method, and merges all the abstract states from these call sites into a single input. For instance, consider the program in Figure 1, where the `main()` method calls either `foo()` or `bar()`, creates a container object s containing an `A` or `B` object, and operates on this container s by calling `isEmpty()` and `taz()`. When the pointer analysis based on 0-CFA is applied to `main`, it imprecisely concludes that the returned container from `foo()` may contain an `A` or `B` object, instead of the true case of containing only an `A` object. Another imprecise conclusion is what we call *call graph reachability.* The analysis infers that at one point of execution, the call stack may contain both `foo()` and `B::equals()`, the second on top of the first, i.e., `B::equals()` is reachable from `foo()`. Note that this reachability never materializes during the execution of the program. The main source of both kinds of imprecision is that 0-CFA does not differentiate between the calls to `add()` from **i1** in `foo()` and **i2** in `bar()`. It merges the abstract states from both call sites and analyzes `add()` under the assumption $[x \to \{h_2, h_4\}]$, which means that x points to a heap object allocated at h_2 or h_4, so the object

x has type `A` or `B`. Note that once this assumption is made, the analysis cannot avoid the two kinds of imprecision discussed above.

One way to resolve 0-CFA's imprecision is to use an analysis based on k-CFA with $k > 0$, which analyzes method calls separately if the call stacks at these call sites store sufficiently different sequences of call sites. For instance, the pointer analysis based on 1-CFA analyzes a method multiple times, once for each of its call sites. Hence, when it is applied to our example, it differentiates two call sites to `add()` (i.e., **i1** and **i2**), and analyzes `add()` twice, once for the call site **i1** with the assumption $[x \to \{h_2\}]$ on the parameter x, and again for the call site **i2** with the assumption $[x \to \{h_4\}]$. This differentiation enables the analysis to infer that the returned container from `foo()` contains objects of the type `A` only, and also that `B::equals()` is not reachable from `foo()`. In other words, both kinds of imprecision of 0-CFA are eliminated with 1-CFA.

An alternative solution to the imprecision issue is to use SBA. Unlike k-CFA, SBA does not distinguish contexts based on sequences of call sites stored in the call stack. Instead, it decides that two calling contexts differ when the abstract states at call sites are different. SBA re-analyzes a method in a calling context only if it has not seen the abstract state τ of this context before. In Figure 1, the abstract states at call sites **i1** and **i2** are, respectively, $[s_1 \to \{h_1\}, a \to \{h_2\}]$ and $[s_2 \to \{h_3\}, b \to \{h_4\}]$, which become the following input abstract states to `add()` after the actual parameters are replaced with the formal parameters; $[\mathit{this} \to \{h_1\}, x \to \{h_2\}]$ and $[\mathit{this} \to \{h_3\}, x \to \{h_4\}]$. Since these inputs are different, SBA analyzes method `add()` separately for the calls from **i1** and **i2**, and reaches the same conclusion about the return value of `foo()` and call graph reachability as that of 1-CFA described previously. This agreement in analysis results is not an accident. We prove in Section 3 that SBA's results always coincide with those of ∞-CFA, a version of k-CFA that does not put a bound on the length of call-strings.

An important feature of SBA is that at every control-flow join point in a program, incoming abstract states to this point are combined to a single abstract state via a lossy join operator (if they all originate from the same input abstract state to the current method). This greatly helps the scalability of SBA, because it leads to fewer distinct abstract states at call sites and reduces the number of times that each method should be analyzed. For instance, when SBA analyzes the program in Figure 1, it encounters two incoming abstract states at the join point **j1**, $\tau_1 = [s \to \{h_1\}]$ from the true branch and $\tau_2 = [s \to \{h_3\}]$ from the false branch. The analysis combines τ_1 and τ_2 using a lossy join operator, and results in $\tau' = [s \to \{h_1, h_3\}]$. As a result, at the subsequent call site **i5**, the analysis has only one input abstract state τ', instead of two (i.e., τ_1 and τ_2), and it analyzes the method `taz()` only once.

Using a lossy join operator differentiates SBA from the well-known distributive summary-based analysis [RHS95, SP81], which uses a lossless join. If such an analysis were applied to our program, it would collect τ_1, τ_2 as the set $\{\tau_1, \tau_2\}$ at the join point **j1**, and analyze the call to `taz()` twice. As a result, the coincidence between the results of SBA and ∞-CFA does not follow from what

$$\begin{array}{rrcl}
\text{(method)} & m & \in & \mathbf{M} = \{\, m_{main}, \ldots \,\} \\
\text{(atomic command)} & a & \in & \mathbf{A} \\
\text{(method call)} & i & \in & \mathbf{I} \\
\text{(statement)} & s & \in & \mathbf{S} \triangleq (\mathbf{A} \cup \mathbf{I}) \\
\text{(CFG node)} & n & \in & \mathbf{N} \\
\text{(CFG edge)} & e & \in & \mathbf{E} \subseteq \mathbf{N} \times \mathbf{S} \times \mathbf{N} \\
 & e & \triangleq & \langle n_1, s, n_2 \rangle \\
 & p & \in & \mathbf{P} \triangleq (\mathbf{N} \cup \mathbf{E})
\end{array}
\qquad
\begin{array}{rl}
\mathit{origin}(\langle n_1, s, n_2 \rangle) \triangleq & n_1 \\
\mathit{stmt}(\langle n_1, s, n_2 \rangle) \triangleq & s \\
\mathit{target}(\langle n_1, s, n_2 \rangle) \triangleq & n_2 \\
\mathit{callEdge}(\langle n_1, a, n_2 \rangle) \triangleq & \mathit{false} \\
\mathit{callEdge}(\langle n_1, i, n_2 \rangle) \triangleq & \mathit{true} \\
\text{(method of node/edge) } \mathit{method} \in & \mathbf{P} \to \mathbf{M} \\
\text{(entry node of method) } \mathit{entry} \in & \mathbf{M} \to \mathbf{N} \\
\text{(exit node of method) } \mathit{exit} \in & \mathbf{M} \to \mathbf{N}
\end{array}$$

Fig. 2. Notation for interprocedural control flow graphs

was established previously by Sharir and Pnueli. In fact, proving it requires new proof techniques, as we explain in Section 3.

According to our experiments reported in Section 5, SBA scales better than k-CFA for high k values. This is because SBA usually distinguishes calling contexts of a method less than k-CFA, and re-analyzes the method less often than k-CFA. Concretely, a method may be invoked multiple times with call stacks storing different sequences of call sites but with the same abstract state. In this case, k-CFA re-analyzes the method for each call sequence in the stack, but SBA analyzes the method only once and reuses the computed summary for all the invocations of this method. In effect, SBA merges multiple k-CFA contexts into a single SBA context in this case. This phenomenon can be seen in Figure 1 at the two calls to `isEmpty()` in **i3** and **i4**. Since these call sites are different, `isEmpty()` would be analyzed twice by k-CFA with $k \geq 1$. However, the abstract state at both of the call sites is the same $[s \to \{h_1, h_3\}]$. Hence, SBA analyzes the method only once and reuses the computed summary for the second call.

3 Formal Description and Correspondence Theorem

This section formalizes an unbounded k-CFA and a summary-based interprocedural analysis. The former is an idealization of usual k-CFA that does not put a bound on the length of tracked call strings (i.e., sequences of call sites in the call stack), and records analysis results separately for each call string. To emphasize the absence of bound, we call this analysis ∞-CFA. The summary-based analysis is a non-distributive variant of the standard summary-based approach for distributive (and disjunctive) analyses [RHS95]. It treats join points approximately using a lossy join operator, unlike the standard approach, and trades precision for performance. The main result of the section is that the summary-based analysis has the same precision as ∞-CFA, despite the lossy join.

3.1 Interprocedural Control Flow Graph

In our formalism, we assume that programs are specified in terms of interprocedural control flow graphs $\mathcal{G} = (\mathbf{M}, \mathbf{A}, \mathbf{I}, \mathbf{N}, \mathbf{E}, \mathit{method}, \mathit{entry}, \mathit{exit})$ in Figure 2. Set $\mathbf{M}$ consists of method names in a program, and $\mathbf{A}$ and $\mathbf{I}$ specify available

$$
\begin{array}{rlll}
\text{(abstract state)} & \tau \in \mathbf{\Gamma} = \{\, \tau_{init}, \dots \,\} & & \\
\text{(lattice operations)} & \bigsqcup, \bigsqcap \in \mathcal{P}(\mathbf{\Gamma}) \to \mathbf{\Gamma} & \bot, \top \in \mathbf{\Gamma} & \sqsubseteq \subseteq \mathbf{\Gamma} \times \mathbf{\Gamma} \\
\text{(transfer functions)} & [\![a]\!] \in \mathbf{\Gamma} \to \mathbf{\Gamma} & & \\
\text{(targets of call)} & \mathit{calls}(s, \tau) \in \mathcal{P}(\mathbf{M}) & & \\
\text{(call string)} & \pi \in \mathbf{\Pi} \triangleq \bigcup_{n \geq 0} (\mathbf{M} \cup \mathbf{E})^n & & \\
\text{(∞-CFA annotation)} & \kappa \in \mathbf{A}_{\mathsf{cfa}} = (\mathbf{P} \times \mathbf{\Pi}) \to \mathbf{\Gamma} & & \\
\text{(SBA annotation)} & \sigma \in \mathbf{A}_{\mathsf{sba}} = (\mathbf{P} \times \mathbf{\Gamma}) \to \mathbf{\Gamma} & &
\end{array}
$$

Fig. 3. Analysis domains and transfer functions

$$
F_{\mathsf{cfa}}(\kappa)(n, \pi) = \begin{cases}
\bigsqcup \{\, \kappa(e, \pi) \mid n = \mathit{target}(e) \,\} & \text{if } \nexists m : \ n = \mathit{entry}(m) \\
\bigsqcup \{\, \tau \mid \exists e, \pi_1 : \mathit{callEdge}(e) \wedge \pi = m \oplus e \oplus \pi_1 & \text{if } n = \mathit{entry}(m) \\
\quad \wedge\, \tau = \kappa(\mathit{origin}(e), \pi_1) \wedge m \in \mathit{calls}(\mathit{stmt}(e), \tau) \,\} &
\end{cases}
$$

$$
F_{\mathsf{cfa}}(\kappa)(e, \pi) = \begin{cases}
[\![\mathit{stmt}(e)]\!](\kappa(\mathit{origin}(e), \pi)) & \text{if } \neg \mathit{callEdge}(e) \\
\bigsqcup \{\, \tau \mid \exists \tau_1, m : \tau_1 = \kappa(\mathit{origin}(e), \pi) & \text{if } \mathit{callEdge}(e) \\
\quad \wedge\, m \in \mathit{calls}(\mathit{stmt}(e), \tau_1) \wedge \tau = \kappa(\mathit{exit}(m), m \oplus e \oplus \pi) \,\} &
\end{cases}
$$

Fig. 4. Transfer function F_{cfa} on ∞-CFA annotations

atomic commands and method call instructions. Sets $\mathbf{N}$ and $\mathbf{E}$ determine nodes and intraprocedural edges of a control flow graph. Each node in this graph belongs to a method given by the function *method*. The functions *entry* and *exit* decide the entry and exit nodes of each method. The figure also shows defined entities—*origin*, *stmt*, *target*, and *callEdge*, which can be used to obtain components of an edge and to decide the type of the edge. We assume all the five sets in a control flow graph are finite.

Our control flow graphs are required to satisfy well-formedness conditions. First, $m_{main} \in \mathbf{M}$. Second, for all $m \in \mathbf{M}$ and $e \in \mathbf{E}$,

$$
\begin{array}{l}
\mathit{entry}(m) \neq \mathit{exit}(m) \wedge (\mathit{method} \circ \mathit{entry})(m) = (\mathit{method} \circ \mathit{exit})(m) = m \wedge \\
(\mathit{method} \circ \mathit{target})(e) = (\mathit{method} \circ \mathit{origin})(e) = \mathit{method}(e).
\end{array}
$$

The first conjunct means that the entry node and the exit node of a method are different, the second says that *entry* and *exit* pick nodes belonging to their argument method, and the last conjunct states that an edge and its source and target nodes are in the same method.

3.2 Formal Description of Analyses

Both ∞-CFA and the summary-based analysis assume $(\mathbf{\Gamma}, \tau_{init}, [\![\_]\!], \mathit{calls})$ in Figure 3, which are needed for performing an intraprocedural analysis as well as processing dynamically dispatched method calls. Component $\mathbf{\Gamma}$ is a finite complete lattice, and consists of abstract states used by the analysis. The next $\tau_{init} \in \mathbf{\Gamma}$ is an initial abstract state to the root method m_{main}, and $[\![a]\!]$ represents abstract transfer functions for atomic commands a. The final component *calls* takes a

$$F_{\mathsf{sba}}(\sigma)(n,\tau) = \begin{cases} \bigsqcup\{\,\sigma(e,\tau) \mid n = \mathit{target}(e)\,\} & \text{if } \nexists m : n = \mathit{entry}(m) \\ \bigsqcup\{\,\tau \mid \exists e, \tau_1 : \mathit{callEdge}(e) \wedge \tau{=}\sigma(\mathit{origin}(e),\tau_1) & \text{if } n = \mathit{entry}(m) \\ \quad \wedge\, m \in \mathit{calls}(\mathit{stmt}(e),\tau)\,\} & \end{cases}$$

$$F_{\mathsf{sba}}(\sigma)(e,\tau) = \begin{cases} [\![\mathit{stmt}(e)]\!](\sigma(\mathit{origin}(e),\tau)) & \text{if } \neg\mathit{callEdge}(e) \\ \bigsqcup\{\,\tau' \mid \exists \tau_1, m : \tau_1 = \sigma(\mathit{origin}(e),\tau) & \text{if } \mathit{callEdge}(e) \\ \quad \wedge\, m \in \mathit{calls}(\mathit{stmt}(e),\tau_1) \wedge \tau' = \sigma(\mathit{exit}(m),\tau_1)\,\} & \end{cases}$$

Fig. 5. Transfer function F_{sba} on SBA annotations

pair (s,τ), and conservatively estimates target methods of a call s in (concrete) states described by τ, if s is a method call. Otherwise, it returns the empty set.

We require that the components of the analysis satisfy the following properties: (i) $\tau_{init} \neq \bot$; (ii) $\mathit{calls}(s, \_)$ and $[\![a]\!]$ are monotone with respect to the order in Γ or the subset order[1]; (iii) $\mathit{calls}(s,\bot) = \emptyset$ and $[\![a]\!](\bot) = \bot$; (iv) for all s and τ, $m_{main} \notin \mathit{calls}(s,\tau)$, and if s is not a method call, $\mathit{calls}(s,\tau) = \emptyset$.

∞-CFA Analysis. The ∞-CFA analysis is an interprocedural analysis that uses call strings of arbitrary length as calling contexts and analyzes a method separately for each call string. If a reader is familiar with k-CFA, we suggest to view ∞-CFA as the limit of k-CFA with k tending towards ∞. Indeed, ∞-CFA computes a result that is as precise as any k-CFA analysis.

The ∞-CFA works by repeatedly updating a map $\kappa \in \mathbf{A}_{\mathsf{cfa}} = (\mathbf{P} \times \mathbf{\Pi}) \to \mathbf{\Gamma}$, called **$\infty$-CFA annotation**. The first argument p to κ is a program node or an edge, and the second π a call string defined in Figure 3, which is a finite sequence of method names and edges. A typical call string is $m_2 \oplus e_2 \oplus m_1 \oplus e_1 \oplus m_{main}$. It represents a chain of calls $m_{main} \to m_1 \to m_2$, where m_1 is called by the edge e_1 and m_2 by e_2. The function κ maps such p and π to an abstract state τ, the current estimation of concrete states reaching p with π on the call stack.

We order ∞-CFA annotations pointwise: $\kappa \sqsubseteq \kappa' \iff \forall p, \pi : \kappa(p,\pi) \sqsubseteq \kappa'(p,\pi)$. This order makes the set of ∞-CFA annotations a complete lattice. The ∞-CFA analysis computes a fixpoint on ∞-CFA annotations:

$$\kappa_{\mathsf{cfa}} = \mathsf{leastFix}\ \lambda\kappa.\, (\kappa_I \sqcup F_{\mathsf{cfa}}(\kappa)). \tag{1}$$

Here κ_I is the initial ∞-CFA annotation, and models our assumption that a given program starts at m_{main} in a state satisfying τ_{init}:

$$\kappa_I(p,\pi) = \text{if } ((p,\pi) = (\mathit{entry}(m_{main}), m_{main})) \text{ then } \tau_{init} \text{ else } \bot.$$

Function F_{cfa} is the so called transfer function, and overapproximates one-step execution of atomic commands and method calls in a given program. Figure 4 gives the definition of F_{cfa}. Although this definition looks complicated, it comes from a simple principle: F_{cfa} updates its input κ simply by propagating abstract states in κ to appropriate next nodes or edges, while occasionally pushing or popping call sites and invoked methods in the tracked call string.

[1] This means $\forall \tau, \tau' \in \Gamma : \tau \sqsubseteq \tau' \implies (\mathit{calls}(s,\tau) \subseteq \mathit{calls}(s,\tau') \wedge [\![a]\!](\tau) \sqsubseteq [\![a]\!](\tau'))$.

We make two final remarks on ∞-CFA. First, F_{cfa} is monotone with respect to our order on ∞-CFA annotations. This ensures that the least fixpoint in (1) exists. Although the monotonicity is an expected property, we emphasize it here because the transfer function of our next interprocedural analysis SBA is not monotone with respect to a natural order on analysis results. Second, the domain of ∞-CFA annotations is infinite, so a finite number of iterations might be insufficient for reaching the least fixpoint in (1). We are not concerned with this potential non-computability, because we use ∞-CFA only as a device for comparing the precision of SBA in the next subsection with that of k-CFA.

Summary-Based Analysis. The summary-based analysis SBA is another approach to analyze methods context-sensitively. Just like ∞-CFA, it keeps separate analysis results for different calling contexts, but differs from ∞-CFA in that it uses input abstract states to methods as contexts, instead of call strings.

The main data structures of SBA are **SBA annotations** σ:

$$\sigma \in \mathbf{A}_{\mathsf{sba}} = (\mathbf{P} \times \mathbf{\Gamma}) \to \mathbf{\Gamma}.$$

An SBA annotation σ specifies an abstract state $\sigma(p, \tau)$ at each program point p for each calling context τ. Recall that a calling context here is just an initial abstract state to the current method. SBA annotations are ordered pointwise: $\sigma \sqsubseteq \sigma' \iff \forall p, \tau : \sigma(p, \tau) \sqsubseteq \sigma'(p, \tau)$. With this order, the set of SBA annotations forms a complete lattice. Further, it is finite as $\mathbf{P}$ and $\mathbf{\Gamma}$ are finite.

The summary-based analysis is an iterative algorithm for computing a fixpoint of some function on SBA annotations. It starts with setting the current SBA annotation to σ_I below:

$$\sigma_I(p, \tau) = \text{if } ((p, \tau) = (\mathit{entry}(m_{\mathit{main}}), \tau_{\mathit{init}})) \text{ then } \tau_{\mathit{init}} \text{ else } \bot,$$

which says that only the entry node of m_{main} has the abstract state τ_{init} under the context τ_{init}. Then, it repeatedly updates the current SBA annotation using the transfer function F_{sba} in Figure 5. The function propagates abstract states at all program nodes and edges along interprocedural control-flow edges. In doing so, it approximates one-step execution of every atomic command and method call in a given program. The summary-based analysis does the following fixpoint computation and calculates σ_{sba}:

$$\sigma_{\mathsf{sba}} = \mathsf{fix}^{\sigma_I}\ (\lambda\sigma.\ \sigma \sqcup F_{\mathsf{sba}}(\sigma)). \tag{2}$$

Let $G = (\lambda\sigma.\ \sigma \sqcup F_{\mathsf{sba}}(\sigma))$. Here $(\mathsf{fix}^{\sigma_I}\ G)$ generates the sequence $G^0(\sigma_I)$, $G^1(\sigma_I)$, $G^2(\sigma_I), \ldots$, until it reaches a fixpoint $G^n(\sigma_I)$ such that $G^n(\sigma_I) = G^{n+1}(\sigma_I)$. This fixpoint $G^n(\sigma_I)$ becomes the result σ_{sba} of $\mathsf{fix}^{\sigma_I}\ G$.

Note that fix always reaches a fixpoint in (2). The generated sequence is always increasing because $\sigma \sqsubseteq G(\sigma)$ for every σ. Since the domain of SBA annotations is finite, this increasing sequence should reach a fixpoint. One might wonder why SBA does not use the standard least fixpoint. The reason is that our transfer function F_{sba} is not monotone, so the standard theory for least fixpoints does not apply. This is in contrast to ∞-CFA that has the monotone transfer function.

Non-monotone transfer functions commonly feature in program analyses for numerical properties that use widening operators [Min06, CC92], and the results of these analyses are computed similarly to what we described above (modulo the additional use of a widening operator).

3.3 Correspondence Theorem

The main result of this section is the Correspondence Theorem, which says that ∞-CFA and SBA have the same precision.

Recall that the results of SBA and ∞-CFA are functions of different types: the domain of σ_{sba} is $\mathbf{P} \times \mathbf{\Gamma}$, whereas that of κ_{cfa} is $\mathbf{P} \times \mathbf{\Pi}$. Hence, to connect the results of both analyses, we need a way to relate functions of the first kind with those of the second. For this purpose, we use a particular kind of functions:

Definition 1. *A* **translation function** η *is a map of type* $\mathbf{M} \times \mathbf{\Pi} \to \mathbf{\Gamma}$.

Intuitively, $\eta(m, \pi) = \tau$ expresses that although a call string π and an abstract state τ are different types of calling contexts, we will treat them the same when they are used as contexts for method m.

One important property of a translation function η is that it induces maps between SBA and ∞-CFA annotations:

$$\begin{array}{ll} L(\eta, -) : \mathbf{A}_{\mathsf{sba}} \to \mathbf{A}_{\mathsf{cfa}} & L(\eta, \sigma) = \lambda(p, \pi).\, \sigma(p, \eta(\mathit{method}(p), \pi)), \\ R(\eta, -) : \mathbf{A}_{\mathsf{cfa}} \to \mathbf{A}_{\mathsf{sba}} & R(\eta, \kappa) = \lambda(p, \tau).\, \bigsqcap\{\kappa(p, \pi) \mid \tau \sqsubseteq \eta(\mathit{method}(p), \pi)\}. \end{array}$$

Both L and R use η to convert calling contexts of one type to those of the other. The conversion in $L(\eta, \sigma)$ is as we expect; it calls η to change an input call string π to an input abstract state $\eta(\mathit{method}(p), \pi)$, which is then fed to the given SBA annotation σ. On the other hand, the conversion in $R(\eta, \kappa)$ is unusual, but follows the same principle of using η for translating contexts. Conceptually, it changes an input abstract state τ to a set of call strings π that would be translated to an overapproximation of τ by η (i.e., $\tau \sqsubseteq \eta(\mathit{method}(p), \pi)$), looks up the values of κ at these call strings, and combine the looked-up values by the meet operation. The following lemma relates $L(\eta, -)$ and $R(\eta, -)$:

Lemma 1. *For all σ and κ, if $\sigma \sqsubseteq R(\eta, \kappa)$, then $L(\eta, \sigma) \sqsubseteq \kappa$.*

The definition of a translation function does not impose any condition, and permits multiple possibilities. Hence, a natural question is: what is a good translation function η that would help us to relate the results of the SBA analysis with those of the ∞-CFA analysis? The following lemma suggests one such candidate η_{sba}, which is constructed from the results σ_{sba} of the SBA analysis:

Lemma 2. *There exists a unique translation function $\eta : \mathbf{M} \times \mathbf{\Pi} \to \mathbf{\Gamma}$ such that for all $m \in \mathbf{M}$, $e \in \mathbf{E}$ and $\pi \in \mathbf{\Pi}$,*

$$\begin{array}{l} \eta(m_{main}, m_{main}) = \tau_{init}, \\ \eta(m, m \oplus e \oplus \pi) = \mathit{if}\ (m \in \mathit{calls}(\mathit{stmt}(e), \sigma_{\mathsf{sba}}(\mathit{origin}(e), \eta(\mathit{method}(e), \pi))) \\ \qquad\qquad\qquad\qquad \wedge\ \mathit{callEdge}(e))\ \mathit{then}\ \sigma_{\mathsf{sba}}(\mathit{origin}(e), \eta(\mathit{method}(e), \pi))\ \mathit{else}\, \bot, \\ \eta(m, \pi) = \bot \qquad \textit{(for all the other cases).} \end{array}$$

We denote this translation with η_{sba}.

Intuitively, for each call string π, the translation η_{sba} in the lemma follows the chain of calls in π while tracking corresponding abstract input states stored in σ. When this chasing is over, it finds an input abstract state τ corresponding to the given π. For instance, given a method m_2 and a call string $m_2 \oplus e_2 \oplus m_1 \oplus e_1 \oplus m_{main}$, if all the side conditions in the lemma are met, η_{sba} returns abstract state $\sigma_{\mathsf{sba}}(\mathit{origin}(e_2), \sigma_{\mathsf{sba}}(\mathit{origin}(e_1), \tau_{init}))$. This corresponds to the input abstract state to method m_2 that arises after method calls first at e_1 and then e_2.

Another good intuition is to view η_{sba} as a *garbage collector*. Specifically, for each method m, the set

$$\Gamma_m = \{\eta_{\mathsf{sba}}(m, \pi) \mid \pi \in \mathbf{\Pi}\}. \tag{3}$$

identifies input abstract states for m that contribute to the analysis result σ_{sba} along some call chain from m_{main} to m; every other input abstract state τ for m is garbage even if it was used during the fixpoint computation of σ_{sba} and so $\sigma_{\mathsf{sba}}(\mathit{entry}(m), \tau) \neq \bot$.

Our Correspondence Theorem says that the SBA analysis and the ∞-CFA analysis compute the same result modulo the translation via $L(\eta_{\mathsf{sba}}, -)$.

Theorem 2 (Correspondence). $L(\eta_{\mathsf{sba}}, \sigma_{\mathsf{sba}}) = \kappa_{\mathsf{cfa}}$.

One important consequence of this theorem is that both analyses have the same estimation about reachable concrete states at each program point, if we garbage-collect the SBA's result using η_{sba}:

Corollary 1. *For all $p \in \mathbf{P}$ and $m \in \mathbf{M}$, if $\mathit{method}(p) = m$, then*

$$\{\kappa_{\mathsf{cfa}}(p, \pi) \mid \pi \in \mathbf{\Pi}\} = \{\sigma_{\mathsf{sba}}(p, \tau') \mid \tau' \in \Gamma_m\}, \text{ where } \Gamma_m \text{ is defined by } (3).$$

Overview of Proof of the Correspondence Theorem. Proving the Correspondence Theorem is surprisingly tricky. A simple proof strategy is to show that the relationship in the theorem is maintained by each step of the fixpoint computations of ∞-CFA and SBA, but this strategy does not work. Since ∞-CFA and SBA treat the effects of method calls (i.e., call edges) very differently, the relationship in the theorem is not maintained during fixpoint computations. Further difficulties arise because the SBA analysis uses a non-monotone transfer function F_{sba} and does not necessarily compute the least fixpoint of $\lambda\sigma.\, \sigma_I \sqcup F_{\mathsf{sba}}(\sigma)$—these render standard techniques for reasoning about fixpoints no longer applicable.

In this subsection, we outline our proof of the Correspondence Theorem, and point out proof techniques that we developed to overcome difficulties mentioned above. The full proof is included in the Appendix.

Let $G_{\mathsf{cfa}} = \lambda\kappa.\, \kappa_I \sqcup F_\kappa(\kappa)$ and $G_{\mathsf{sba}} = \lambda\sigma.\, \sigma_I \sqcup F_{\mathsf{sba}}(\sigma)$. Recall that the ∞-CFA analysis computes the least fixpoint of G_{cfa} while the SBA analysis computes some pre-fixpoint of G_{sba} (i.e., $G_{\mathsf{sba}}(\sigma_{\mathsf{sba}}) \sqsubseteq \sigma_{\mathsf{sba}}$) via an iterative process. Our proof consists of the following four main steps.

1. First, we prove that $G_{\mathsf{cfa}}(L(\eta_{\mathsf{sba}}, \sigma_{\mathsf{sba}})) \sqsubseteq L(\eta_{\mathsf{sba}}, \sigma_{\mathsf{sba}})$. That is, $L(\eta_{\mathsf{sba}}, \sigma_{\mathsf{sba}})$ is a pre-fixpoint of G_{cfa}. This implies

$$\kappa_{\mathsf{cfa}} \sqsubseteq L(\eta_{\mathsf{sba}}, \sigma_{\mathsf{sba}}), \tag{4}$$

a half of the conclusion in the Correspondence Theorem. To see this implication, note that the function G_{cfa} is monotone and works on a complete lattice, and the analysis computes the least fixpoint κ_{cfa} of G_{cfa}. According to the standard result, the least fixpoint is also the least pre-fixpoint, so κ_{cfa} is less than or equal to another pre-fixpoint $L(\eta_{\mathsf{sba}}, \sigma_{\mathsf{sba}})$.

2. We next construct another translation, denoted η_{cfa}, this time from the result of the ∞-CFA analysis: $\eta_{\mathsf{cfa}} = \lambda(m, \pi).\, \kappa_{\mathsf{cfa}}(\mathit{entry}(m), \pi)$. Then, we show
$$\sigma_{\mathsf{sba}} \sqsubseteq R(\eta_{\mathsf{cfa}}, \kappa_{\mathsf{cfa}}). \tag{5}$$
The proof of this inequality uses our new technique for verifying that an SBA annotation overapproximates σ_{sba}, a pre-fixpoint of a non-monotone function G_{sba}. We will explain this technique at the end of this subsection.
3. Third, we apply Lemma 1 to the inequality in (5), combine the result of this application with (4), and derive
$$L(\eta_{\mathsf{cfa}}, \sigma_{\mathsf{sba}}) \sqsubseteq \kappa_{\mathsf{cfa}} \sqsubseteq L(\eta_{\mathsf{sba}}, \sigma_{\mathsf{sba}}). \tag{6}$$
4. Finally, using the relationship between σ_{sba} and κ_{cfa} in (6), we show that $\eta_{\mathsf{cfa}} = \eta_{\mathsf{sba}}$. Note that conjoined with the same relationship again, this equality entails $L(\eta_{\mathsf{sba}}, \sigma_{\mathsf{sba}}) = \kappa_{\mathsf{cfa}}$, the claim of the Correspondence theorem.

Before finishing, let us explain a proof technique used in the second step. An SBA annotation σ is **monotone** if $\forall p, \tau, \tau' : \tau \sqsubseteq \tau' \implies \sigma(p, \tau) \sqsubseteq \sigma(p, \tau')$. Our proof technique is summarised in the following lemma:

Lemma 3. *For all SBA annotations σ, if σ is monotone, $G_{\mathsf{sba}}(\sigma) \sqsubseteq \sigma$ and*
$$\forall m : \tau \sqsubseteq \sigma(\mathit{entry}(m), \tau), \tag{7}$$
then $\sigma_{\mathsf{sba}} \sqsubseteq \sigma$.

We remind the reader that if G_{sba} is a monotone function and σ_{sba} is its least fixpoint, we do not need the monotonicity of σ and the condition in (7) in the lemma. In this case, $\sigma_{\mathsf{sba}} \sqsubseteq \sigma$ even without these conditions. This lemma extends this result to the non-monotone case, and identifies additional conditions.

The conclusion of the second step in our overview above is obtained using Lemma 3. In that step, we prove that (1) $R(\eta_{\mathsf{cfa}}, \kappa_{\mathsf{cfa}})$ is *monotone*; (2) it is a pre-fixpoint of G_{sba}; (3) it satisfies the condition in (7). Hence, Lemma 3 applies, and gives $\sigma_{\mathsf{sba}} \sqsubseteq R(\eta_{\mathsf{cfa}}, \kappa_{\mathsf{cfa}})$.

A final comment is that when the abstract domain of a static analysis is infinite, if it is a complete lattice, we can still define SBA similar to our current definition. The only change is that the result of SBA, σ_{sba}, is now defined in terms of the limit of a potentially infinite chain (generated by the application of G_{sba} and the least-upper-bound operator for elements at limit ordinals), instead of a finite chain. This new SBA is not necessarily computable, but we can still ask whether its result coincides with that of ∞-CFA. We believe that the answer is yes: most parts of our proof seem to remain valid for this new SBA, while the remaining parts (notably the proof of Lemma 3) can be modified relatively easily to accommodate this new SBA. This new Coincidence theorem, however, is limited; it does not say anything about analyses with widening.

4 Application to Pointer Analysis

We now show how to apply the summary-based approach to a pointer analysis for object-oriented programs, which also computes the program's call graph.

The input to the analysis is a program in the form of an interprocedural control flow graph (defined in Section 3.1). Figure 6 shows the kinds of statements it considers: atomic commands that create, read, and write pointer locations, via local variables v, global variables (i.e., static fields) g, and object fields (i.e., instance fields) f. We label each allocation site with a unique label h. We elide statements that operate on non-pointer data as they have no effect on our analysis. For brevity we presume that method calls are non-static and have a lone argument, which serves as the receiver, and a lone return result. We use functions *arg* and *ret* to obtain the formal argument and return variable, respectively, of each method. Finally, our analysis exploits type information and uses function

(allocation site)	$h \in \mathbf{H}$	(subtypes)	$sub \in \mathbf{T} \to \mathcal{P}(\mathbf{T})$
(local variable)	$v \in \mathbf{V}$	(class hierarchy analysis)	$cha \in (\mathbf{M} \times \mathbf{T}) \to \mathbf{M}$
(global variable)	$g \in \mathbf{G}$	(method argument)	$arg \in \mathbf{M} \to \mathbf{V}$
(object field)	$f \in \mathbf{F}$	(method result)	$ret \in \mathbf{M} \to \mathbf{V}$
(class type)	$t \in \mathbf{T}$	(abstract contexts)	$\mathbf{\Gamma} = \mathbf{V} \to \mathcal{P}(\mathbf{H})$
(atomic command)	$a ::= v = null \mid$	(points-to of locals)	$\mathsf{ptsV} \in \mathbf{V} \to \mathcal{P}(\mathbf{H})$
$v = new\ h \mid v = (t)\ v' \mid g = v \mid$		(points-to of globals)	$\mathsf{ptsG} \in \mathbf{G} \to \mathcal{P}(\mathbf{H})$
$v = g \mid v.f = v' \mid v' = v.f$		(points-to of fields)	$\mathsf{ptsF} \in (\mathbf{H} \times \mathbf{F}) \to \mathcal{P}(\mathbf{H})$
(method call)	$i ::= v' = v.m()$	(call graph)	$\mathsf{cg} \subseteq (\mathbf{\Gamma} \times \mathbf{E} \times \mathbf{\Gamma} \times \mathbf{M})$
(allocation type)	$type \in \mathbf{H} \to \mathbf{T}$		

Fig. 6. Data for our pointer analysis

$$\llbracket v = null \rrbracket(\mathsf{ptsV}) = \mathsf{ptsV}[v \mapsto \emptyset] \tag{8}$$

$$\llbracket v = new\ h \rrbracket(\mathsf{ptsV}) = \mathsf{ptsV}[v \mapsto \{\ h\ \}] \tag{9}$$

$$\llbracket g = v \rrbracket(\mathsf{ptsV}) = \mathsf{ptsV} \tag{10}$$

$$\llbracket v.f = v' \rrbracket(\mathsf{ptsV}) = \mathsf{ptsV} \tag{11}$$

$$\llbracket v' = (t)\ v \rrbracket(\mathsf{ptsV}) = \mathsf{ptsV}[v' \mapsto \{\ h \in \mathsf{ptsV}(v) \mid type(h) \in sub(t)\ \}] \tag{12}$$

$$\llbracket v = g \rrbracket(\mathsf{ptsV}) = \mathsf{ptsV}[v \mapsto \mathsf{ptsG}(g)] \tag{13}$$

$$\llbracket v' = v.f \rrbracket(\mathsf{ptsV}) = \mathsf{ptsV}[v' \mapsto \bigcup \{\mathsf{ptsF}(h, f) \mid h \in \mathsf{ptsV}(v)\}] \tag{14}$$

$$calls(v' = v.m(), \mathsf{ptsV}) = \{\ cha(m, type(h)) \mid h \in \mathsf{ptsV}(v)\ \} \tag{15}$$

$$\llbracket g = v \rrbracket(\mathsf{ptsG}) = \lambda g'.\ \text{if } (g' = g) \text{ then } (\mathsf{ptsG}(g) \cup \mathsf{ptsV}(v)) \text{ else } \mathsf{ptsG}(g) \tag{16}$$

$$\begin{aligned}\llbracket v.f = v' \rrbracket(\mathsf{ptsF}) = {} & \lambda(h, f').\ \text{if } (h \in \mathsf{ptsV}(v) \wedge f' = f) \\ & \text{then } (\mathsf{ptsF}(h, f') \cup \mathsf{ptsV}(v')) \text{ else } \mathsf{ptsF}(h, f')\end{aligned} \tag{17}$$

Fig. 7. Transfer functions for our pointer analysis

type to obtain the type of objects allocated at each site, function *sub* to find all the subtypes of a type, and function $cha(m, t)$ to obtain the target method of calling method m on a receiver object of run-time type t.

We specify the analysis in terms of the data $(\mathbf{\Gamma}, \tau_{init}, [\![\_]\!], calls)$ in Section 3. Abstract states $\tau \in \mathbf{\Gamma}$ in our analysis are abstract environments ptsV that track points-to sets of locals. Our analysis uses allocation sites for abstract memory locations. Thus, points-to sets are sets of allocation sites. The lattice operations are standard, for instance, the join operation takes the pointwise union of points-to sets: $\bigsqcup \{\mathsf{ptsV}_1, ..., \mathsf{ptsV}_n\} = \lambda v. \bigcup_{i=1}^{n} \mathsf{ptsV}_i(v)$. The second component τ_{init} is the abstract environment $\lambda v.\emptyset$ which initializes all locals to empty points-to sets. The remaining two components $[\![\_]\!]$ and *calls* are shown in Figure 7. We elaborate upon them next. Equations (8)–(14) show the effect of each statement on points-to sets of locals. We explain the most interesting ones. Equation (12) states that cast statement $v' = (t)\ v$ sets the points-to set of local v' after the statement to those allocation sites in the points-to set of v before the statement that are subtypes of t. Equations (13) and (14) are transfer functions for statements that read globals and fields. Since ptsV tracks points-to information only for locals, we use separate data ptsG and ptsF to track points-to information for globals and fields, respectively. These data are updated by transfer functions for statements that write globals and fields, shown in Equations (16) and (17). Since the transfer functions both read and write data ptsG and ptsF, the algorithm for our combined points-to and call graph analysis has an outer loop that calls the SBA algorithm from Section 3 until ptsG and ptsF reach a fixpoint, starting with empty data for them in the initial iteration, $\lambda g.\emptyset$ and $\lambda(h, f).\emptyset$. This outer loop implements a form of the reduced product [CC79] of our flow-sensitive points-to analysis for locals and the flow-insensitive analysis for globals and fields.[2] It is easy to see that the resulting algorithm terminates as $\mathbf{\Gamma}$ is finite.

Finally, in addition to points-to information, our analysis produces a context sensitive call graph, denoted by a set cg containing each tuple (τ_1, e, τ_2, m) such that the call at control-flow edge e in context τ_1 of its containing method may call the target method m in context τ_2. It is straightforward to compute this information by instrumenting the SBA algorithm to add tuple (τ_1, e, τ_2, m) to cg whenever it visits a call site e in context τ_1 and computes a target method m and a target context τ_2.

5 Empirical Evaluation

We evaluated various interprocedural approaches on our pointer analysis using ten Java programs from the DaCapo benchmark suite (`http://dacapobench.org`), shown in Table 1. All experiments were done using Oracle HotSpot JVM 1.6.0 on a Linux machine with 32GB RAM and AMD Opteron 3.0GHz processor. We also measured the precision of these approaches on three different clients of the pointer analysis. We implemented all our approaches and clients using the Chord

[2] We used flow-insensitive analysis for globals and fields to ensure soundness under concurrency—many programs in our experiments are concurrent.

Table 1. Program statistics by flow and context insensitive call graph analysis ($\mathsf{0CFA}_I$)

	brief description	classes		methods		bytecode (KB)		KLOC	
		app	total	app	total	app	total	app	total
antlr	parser/translator generator	109	1,091	873	7,220	81	467	26	224
avrora	microcontroller simulator/analyzer	78	1,062	523	6,905	35	423	16	214
bloat	bytecode optimization/analysis tool	277	1,269	2,651	9,133	195	586	59	258
chart	graph plotting tool and pdf renderer	181	1,756	1,461	11,450	101	778	53	366
hsqldb	SQL relational database engine	189	1,341	2,441	10,223	190	670	96	322
luindex	text indexing tool	193	1,175	1,316	7,741	99	487	38	237
lusearch	text search tool	173	1,157	1,119	7,601	77	477	33	231
pmd	Java source code analyzer	348	1,357	2,590	9,105	186	578	46	247
sunflow	photo-realistic rendering system	165	1,894	1,328	13,356	117	934	25	419
xalan	XSLT processor to transform XML	42	1,036	372	6,772	28	417	9	208

program analysis platform for Java bytecode (`http://jchord.googlecode.com`). We next describe the various approaches and clients.

Interprocedural Approaches. The approaches we evaluated are shown in Table 2. They differ in three aspects: (i) the kind of implementation (tabulation algorithm from [RHS95] called RHS for short vs. BDD); (ii) the degree of call-strings context sensitivity (i.e., the value of k); and (iii) flow sensitive vs. flow insensitive tracking of points-to information for locals. The approach in Section 4 is the most precise one, SBA_S. It is a non-distributive summary-based approach that yields unbounded k-CFA context sensitivity, tracks points-to information of locals flow sensitively, and does heap updates context sensitively. It is implemented using RHS. Doing context sensitive heap updates entails SBA_S calling the tabulation algorithm repeatedly in an outer loop that iterates until points-to information for globals and fields reaches a fixpoint (each iteration of this outer loop itself executes an inner loop—an invocation of the tabulation algorithm—that iterates until points-to information for locals reaches a fixpoint). We confirmed that the non-distributive aspect (i.e., the lossy join) is critical to the performance of our RHS implementation: *it ran out of memory on all our benchmarks without lossy join.* In fact, the lossy join even obviated the need for other optimizations in our RHS implementation, barring only the use of bitsets to represent points-to sets.

It is easy to derive the remaining approaches in Table 2 from SBA_S. $\mathsf{0CFA}_S$ is the context insensitive version of SBA_S. It also uses the RHS implementation and leverages the flow sensitive tracking of points-to information of locals in the tabulation algorithm. We could not scale the RHS implementation to simulate k-CFA for $k > 0$. Hence, our evaluation includes a non-RHS implementation: an optimized BDD-based one that tracks points-to information of locals flow insensitively but allows us to do bounded context sensitive k-CFA for $k > 0$. Even this optimized implementation, however, ran out of memory on all our benchmarks beyond $k = 2$. Nevertheless, using it up to $k = 2$ enables us to gauge the precision and performance of a state-of-the-art bounded k-CFA approach.

Table 2. Interprocedural approaches evaluated in our experiments

	kind of implementation	context sensitivity degree (k)	flow sensitivity for locals?
SBA_S	RHS	∞	yes
$\mathsf{0CFA}_S$	RHS	0	yes
kCFA_I	BDD	0,1,2	no

To summarize, the relative precision of the approaches we evaluate is: $\mathsf{SBA}_S \preceq \mathsf{0CFA}_S \preceq \mathsf{0CFA}_I$ and $\mathsf{SBA}_S \preceq \mathsf{2CFA}_I \preceq \mathsf{1CFA}_I \preceq \mathsf{0CFA}_I$. In particular, the only incomparable pairs are ($\mathsf{0CFA}_S$, $\mathsf{1CFA}_I$) and ($\mathsf{0CFA}_S$, $\mathsf{2CFA}_I$).

Pointer Analysis Clients. We built three clients that use the result of our pointer analysis: downcast safety, call graph reachability, and monomorphic call site inference. The result used by these clients is the context sensitive call graph, $\mathsf{cg} \subseteq (\mathbf{C} \times \mathbf{E} \times \mathbf{C} \times \mathbf{M})$, and context sensitive points-to sets of locals at each program point, $\mathsf{pts} \in (\mathbf{N} \times \mathbf{C}) \to \mathbf{\Gamma}$, where contexts $c \in \mathbf{C}$ are abstract environments (in domain $\mathbf{\Gamma}$) for the SBA_* approaches and bounded call strings (in domain $\mathbf{\Pi}$) for the CFA_* approaches. The above result signatures are the most general, for instance, a context insensitive approach like $\mathsf{0CFA}_I$ may use a degenerate $\mathbf{C}$ containing a single context, and a flow insensitive approach like kCFA_I may ignore program point n in $\mathsf{pts}(n, c)$, giving the same points-to information at all program points for a local variable of a method under context c. We next formally describe our three clients using the above results.

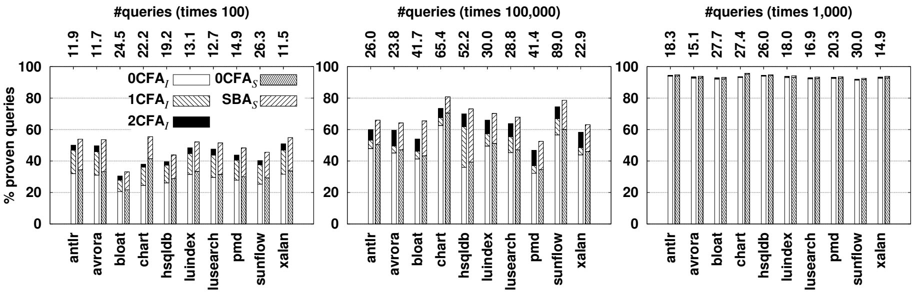
Downcast Safety. This client statically checks the safety of downcasts. A safe downcast is one that cannot fail because the object to which it is applied is guaranteed to be a subtype of the target type. Thus, safe downcasts obviate the need for run-time cast checking. We define this client in terms of the $\mathsf{downcast}$ predicate: $\mathsf{downcast}(e) \iff \exists c : \{\ \mathit{type}(h) \mid h \in \mathsf{pts}(n,c)(v)\ \} \nsubseteq \mathit{sub}(t)$, where the command at control-flow edge e with $\mathit{origin}(e) = n$ is a cast statement $v' = (t)\ v$. The predicate checks if the type of some allocation site in the points-to set of v is not a subtype of the target type t. Each query to this client is a cast statement at e in the program. It is proven by an analysis if $\mathsf{downcast}(e)$ evaluates to false using points-to information pts computed by the analysis.

Call Graph Reachability. This client determines pairwise reachability between every pair of methods. The motivation is that the different approaches in Table 2 may not differ much in broad statistics about the size of the call graph they produce, such as the number of reachable methods, but they can differ dramatically in the number of paths in the graph. This metric in turn may significantly impact the precision of call graph clients.

We define this client in terms of the reach predicate:

$$\begin{aligned}&\mathsf{reach}(m,m') \iff \exists c,e,c' :\ \mathit{method}(e) = m\ \wedge\ (c,e,c',m') \in R,\\ &(\text{where } R = \mathsf{leastFix}\ \lambda X.\ (\mathsf{cg}\ \cup\ \{(c,e,c'',m) \mid \exists c',m',e' :\ \mathit{method}(e') = m'\ \wedge\\ &\qquad\qquad\qquad\qquad (c,e,c',m') \in X\ \wedge\ (c',e',c'',m) \in \mathsf{cg}\})).\end{aligned}$$

The above predicate is true if there exists a path in the context sensitive call graph from m to m'. The existence of such a path means that it may be possible



(a) Downcast safety. (b) Call graph reachability. (c) Monomorphic inference.

Fig. 8. Precision of various interprocedural approaches on clients of pointer analysis

Table 3. Statistics of call graphs computed by various interprocedural approaches

		antlr	avrora	bloat	chart	hsqldb	luindex	lusearch	pmd	sunflow	xalan
number of edges in call graph as % of 0CFA$_I$	0CFA$_I$	26,871	25,427	42,766	41,655	38,703	28,064	27,978	32,447	49,502	25,037
	1CFA$_I$	95.8	96.3	96.4	96.0	92.5	96.3	96.7	96.8	94.2	96.3
	2CFA$_I$	93.6	93.9	94.7	94.6	90.8	94.0	94.3	94.7	91.7	93.8
	0CFA$_S$	98.0	98.6	98.6	81.3	97.2	98.7	98.7	98.2	95.6	98.6
	SBA$_S$	91.4	91.5	92.4	75.8	87.4	91.9	91.6	91.9	86.8	91.5
number of reachable methods as % of 0CFA$_I$	0CFA$_I$	7,220	6,905	9,133	11,450	10,223	7,741	7,601	9,105	13,356	6,772
	1CFA$_I$	98.7	99.0	99.1	99.0	99.1	99.0	99.1	99.2	99.2	99.0
	2CFA$_I$	98.0	98.3	98.6	98.6	98.5	98.4	98.4	98.7	98.6	98.3
	0CFA$_S$	98.9	99.2	99.2	81.8	98.1	99.3	99.3	99.1	95.9	99.2
	SBA$_S$	96.8	97.1	97.3	80.3	96.6	97.3	97.3	97.4	94.4	97.1
$\frac{\text{total \# contexts}}{\text{total \# methods}}$	1CFA$_I$	5.3	5.1	6.9	5.2	5.1	5.1	5.1	5.0	4.9	5.1
	2CFA$_I$	41.9	41.8	54.6	35.7	34.3	38.8	39.9	35.9	31.5	42.4
	SBA$_S$	6.7	6.4	9.9	7.2	6.6	6.4	6.2	6.8	7.4	6.4

to invoke m' while m is on the call stack, either directly or transitively from a call site in the body m. Each query to this client is a pair of methods (m, m') in the program. This query is proven by an analysis if $\mathsf{reach}(m, m')$ evaluates to false using the call graph cg computed by that analysis—no path exists from m to m' in the graph.

Monomorphic Call Inference. Monomorphic call sites are dynamically dispatched call sites with at most one target method. They can be transformed into statically dispatched ones that are cheaper to run. We define a client to statically infer such sites, in terms of the polycall predicate: $\mathsf{polycall}(e) \iff |\{\ m \mid \exists c, c' : (c, e, c', m) \in \mathsf{cg}\}| > 1$, where the command at control-flow edge e is a dynamically dispatching call. Each query to this client is a dynamically dispatched call site e in the program. The query is proven by an analysis if $\mathsf{polycall}(e)$ evaluates to false using call graph cg computed by that analysis.

We next summarize our evaluation results, including precision, interesting statistics, and scalability of the various approaches on our pointer analysis and its three clients described above.

Table 4. Running time of pointer analysis using various approaches

	antlr	avrora	bloat	chart	hsqldb	luindex	lusearch	pmd	sunflow	xalan
0CFA$_I$	1m45s	1m42s	3m10s	4m40s	3m29s	2m34s	2m22s	3m52s	5m00s	2m32s
1CFA$_I$	40m	38m	82m	121m	74m	41m	43m	61m	148m	36m
2CFA$_I$	72m	68m	239m	256m	158m	83m	80m	112m	279m	82m
0CFA$_S$	23m	26m	38m	30m	34m	35m	24m	34m	58m	23m
SBA$_S$	21m	17m	60m	51m	37m	27m	16m	29m	72m	16m

Precision on Clients. Figure 8 shows the precision of the approaches on the three clients. We measure the precision of an approach on a client in terms of how many queries posed by the client can be proven by the approach on each benchmark. The total number of queries is shown at the top. For instance, for antlr, there are 11.9×10^2 queries by the downcast safety client.

The stacked bars in the plots show the fraction of queries proven by the various approaches. We use separate bars for the flow-insensitive and flow-sensitive approaches, and vary only the degree of context sensitivity within each bar. At the base of each kind of bar is the fraction of queries proven by the context insensitive approaches (0CFA$_I$ and 0CFA$_S$). The bars stacked above them denote fractions of queries proven exclusively by the indicated context sensitive approaches. For instance, for the downcast safety client on antlr, the left bar shows that 0CFA$_I$ proves 32% queries, 1CFA$_I$ proves an additional 15% queries (for a total of 47% proven queries), and 2CFA$_I$ proves another 3% queries (for a total of 50% proven queries). The right bar shows that 0CFA$_S$ proves 34% queries, and SBA$_S$ proves an additional 20% queries (for a total of 54% proven queries). We next briefly summarize the results.

The SBA$_S$ approach is theoretically the most precise of all five approaches. Compared to the next most precise approach 2CFA$_I$, it proves 12% more downcast safety queries on average per benchmark, and 9% more call graph reachability queries, but only 0.6% more monomorphic call site inference queries. The largest gain of SBA$_S$ over 2CFA$_I$ is 21.3%, and occurs on bloat for the call graph reachability client. The relatively lower benefit of increased context sensitivity for the monomorphic call site inference client is because the context insensitive approaches are themselves able to prove over 90% of the queries by this client on each benchmark. We also observe that 0CFA$_S$ proves only slightly more queries than 0CFA$_I$ for each client on each benchmark, suggesting that flow sensitivity is ineffective without an accompanying increase in context sensitivity. In particular, with the exception of chart, 0CFA$_S$ proves less queries than 1CFA$_I$.

Call Graph Statistics. We found it instructive to study various statistics of the call graphs computed by the different approaches. The first two sets of rows in Table 3 show the number of reachable methods and the number of edges in the call graphs computed by the different approaches. Both decrease with an increase in the precision of the approach, as expected. But the reduction is much smaller compared to that in the number of unproven queries for the call graph reachability client, shown in Figure 8(b). An unproven reach(m, m') query indicates the presence of one or more paths in the call graph from m to m' and the higher the number of such unproven queries, the higher the number of paths in the call graph. The average reduction in the number of such unproven queries from $\mathsf{0CFA}_I$ to SBA_S is 41%, but the corresponding average reduction in the number of call graph edges is only 10.8%, and that in the number of reachable methods is even smaller, at 4.8%. From these numbers, we conclude that the various approaches do not differ much in coarse-grained statistics of the call graphs they produce (e.g., the number of reachable methods) but they can differ dramatically in finer-grained statistics (e.g., the number of paths), which in turn can greatly impact the precision of certain clients.

Scalability. Lastly, we compare the scalability of the different approaches. Table 4 shows their running time on our pointer analysis, exclusive of the clients' running time which is negligible. The running time increases from $\mathsf{0CFA}_I$ to $\mathsf{2CFA}_I$ with large differences between the different flow insensitive approaches. The similar running times of $\mathsf{0CFA}_S$ and SBA_S is because of the use of the tabulation algorithm with almost identical implementation for both. Finally, SBA_S runs much faster than $\mathsf{2CFA}_I$ on all benchmarks.

The improved performance of SBA_S over $\mathsf{2CFA}_I$ can be explained by the ratio of the number of contexts to that of reachable methods computed by each approach. This ratio is shown in the bottom set of rows in Table 3 for $\mathsf{1CFA}_I$, $\mathsf{2CFA}_I$, and SBA_S. (It is not shown for context insensitive approaches $\mathsf{0CFA}_I$ and $\mathsf{0CFA}_S$ as it is the constant 1 for them.) These numbers elicit two key observations. First, the rate at which the ratio increases as we go from $\mathsf{0CFA}_I$ to $\mathsf{2CFA}_I$ suggests that call-strings approaches with $k \geq 3$ run out of memory by computing too many contexts. Second, $\mathsf{2CFA}_I$ computes almost 4X-7X more contexts per method than SBA_S on each benchmark, implying that the summary-based approach used in SBA_S is able to merge many call-string contexts.

The primary purpose of the empirical evaluation in this work was to determine how precise k-CFA can get using arbitrary k. The proof of equivalence between ∞-CFA and SBA enabled us to use SBA_S for this evaluation. However, other works [MRR05, LH08] have shown that, in practice, using *object-sensitivity* [MRR02, SBL11] to distinguish calling contexts for object-oriented programs is more precise and scalable than k-CFA. Though call string and object-sensitive contexts are incomparable in theory, an interesting empirical evaluation in future work would be to compare the precision of ∞-CFA with analyses using object-sensitive contexts.

6 Related Work

This section relates our work to existing equivalence results, work on summary-based approaches, and work on cloning-based approaches of which call-strings approaches are an instance.

Equivalence Results. Sharir and Pnueli [SP81] prove that the summary-based and call-strings approaches are equivalent in the finite, distributive setting. They provide constructive algorithms for both approaches in this setting: an iterative fixpoint algorithm for the summary-based approach and an algorithm to obtain a finite bound on the lengths of call strings to be computed for the call-strings approach. They prove each of these algorithms equivalent to the meet-over-all-valid-paths (MVP) solution (see Corollary 3.5 and Theorem 5.4 in [SP81]). Their equivalence proof thus relies on the distributivity assumption. Our work can be viewed as an extension of their result to the more general non-distributive setting. Also, they do not provide any empirical results, whereas we measure the precision and scalability of both approaches on a widely-used pointer analysis, using real-world programs and clients.

For points-to analyses, Grove and Chambers [GC01] conjectured that Agesen's Cartesian Product Algorithm (CPA) [Age95] is strictly more precise than ∞-CFA, and that SBA(which they refer as SCS for Simple Class Set)has the same precision as ∞-CFA. The first conjecture was shown to be true by Besson [Bes09] while we proved that the second conjecture also holds in this work.

Might et al. [MSH10] show the equivalence between k-CFA in the object-oriented and functional paradigms. The treatment of objects vs. closures in the two paradigms causes the same k-CFA algorithm to be polynomial in program size in the object-oriented paradigm but EXPTIME-complete in the functional paradigm. Our work is orthogonal to theirs. Specifically, our formal setting is agnostic to language features, assuming only a finite abstract domain $\mathbf{\Gamma}$ and monotone transfer functions $[\![\_]\!]$, and indeed instantiating these differently for different language features can cause the k-CFA algorithm to have different complexity.

Summary-Based Interprocedural Analysis. Sharir and Pnueli [SP81] first proposed using functional summaries to solve interprocedural dataflow problems precisely. Later, Reps et al. [RHS95] proposed an efficient quadratic representation of functional summaries for finite, distributive dataflow problems, and the tabulation algorithm based on CFL-reachability to solve them in cubic time. More recent works have applied the tabulation algorithm in non-distributive settings, ranging from doing a fully lossy join to a partial join to a lossless join. All these settings besides lossy join are challenging to scale, and either use symbolic representations (e.g., BDDs in [BR01]) to compactly represent multiple abstract states, or share common parts of multiple abstract states without losing precision (e.g., [YLB^{+}08, MSRF04]) or at the expense of precision (e.g., [BPR01]). Summary-based approaches like CFA2 [VS10] have also been proposed for functional languages to perform fully context-sensitive control-flow analysis. Our work is motivated by the desire to understand the formal relationship between the widely-used summary-based approach in non-distributive settings and the call-strings approach, which is also prevalent as we survey next.

Cloning-Based Interprocedural Analysis. There is a large body of work on bounded call-string-like approaches that we collectively call *cloning-based* approaches. Besides k-CFA [Shi88], another popular approach is *k-object sensitive analysis* for object-oriented programs [MRR02, SBL11]. Many recent works express cloning-based pointer analyses in Datalog and solve them using specialized Datalog solvers [Wha07, BS09]. These solvers exploit redundancy arising from large numbers of similar contexts computed by these approaches for high k values. They either use BDDs [BLQ^{+}03, WL04, ZC04] or explicit representations from the databases literature [BS09] for this purpose. Most cloning-based approaches approximate recursion in an ad hoc manner. An exception is the work of Khedker et al. [KMR12, KK08] which maintains a single representative call string for each equivalence class. Unlike the above approaches, it does not approximate recursion in an ad hoc manner, and yet it is efficient in practice by avoiding the computation of redundant call-string contexts. Our pointer analysis achieves a similar effect but by using the tabulation algorithm.

7 Conclusion

We showed the equivalence between the summary-based and unbounded call-strings approaches to interprocedural analysis, in the presence of a lossy join. Our result extends the formal relationship between these approaches to a setting more general than the distributive case in which this result was previously proven. We presented new implications of our result to the theory and practice of interprocedural analysis. On the theoretical side, we introduced new proof techniques that enable to reason about relationships that do not hold between two fixpoint computations at each step, but do so when a form of garbage collection is applied to the final results of those computations. On the practical side, we empirically compared the summary-based and bounded call-strings approaches on a widely-used pointer analysis with a lossy join. We found the summary-based approach on this analysis is more scalable while providing the same precision as the unbounded call-strings approach.

Acknowledgement. We thank the anonymous reviewers for insightful comments. This work was supported by DARPA under agreement #FA8750-12-2-0020, NSF award #1253867, gifts from Google and Microsoft, and EPSRC. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[Age95] Agesen, O.: The cartesian product algorithm. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995)

[Bes09] Besson, F.: CPA beats ∞-CFA. In: FTfJP (2009)

[BLQ$^+$03] Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: PLDI (2003)

[BPR01] Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)

[BR01] Ball, T., Rajamani, S.: Bebop: a path-sensitive interprocedural dataflow engine. In: PASTE (2001)

[BS09] Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: OOPSLA (2009)

[CC79] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL (1979)

[CC92] Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4) (1992)

[FYD$^+$08] Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM TOSEM 17(2) (2008)

[GC01] Grove, D., Chambers, C.: A framework for call graph construction algorithms. ACM TOPLAS 23(6) (2001)

[KK08] Khedker, U.P., Karkare, B.: Efficiency, precision, simplicity, and generality in interprocedural dataflow analysis: Resurrecting the classical call strings method. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 213–228. Springer, Heidelberg (2008)

[KMR12] Khedker, U.P., Mycroft, A., Rawat, P.S.: Liveness-based pointer analysis. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 265–282. Springer, Heidelberg (2012)

[LH08] Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM TOSEM 18(1) (2008)

[Min06] Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1) (2006)

[MRR02] Milanova, A., Rountev, A., Ryder, B.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: ISSTA (2002)

[MRR05] Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM TOSEM 14(1) (2005)

[MSH10] Might, M., Smaragdakis, Y., Horn, D.: Resolving and exploiting the k-CFA paradox: illuminating functional vs. oo program analysis. In: PLDI (2010)

[MSRF04] Manevich, R., Sagiv, M., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)

[RHS95] Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL (1995)

[SBL11] Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: POPL (2011)

[Shi88] Shivers, O.: Control-flow analysis in scheme. In: PLDI (1988)

[SP81] Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, ch. 7. Prentice-Hall (1981)

[VS10] Vardoulakis, D., Shivers, O.: CFA2: A Context-Free Approach to Control-Flow Analysis. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 570–589. Springer, Heidelberg (2010)

[Wha07] Whaley, J.: Context-Sensitive Pointer Analysis using Binary Decision Diagrams. PhD thesis, Stanford University (March 2007)

[WL04] Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI (2004)

[YLB$^+$08] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)

[ZC04] Zhu, J., Calman, S.: Symbolic pointer analysis revisited. In: PLDI (2004)