IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries[*]

Atanas Rountev, Mariana Sharp, and Guoqing Xu

Ohio State University, USA

Abstract. A key scalability challenge for interprocedural dataflow analysis comes from large libraries. Our work addresses this challenge for the general category of interprocedural distributive environment (IDE) dataflow problems. Using pre-computed library summary information, the proposed approach reduces significantly the cost of whole-program IDE analyses without any loss of precision. We define an approach for library summary generation by using a graph representation of dataflow summary functions, and by abstracting away redundant dataflow facts that are internal to the library. Our approach also handles object-oriented features, by employing an IDE type analysis as well as special handling of polymorphic library call sites whose target methods depend on the future (unknown) client code. Experimental results show that dramatic cost savings can be achieved with the help of these techniques.

1 Introduction

Interprocedural dataflow analysis plays an important role in compilers and various software tools. A key scalability challenge for analysis algorithms comes from large libraries. Systems are inevitably built with standard libraries (e.g., Java J2SE or C++ STL), domain-specific libraries (e.g., graphics, linear algebra, etc.), and middleware (e.g., EJB). The size of the client code is often a small fraction of the size of the library code being used by that client code.

In this paper we focus on whole-program interprocedural dataflow analysis for Java. However, the proposed approach should also be applicable to other object-oriented languages. Our target is a general category of dataflow problems referred to as *interprocedural distributive environment* (IDE) problems [1]. The goal is to reduce the cost of whole-program IDE analyses by using pre-computed *library summary information.* Library code is analyzed independently of any client code, in oder to produce a library summary stored on disk; this summary is reusable for subsequent analysis of any client code. The summary-generation analysis produces a *precise* summary: the solution for the client code, computed using the summary, is as precise as the solution what would have been computed if we were to use a whole-program analysis of client+library code.

Existing work by Sagiv et al. [1] already provides a solution for one key problem: the representation and manipulation of dataflow functions. Based on their

[*] This material is based upon work supported by NSF under grant CCF-0546040.

L. Hendren (Ed.): CC 2008, LNCS 4959, pp. 53–68, 2008.

techniques, we define a general approach for library summary generation. One important problem is that the summary may contain redundant dataflow facts that do not affect the analysis of the client code. We solve this problem through *abstracted* versions of summary functions, in order to filter out callee-local details. Another key problem are polymorphic library call sites whose target methods depend on the future (unknown) client code. We propose the use of IDE type analysis to identify a subset of these sites that are client-*in*dependent and can be processed precisely. The client-dependent call sites are left unresolved in the summary, until client code becomes available. This approach also handles library callback sites that may invoke callback methods defined in future clients.

Contributions. This work makes the following specific contributions:

- *Whole-program analysis.* A general framework for whole-program IDE analyses for object-oriented programs, which extends the classical approach for procedural languages [2,3,4,1] through an IDE type analysis (Section 2).
- *Summary-generation analyses.* A general algorithm for summary generation with abstracted summary functions, capturing the dataflow effects of sets of control-flow paths, with special treatment of polymorphic calls.
- *Dependence analysis and type analysis.* Two instances of the general approach: an IDE data dependence analysis that plays an important role in the construction of system dependence graphs, and an IDE type analysis.
- *Experimental evaluation.* A study using the 10238 classes of the Java libraries, and 20 client programs. The experimental results show that dramatic cost savings can be achieved with the help of these techniques

2 Whole-Program IDE Dataflow Problems

In *interprocedural distributive environment* (IDE) dataflow problems [1], the dataflow facts are maps ("environments") from some set of symbols D to lattice elements from a semi-lattice L. The IDE class is a general category of dataflow problems, examples of which are copy-constant propagation and linear-constant propagation [1], object naming analysis [5], 0-CFA type analysis [6,7,8], and all IFDS (interprocedural, finite, distributive, subset) problems [3] such as reaching definitions, available expressions, live variables, truly-live variables, possibly-uninitialized variables, flow-sensitive side-effects [9], some forms of may-alias and must-alias analysis [4], and interprocedural slicing [10].

A program is represented by an interprocedural control-flow graph (ICFG). Each call expression is represented by two nodes: a *call-site* node and a *return-site* node. Interprocedural edges connect a call-site node with the start node of the invoked procedure p, and the exit node of p with the return-site node (assuming a single exit node per procedure.) An intraprocedural edge may also be added from the call-site to the return-site [1]. A *valid* ICFG path has (call-site, start) and (exit, return-site) edges that are properly matched [2,3,1].

An *environment* is a map $D \to L$ where D is a finite set of symbols and L is a finite-height meet semi-lattice with a top element $\top$ and a meet operator $\wedge$. Let $Env(D, L)$ be the set of all environments for a given pair (D, L). The meet operator $\wedge$ extended to environments is $env_1 \wedge env_2 = \lambda d.(env_1(d) \wedge env_2(d))$. The top element in $Env(D, L)$, denoted by Ω, is $\lambda d.\top$. For any $env \in Env(D, L)$, $d \in D$, and $l \in L$, $env[d \mapsto l]$ denotes an environment in which each symbol d' is mapped to $env(d')$, except for d which is mapped to l.

Functions $t : Env(D, L) \to Env(D, L)$ are *environment transformers*. A distributive transformer t distributes over $\wedge$. An instance of an IDE problem is (G, D, L, M) where G is the ICFG and M is a map that associates distributive transformers with the edges of G. A safe analysis for an IDE problem computes an over-approximation of the meet-over-all-valid-paths solution for any node n: the solution at n is $\leq$ the meet of $f_q(\Omega)$ for all valid paths q from the start node of the program to n, where f_q is the composition of the transformers of q's edges.

Some problems are naturally defined with the approximation that any ordering of statements in a procedure is possible; these are intraprocedurally *flow-insensitive* problems. They can be encoded by conceptually modifying each procedure's CFG to represent arbitrary compositions and meets of transformers, using a switch-in-a-loop structure [11]. In this case all nodes in the same procedure have the same solution. A *context-insensitive* problem does not distinguish the different calling contexts of a procedure. A flow- and context-insensitive problem can be modeled by a single conceptual switch-in-a-loop graph for the entire program; in this case all program statements have the same solution.

Solving IDE Problems. Sagiv et al. [1] define a technique for precise computation of the meet-over-all-valid-paths solution, based on the "functional" approach by Sharir and Pnueli [2]. The first phase on the functional approach computes a *summary function* ϕ_n for each ICFG node n, representing the solution at n as a function of the solution at the start node of the procedure p containing n. If n is the exit node of p, ϕ_n is a summary function for the entire procedure p. During a bottom-up traversal of the SCC-DAG of the call graph, the functions for p's callees are used to model the effects of calls made by p. In the second phase, the actual solution is determined at each ICFG node through top-down propagation based on the summary functions. It is possible to merge these two phases, resulting in a single top-down algorithm which computes ϕ_n incrementally only for lattice elements that reach p's entry. The work in [1] applies this technique to IDE problems (where ϕ_n are environment transformers) by using a *compact graph representation* for transformers; as a result, a summary function can be modeled by a (small) graph. The composition, meet, and application of transformers can be implemented as inexpensive graph operations, and the analysis algorithms can be designed based on a generalized form of graph reachability (essentially, graph summarization along valid paths).

2.1 Interprocedural Dependence Analysis

To illustrate the general approach for solving IDE problems, we will use a particular form of interprocedural dependence analysis for Java.[1] For each method m

[1] Without loss of generality, the subsequent discussion assumes a certain simplified program representation (based on Jimple in the Soot analysis framework [12]). For brevity, details of this representation are provided elsewhere [13].

with a non-void return type, the analysis computes the set of formal parameters of m on which the return value of m may depend directly or transitively. This output is essentially a set of *transitive-dependence summary edges* [10] which play a key role in a variety of analyses for interprocedural slicing, program refactoring, change impact analysis, etc. For simplicity, we restrict the discussion to data dependencies (control dependencies are easy to add to the formulation, and are handled by our implementation), non-exceptional flow of control, and stack memory locations (i.e., dependencies through the heap are not modeled; they could be added using a conservative approach which maps each expression `x.fld` to a single abstract location `fld`). Even with these restrictions, the analysis exhibits the essential features of flow- and context-sensitive IDE analyses.

We propose an IDE formulation[2] in which D is the set of all local variables and formal parameters, and the L is the powerset of the set F of formal parameters, with partial order $\supseteq$ and meet $\cup$. For any $env \in Env(D, L)$, the value of $env(d)$ for local/formal d in method m is the set of formal parameters f of m such that the current value of d may directly or transitively depend on the value that f had at the start of m. The final solutions at statements *return x* in a method m are used to find all formals of m on which its return value may depend.[3] Each such formal parameter defines an interprocedural transitive dependence which is a key component for the construction of the *system dependence graph* [10].

For an assignment $d := expr\{d_1, \ldots, d_k\}$, where the side-effect-free non-call expression $expr$ uses $d_i \in D$, and the input environment is env, the transformed environment is $env[d \mapsto \bigcup_i env(d_i)]$. Here d becomes dependent on every formal f on which some d_i is dependent. If $expr$ does not use any $d_i \in D$ (e.g., it is a constant expression), the transformed environment is $env[d \mapsto \emptyset]$. For all other non-call statements, as well as for calls without return values, the transformer is the identity function. A call $d := m(d_1, \ldots, d_k)$, can be treated as a sequence of actual-to-formal assignments, followed by the summary function for the callee method m, followed by an assignment of m's return value to d (with filtering due to scope changes). It is easy to prove that these transformers are distributive.

In general, an environment transformer can be represented by a bipartite directed graph with $2(|D|+1)$ nodes [1]. In each partition, $|D|$ nodes are labeled with $d \in D$, and one node is labeled with a special symbol Λ. The edges in the graph are labeled with functions $L \to L$. For the dependence analysis from above, there are only two kinds of edge labels: the identity function $\lambda l.l$ and the constant function $\lambda l.\emptyset$. Examples of these graphs are shown in Figure 1. The key property of this representation is that it is closed under transformer composition and meet. In essence, transformer meet corresponds to graph union, and composition is similar to graph transitive closure (with edge label composition).

[2] While inspired by [4], our formulation differs significantly from this previous work.

[3] In this case, a solution captures the effects of same-level valid paths [4] — that is, paths with the same number of calls and returns, starting at method entry. The solution at method entry is $\Omega[f_i \mapsto \{f_i\}]$: each formal f_i of the method depends on itself, and every other $d \in D$ is mapped to $\emptyset$ (i.e., d does not yet have dependencies).

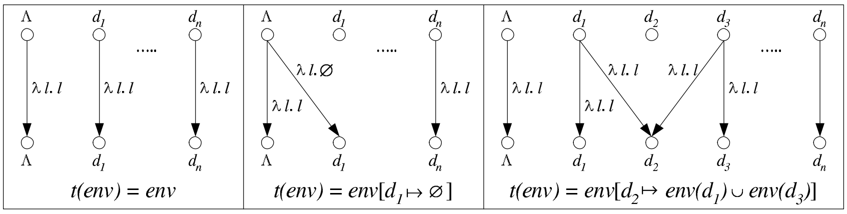


Fig. 1. Graph representation of environment transformers t

2.2 Type Analysis

The standard IDE formulation is applicable to procedural languages. For object-oriented languages, polymorphic calls require resolution of target methods, which can be done by any call graph construction analysis. We propose the use of one such approach: *0-CFA type analysis* [7,6,8]. This analysis has been investigated extensively, and has been shown to be a good compromise between cost and precision [14]. We we have restated 0-CFA as an IDE problem, which makes it a natural choice for use in a general IDE framework. Consider any IDE analysis A which requires a call graph in order to construct its ICFG. One option is to run 0-CFA as a pre-processing step before A. Alternatively, 0-CFA can be embedded in A by using the product of 0-CFA's environment set and A's environment set, resulting in a generalized type-aware version of A.

Intraprocedural IDE type analysis. First, we briefly outline the formulation of *intra*procedural 0-CFA analysis for Java as an IDE problem. Let D be the set of all local variables, formal parameters (including `this`), and fields. Also, let T be the set of all types that correspond to objects created at run time. An environment is a map $D \rightarrow 2^T$. The powerset 2^T is a lattice with partial order $\supseteq$ and meet $\cup$. For any environment *env* and local/formal/field $d \in D$, the value of $env(d)$ is a set of types for the run-time objects that may be referred to by d.

For brevity, we discuss only the following two categories of statements; our implementation handles the general case. First, in $d := alloc(X)$, an object of type X is created and a reference to it is assigned to $d \in D$. The environment transformer in this case is $\lambda env.env[\, d \mapsto env(d) \cup \{X\}\,]$; that is, type X is added to the set of types for d. Second, for an assignment $d_1 := d_2$ where $d_1, d_2 \in D$, the transformer is $\lambda env.env[\, d_1 \mapsto env(d_1) \cup env(d_2)\,]$: the set of types for d_2 is added to the set of types for d_1. These transformers are distributive, and therefore this is an IDE problem. Since 0-CFA is a flow-insensitive analysis, the transformers do not perform "kills" — i.e., $(t(env))(d) \supseteq env(d)$ for any transformer t.

In our formulation, the intraprocedural aspects of 0-CFA are equivalent to combining all transformers for a method's body through transformer composition and meet. The resulting transformer is the intraprocedural solution for a method. If transformers are represented by graphs (as defined in [1]), intraprocedural 0-CFA computes a fixed point under the corresponding graph operations. The resulting graph is a "one-hop" representation of all intraprocedural 0-CFA effects of a method: given some input environment which represents the state immediately before the execution of the method, only one application of the fixed-point graph to this input is enough to produce all necessary state updates.

Interprocedural aspects. Consider non-polymorphic calls (e.g., calls to static methods or constructors). Parameter passing and return values can be modeled as assignments, with the corresponding transformers. These transformers can be combined with the ones from the method bodies (with closure under composition and meet) to obtain one single transformer t^* for the entire program. The value of $t^*(\Omega)$ is the type analysis solution; here Ω is the environment that assigns to each $d \in D$ an empty set of types. Since 0-CFA is flow- and context-insensitive, there is only one solution for the entire program. A polymorphic call `x.m()` can be represented as a switch statement, with one branch per possible target method. The set of possible targets can be determined by examining the class hierarchy. Since we are interested in *on-the-fly* call graph construction, each target is considered infeasible until evidence to the contrary is seen. To achieve this, special transformers are introduced for the outgoing edges of the multi-way branch, in order to prune the receiver types. These transformers are of the form $\lambda env.env[\, x \mapsto env(x) \cap ReceiverTypes \,]$ for a call through `x`. Here *ReceiverTypes* is the set of receiver types for which virtual dispatch would invoke the target method for this branch. If the pruned type set is empty, the call is ignored.

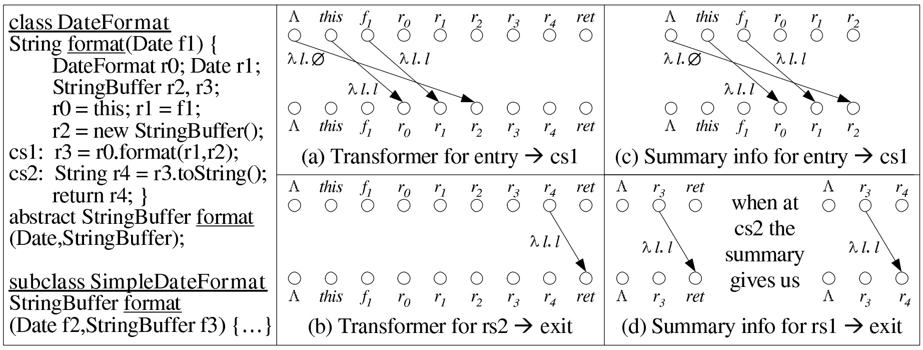
3 Summary Generation for Object-Oriented Libraries

Consider a large library *Lib* which is to be used by many (unknown) clients. Furthermore, suppose we already have some existing whole-program IDE dataflow analysis. Clearly, it is desirable to perform some of the analysis work for *Lib* in advance, independently of any library clients. The library summary information generated by this *summary generation analysis* can be stored on disk, and later used by a *summary-based analysis* of any client component *Main*. Our focus is on precision-preserving summary generation: for any ICFG node in *Main*, the solution computed by the summary-based analysis should be the same as the solution that would have been computed by the original whole-program analysis.

The proposed summary-generation approach performs as many transformer meets and compositions as possible in the library, and uses the result as summary information. Two key problems arise when applying this idea. First, the targets of call sites in the library may depend on the unknown code in client components. Some of these targets may be library methods that are feasible only for some (but not all) clients. Some call sites may even invoke callback methods defined in client code. Second, the library summary may contain redundant information that is internal to the library and does not affect the analysis of clients. For example, while locals in the library play an important role *during* the computation of summary functions, they may be irrelevant *after* the functions are computed.

3.1 Stage 1: Intraprocedural Summary Generation

For a library method that does not make any calls, the summary information can be computed as follows. The transformers for nodes in the method are



```
class DateFormat
String format(Date f1) {
      DateFormat r0; Date r1;
      StringBuffer r2, r3;
      r0 = this; r1 = f1;
      r2 = new StringBuffer();
cs1:  r3 = r0.format(r1,r2);
cs2:  String r4 = r3.toString();
      return r4; }
abstract StringBuffer format
(Date,StringBuffer);

subclass SimpleDateFormat
StringBuffer format
(Date f2,StringBuffer f3) {...}
```

Λ this f1 r0 r1 r2 r3 r4 ret
λl.∅
λl.l
λl.l
Λ this f1 r0 r1 r2 r3 r4 ret
Λ this f1 r0 r1 r2
λl.∅
λl.l
λl.l
Λ this f1 r0 r1 r2
Λ this f1 r0 r1 r2 r3 r4 ret
λl.l
Λ this f1 r0 r1 r2 r3 r4 ret
Λ r3 ret
λl.l
Λ r3 ret
when at cs2 the summary gives us
Λ r3 r4
λl.l
Λ r3 r4

(a) Transformer for entry → cs1

(b) Transformer for rs2 → exit

(c) Summary info for entry → cs1

(d) Summary info for rs1 → exit

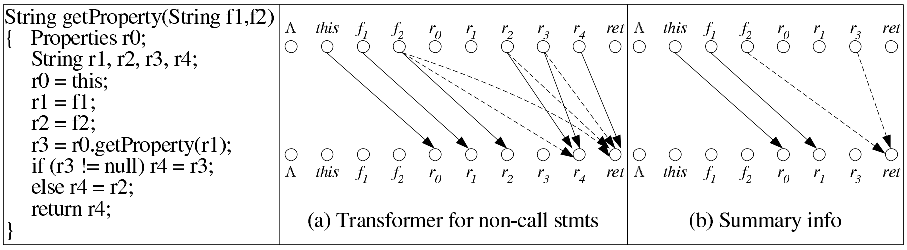
Fig. 2. Summary information for dependence analysis (only non-trivial edges)

combined using composition and meet: for each node n, the summary function is $\phi_n = \bigwedge f_q$, where the meet is over all paths q from the start node to n, and f_q is the composition of the transformers of q's edges. The summary function for the exit node (represented as a graph) serves as the summary function for the method.[4] If type analysis is performed as a pre-processing step (as opposed to being embedded in the main IDE analysis), the summary information also contains the graph representation of the fixed-point transformer for type analysis.

Suppose the analyzed method contains a set of call-site nodes $cs_1, cs_2, \ldots, cs_k$ with the corresponding return-site nodes rs_i. In this case, the summary generation produces *a set of summary functions* ψ^n_m, where n is the entry node or some rs_i, and m is the exit node or some cs_i. Transformer ψ^n_m is the meet of f_q for all intra-method paths q from n to m such that q contains no calls other than n and m. This set of summary functions captures all intraprocedural effects of the method, and leaves unresolved the effects of all calls. In addition to the set of ψ^n_m (represented as graphs), the summary information for the method also contains descriptions of all call sites (e.g., compile-time target methods, actual parameters, etc.). If the type analysis is a separate pre-processing step, the summary also contains a single transformer which is the fixed-point meet and composition of all type-analysis transformers for non-call statements in the method.

► ***Examples.*** Figure 2 shows an example based on class `DateFormat` and its subclass `SimpleDateFormat` from `java.text` in the Java 1.4.2 libraries. Consider the dependence analysis from Section 2.1. Part (a) illustrates transformer $\psi^{entry}_{cs_1}$, which corresponds to a single path along which `r0`, `r1`, and `r2` are assigned (for brevity, the constructor call for `StringBuffer` is not discussed). Part (b) shows $\psi^{rs_2}_{exit}$, using an artificial variable `ret` to represent the method's return value.

[4] Strictly speaking, since summary generation is independent of any client code, the whole-program D and L are not fully known, and the summary function is not a single transformer but rather an infinite set of transformers, one per possible client.



```
String getProperty(String f1,f2)
{   Properties r0;
    String r1, r2, r3, r4;
    r0 = this;
    r1 = f1;
    r2 = f2;
    r3 = r0.getProperty(r1);
    if (r3 != null) r4 = r3;
    else r4 = r2;
    return r4;
}
```

Λ this f1 f2 r0 r1 r2 r3 r4 ret
Λ this f1 f2 r0 r1 r2 r3 r4 ret
(a) Transformer for non-call stmts
Λ this f1 f2 r0 r1 r3 ret
Λ this f1 f2 r0 r1 r3 ret
(b) Summary info

Fig. 3. Summary information for type analysis (only non-trivial edges)

Figure 3 shows another example, based on `java.util.Properties`, to illustrate the type analysis. The transformers for all non-call statements are combined through composition and meet. The resulting fixed-point transformer is shown in part (a) of the figure. Unlike the multiple ψ_m^n needed for the flow-sensitive dependence analysis, only one ψ is needed for the flow-insensitive type analysis. All edges are labeled with $\lambda l.l$. Dashed edges represent transitive relationships due to transformer composition. ◀

Abstracted summary functions. The functions from Figure 2(a)/(b) and Figure 3(a) contain redundant information. Consider $\psi_{cs_1}^{entry}$ which represents the flow from the formals of the method to the actual parameters of the call at cs_1 (including local `r0` which refers to the receiver object). Here the only relevant elements of D are `this`, `f1`, `r0`, `r1`, and `r2`. Thus, the summary information can store an *abstracted summary function* $\widehat{\psi}_{cs_1}^{entry}$ instead of the original summary function $\psi_{cs_1}^{entry}$. Figure 2(c) shows the graph representation of $\widehat{\psi}_{cs_1}^{entry}$. Similar considerations apply to the type analysis. The only elements of D that are relevant outside of the method are formals `this`, `f1`, and `f2`, the return variable `ret`, the actuals `r0` and `r1` at the call site, and the local `r3` to which the return value of the call is assigned. Both `r2` and `r4` can be eliminated from transformer ψ; the resulting abstracted transformer $\widehat{\psi}$ is shown in Figure 3(b).

In general, for any IDE analysis, only a subset of D is relevant with respect to a particular ψ_m^n. Depending on the specific analysis and on n and m, this subset would typically be related to formal parameters, return statements, actuals at calls, and return values at calls. Thus, it should be possible to define a corresponding abstracted transformer $\widehat{\psi}_m^n$ that can be used instead of ψ_m^n without any loss of precision. Obtaining the graph representation of $\widehat{\psi}_m^n$ should be trivial, given the already-computed representation of ψ_m^n. For the dependence analysis discussed earlier, $\widehat{\psi}_m^n$ can be defined as follows: (1) if n is an entry node, the formals should be preserved; (2) if m is an exit node, the return variable `ret` should be preserved; (3) if n is a return-site node, the local variable to which the return value is assigned should be preserved; and (4) if m is a call-site node, the actual parameters should be preserved, including the reference to the receiver object. The abstracted transformer $\widehat{\psi}$ for the type analysis can be defined similarly.

3.2 Stage 2: Interprocedural Summary Generation

In the standard IDE formulation, each call has a single target which is known at analysis time. For a library method that makes calls, its summary information can be computed by "inlining" the summary functions for callee methods, and then performing the intraprocedural propagation outlined above. As a result, a single summary function $\widehat{\psi}_{exit}^{entry}$ would be computed for the entire method.

This approach is possible only in the absence of callbacks from *Lib* to client code. If a library method m contains a callback site, the complete behavior of m is not known at summary-generation time, and it is not possible to create a complete summary function. This is a realistic problem, because callbacks occur often in object-oriented libraries (e.g., due to polymorphic calls in C++, Java, and C#). Consider the abstract method `format` in class `DateFormat` from Figure 2. If a client component creates a subclass with a corresponding non-abstract method `format`, call site cs_1 in Figure 2 could be a callback site. This situation is common for extensible object-oriented libraries. Note that cs_1 is not necessarily a callback site: if the client code simply uses library subclass `SimpleDateFormat`, the target of cs_1 would be the corresponding library method.

Even in the absence of callbacks, it may still be impossible to create precise summary functions. Consider the following Java example: library method m has a virtual call `a.n()` and the compile-time type of `a` is `A`. Suppose library classes `B` and `C` extend `A`, and method `A.n` is overridden by `B.n` and `C.n`. A conservative analysis has to assume that `a.n()` could invoke any of these three methods, and thus the summary function for m will depend on all three callees. But, for example, if a client instantiates only `C`, the summary would be too conservative.

Exit calls. A call site is an *exit call* if it can invoke some method that "exits" the scope of the analysis and therefore the effects of the call cannot be modeled. An exit call is a virtual call `x.m()` for which (1) the declared type of `x` has possible unknown subtypes, and (2) the compile-time target method of the call can be overridden by unknown methods. A library type T (class or interface type) is considered to have potential unknown subtypes in clients when T or some library subtype of T is public and not final.[5] The compile-time target method m of the call site can have unknown overriding methods if (1) m is not private and is not final, and (2) at least one of m's overriding methods in the library (or m itself) is non-final and is visible to clients (i.e., public or protected).

Fixed calls. A *fixed call site* in the library has exactly one possible run-time target method, regardless of what the client code may be. This target is a library method and can be determined at summary generation time. Obviously, an exit call is not a fixed call. A non-exit call is fixed if any of the following cases holds. In case 1, the call invokes a static method or a constructor, and thus the run-time target is the same as the compile-time target. In case 2, the call is a virtual invocation, and conservative analysis of the type hierarchy for the entire library determines that the call has exactly one possible run-time target method

[5] This definition assumes that library packages are sealed, and clients cannot add new classes to them (thus, non-public types cannot be accessed directly by client code).

regardless of client code. For example, for cs_2 in Figure 2, `r3` is of compile-time type `StringBuffer` which is a final class; thus, the only possible target is the corresponding method in this class. In case 3, the call is a virtual invocation, and conservative intraprocedural 0-CFA type analysis determines that the call has exactly one possible run-time target method regardless of client code. Consider a call site `x.m()`. In the graph representation of the transformer $\widehat{\psi}$ computed by the intraprocedural type analysis, the only edges reaching x should be of the form $\Lambda \rightarrow x$; in other words, the only values of `x` should come from inside the method. The label on the $\Lambda \rightarrow x$ edge is exactly the set of possible types for `x`.

Fixed methods. Consider a fixed call site cs and suppose that its unique target method m contains only fixed calls (or no calls at all), and this property transitively holds for all methods reachable from m. We will refer to such m as *fixed methods.* Here the effects of m are fully known at summary-generation time, and can be represented by a summary function $\widehat{\psi}_{exit}^{entry}$ for m, computed through a bottom-up traversal of the SCC-DAG of the "fixed" library call graph (i.e., the call graph in which nodes are fixed methods and edges are fixed calls).

In the method m' containing cs, m's summary function can be instantiated as follows. Consider any pair of summary functions $\widehat{\psi}_{cs}^{n_1}$ and $\widehat{\psi}_{n_2}^{rs}$ computed in m'; here rs is the return site corresponding to cs. The composition of these functions with the summary function for m, followed by the appropriate abstraction operations, produces a summary function $\widehat{\psi}_{n_2}^{n_1}$. If the pair (n_1, n_2) already has a corresponding function (i.e., because there is some call-free-path from n_1 to n_2), the new function is merged with the old one through transformer meet.

► ***Example.*** Figure 2(b) shows $\widehat{\psi}_{exit}^{rs_2}$. Consider call site cs_2, which is fixed. Suppose that its target method `StringBuffer.toString` is also fixed, and its summary function, instantiated at the call site, results in a transformer f_{cs_2} which shows a dependence from `r3` to `r4`. The right part of Figure 2(d) shows the graph representation of f_{cs_2}. The composition of f_{cs_2} and $\widehat{\psi}_{exit}^{rs_2}$ can be used to compute $\widehat{\psi}_{exit}^{rs_1}$. In addition to transformer composition, this computation can also abstract away `r4` because this variable is neither assigned the return value at rs_1, nor used at method exit. In general, after a summary function is instantiated as f_{cs} at a fixed call site cs, any pair of $\widehat{\psi}_{cs}^{n_1}$ and $\widehat{\psi}_{n_2}^{rs}$ can be used to create $\widehat{\psi}_{n_2}^{n_1}$ as an abstracted version of $\widehat{\psi}_{n_2}^{rs} \circ f_{cs} \circ \widehat{\psi}_{cs}^{n_1}$, based on the elements of D that need to be preserved for n_1 and n_2. The left part of Figure 2(d) shows the graph representation of $\widehat{\psi}_{exit}^{rs_1}$ after this abstraction. This summary function together with $\widehat{\psi}_{cs_1}^{entry}$, shown in Figure 2(c), defines the final summary information. ◄

If the summary function for a fixed method m is instantiated at all fixed call sites that invoke it, and if we can conservatively prove that no other call sites can directly invoke m (from the library or from client code), the summary $\widehat{\psi}_{exit}^{entry}$ for m does not need to be stored in the library summary at all. Due to space constraints, additional details on this optimization are presented elsewhere [13].

Table 1. Library summary information

(a) Library				(b) Dependence Analysis				(c) Type Analysis		
Pkg	*Cls*	*Mthd*	*Stmt*	1	2	3	4	1	2	3
`java`	1802	15676	245605	389024	584174	243005	151424	77940	111801	53215
`javax`	2265	17618	254542	390351	582970	278622	209549	88822	117102	65272
`org`	1289	8688	136945	180258	260013	134934	90893	55426	78490	32153
`com`	2373	18235	349957	517577	685128	323522	227939	125492	184347	84665
`sun`	2509	16973	508954	676324	820889	383865	246579	151060	207184	78291
Total	10238	77190	1496003	2153534	2933174	1363948	926384	498740	698924	313596

4 Experimental Evaluation

Study 1: Summary generation. Our experiments used the entire standard Java libraries from Java 2 SDK SE 1.4.2. Some characteristics of the packages in these libraries are summarized in part (a) of Table 1: number of classes *Cls*, number of methods *Mthd*, and number of statements *Stmt* in the intermediate representation (IR) provided by the Soot analysis framework [12].[6] The entire set of 10238 library classes was used as input to the summary-generation analysis. The running time of the analysis was 5491.6 seconds (about 90 minutes), on a single Intel Xeon 2.8GHz CPU in a Dell PowerEdge 1950 server. This time includes all Soot-related costs, the actual analysis time, and the disk I/O. The memory consumption was 1230.3 MB. The summary was written to disk in a straightforward binary format, with all necessary information for dependence analysis and type analysis. The total size of the summary file was 17.9 MB.

Part (b) of Table 1 provides relevant measurements for the dependence analysis. Our implementation generalizes the one outlined in Section 2.1 as follows. First, as an optimization, we compute def-use chains and perform transitive dependence propagation using these chains. Second, our implementation computes control dependencies (in addition to the data dependencies) and uses them when computing transitive dependencies. The IDE formulation from Section 2.1 can be easily extended to capture this generalization. Finally, we use a sparse graph representation of transformers: trivial edges $d \rightarrow d$ are not represented.

Column 1 in part (b) of Table 1 shows the total number of edges in the graph representation of all transformers before any transformer composition or meet is performed. Column 2 shows the total number of such edges after intraprocedural propagation, which starts at each node n that is an entry node or a return site, and computes summary functions ψ^n_m for all m reachable from n along call-free paths. Column 3 shows the total number of edges in the representation of the abstracted transformers $\widehat{\psi}^n_m$. The reduction from column 2 to column 3 eliminates all method-local information that does not directly affect callers or callees of a method. The overall reduction in the number of edges is 53.5%. Part (c) shows similar measurements for the type analysis; here each method has a single summary function. The reduction in the number of edges from column 2

[6] Row `com` includes packages `com` and `COM`; row `sun` includes packages `sun` and `sunw`.

Table 2. Whole-program vs. summary-based analysis: time (sec) and memory (MB)

(a) Program		(b) All Analyses				(c) Dependence Analysis					
Name	*Stmts*	T_{wp}	Δ_T	M_{wp}	Δ_M	T_{wp}	Δ_T	Δ_T^{ub}	M_{wp}	Δ_M	Δ_M^{ub}
`compress`	71729	89.6	52.4%	256.8	30.7%	21.5	86.2%	88.3%	58.1	95.1%	98.2%
`db`	71940	89.8	51.2%	257.2	30.7%	20.7	78.6%	83.2%	58.2	95.0%	98.3%
`jb`	72713	87.9	50.0%	259.3	30.6%	20.6	75.8%	80.6%	59.3	93.8%	96.9%
`raytrace`	74738	92.9	56.6%	262.3	30.3%	21.4	76.6%	79.5%	61.0	91.5%	94.7%
`proxy`	75962	91.4	56.1%	263.5	31.5%	21.4	74.6%	83.9%	61.4	94.9%	98.1%
`jlex`	77134	94.4	43.9%	264.2	30.2%	25.3	60.0%	63.2%	62.2	90.7%	93.8%
`javacup`	78798	97.6	46.0%	269.3	29.6%	24.3	73.7%	75.7%	64.4	88.0%	90.9%
`jess`	79131	96.4	46.6%	271.8	29.4%	23.3	70.3%	70.6%	64.7	87.2%	90.2%
`jack`	81139	103.0	45.4%	270.9	29.5%	25.9	70.3%	73.7%	65.3	86.8%	90.1%
`mpegaudio`	83023	135.8	26.0%	271.3	29.3%	57.9	13.1%	16.6%	65.3	86.4%	90.0%
`rabbit`	90964	100.8	59.7%	287.3	34.0%	23.6	80.3%	82.0%	73.6	94.2%	97.0%
`sablecc`	92171	114.6	44.7%	298.7	27.3%	34.6	61.3%	61.5%	75.9	77.2%	80.3%
`javac`	95498	108.0	43.2%	302.2	27.6%	26.4	63.3%	71.7%	78.8	75.3%	78.3%
`fractal`	106433	110.8	55.0%	315.0	37.1%	25.3	80.3%	84.7%	86.6	94.5%	97.3%
`echo`	110458	117.0	64.2%	321.5	37.7%	30.4	85.3%	85.8%	90.0	94.6%	97.2%
`jtar`	113244	116.4	61.3%	326.8	37.4%	28.8	83.7%	87.6%	92.4	93.0%	95.5%
`jflex`	116938	144.8	50.9%	334.5	35.3%	50.1	61.3%	61.6%	96.1	86.9%	89.5%
`mindterm`	126362	144.9	43.2%	345.9	36.1%	49.7	41.5%	41.6%	102.2	86.4%	89.0%
`muffin`	138140	138.0	51.1%	370.9	38.3%	35.5	56.2%	57.3%	113.4	88.3%	90.6%
`violet`	153895	148.3	66.5%	398.8	43.0%	39.3	83.5%	87.9%	126.8	95.3%	97.6%

(size of ψ, after fixed-point transformer composition and meet) to column 3 (size of $\widehat{\psi}$, after abstracting method-local information) is 55.1%.

Out of all library methods, 25490 (33.0%) are fixed. While fixed methods tend to be smaller and simpler than non-fixed ones, the complete knowledge of their summary functions still has positive effects on the library summary. The instantiation of fixed-method summary functions at calls can be done for 63229 (20.5%) of all library call sites. This instantiation, followed by additional intraprocedural propagation and abstraction, further reduces the number of edges in the representation of summary functions: overall, from column 3 to column 4 of Table 1 part (b), there is 32.1% reduction. Since the type analysis is context-insensitive, instantiation of summary functions at call sites (an inherently context-sensitive operation) is not meaningful for it and was not performed.

Study 2: Summary-based client analysis. The goal of our second study was to measure the cost benefits of summary-based analysis compared to traditional whole-program analysis. Table 2 presents the results of this study on 20 Java programs. Column *Stmts* shows the number of IR statements for all methods reported by whole-program 0-CFA as reachable. Typically, more than 90% of these methods are library methods [13].

We ran two sets of experiments. The first set, shown in part (b) of Table 2, considered the entire set of analyses employed by a Soot user: the IR building, the 0-CFA type analysis interleaved with on-the-fly call graph construction in the Spark module [14], and the dependence analysis (which uses this call graph). This is the complete start-to-finish cost that would have to be paid to obtain dependence information. The second set of experiments, shown in part (c) of Table 2, considered only the dependence analysis, without any Soot-related costs. For each experiment we measured the running time T_{wp} and the peak memory consumption M_{wp} of the whole-program analysis, as well as the corresponding cost reduction Δ_T and Δ_M when using summary-based analyses.

Considering dependence analysis, type analysis, and IR building together, as shown in part (b), the time savings Δ_T are 50.7% and the memory savings Δ_M are 32.8% (average across all 20 programs). A large proportion of these savings is due to Soot-related costs; such savings will be observed for any interprocedural dataflow analysis which uses 0-CFA as a preprocessing step to obtain a program call graph. When considering only the dependence analysis, in part (c), the savings are, on average, 68.8% for Δ_T and 89.8% for Δ_M.

Columns Δ_T^{ub} and Δ_M^{ub} in part (c) show conservative upper bounds on the savings of the summary-based dependence analysis. These measurements were obtained using an artificial summary which contained only summary functions for type analysis, but not for dependence analysis. Thus, the only dependence analysis work was done inside the client code (of course, the resulting solution is unsound). It is impossible to achieve reductions higher than the ones observed with this artificial summary. Comparing columns Δ_T^{ub} and Δ_T, as well as Δ_M^{ub} and Δ_M, it is clear that the savings are very close to this upper bound.

5 Related Work

Various techniques have been used to achieve modularity in static analysis; some of the most relevant approaches are outlined below. A more complete discussion is available in [15], presented from an abstract-interpretation point of view.

Summary functions for interprocedural analysis date back to the functional approach [2], with refinements in [3] for IFDS problems and in [1] for IDE problems. This body of work assumes a procedural language without polymorphic calls; furthermore, there is no separation between client code and library code. A recent generalization [16], which subsumes IFDS and IDE problems, uses conditional micro-transformers to represent and manipulate dataflow functions; it would be interesting to generalize our approach to take advantage of this work.

Our summary-based analyses can be viewed as instances of the theoretical approach presented in [17]. However, this earlier work does not consider (1) type analysis and on-the-fly call graph construction, (2) abstracting away of library-local dataflow facts, or (3) compact graph representation of dataflow functions.

Most analyses that employ summaries perform bottom-up traversal of the call graph, and compute summary functions using the functions computed for the visited callees; examples include [18,19,20,21,22,23,24,25,26]. In [27], libraries are pre-analyzed but the computation of summary functions cannot be performed in the presence of callbacks. Some techniques compute summary information for a software component independently of the callers and callees of that component. One particular technique is a modular approach which computes partial analysis results for each component, combines the results for all components in the program, and then performs the rest of the analysis; examples include [28,29,30,31,32,33]. There have also been proposals for employing summary information provided by the analysis user, as in [34,35,36]. Finally, certain approaches analyze a software component when there is no available information about the surrounding environment, using conservative assumptions about unknown external code (e.g., [37,38,27,39,40,41,42,43,44,45]).

6 Conclusions and Future Work

Summary-based analysis shows promising potential for improving the scalability of interprocedural analysis in the presence of large object-oriented libraries. Our results indicate that summary generation can have practical cost and can produce a small summary file, and most importantly, the analysis of client code becomes substantially cheaper. Future work will investigate other IDE analyses, as well as a standardized API for storing and retrieving summary information.

References

1. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Comp. Sci. 167, 131–170 (1996)
2. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–234 (1981)
3. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL, pp. 49–61 (1995)
4. Reps, T., Sagiv, M., Horwitz, S.: Interprocedural dataflow analysis via graph reachability. Technical Report DIKU-TR94-14, U. Copenhagen (1994)
5. Rountev, A., Connell, B.H.: Object naming analysis for reverse-engineered sequence diagrams. In: ICSE, pp. 254–263 (2005)
6. Grove, D., Chambers, C.: A framework for call graph construction algorithms. TOPLAS 23(6), 685–746 (2001)
7. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: OOPSLA, pp. 281–293 (2000)
8. Heintze, N.: Set Based Program Analysis. PhD thesis, CMU (1992)
9. Callahan, D.: The program summary graph and flow-sensitive interprocedural data flow analysis. In: PLDI, pp. 47–56 (1988)
10. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. TOPLAS 12(1), 26–60 (1990)
11. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural dataflow analysis in the presence of large libraries. Technical Report CISRC-TR01, Ohio State U (2006)
12. Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000 and ETAPS 2000. LNCS, vol. 1781, Springer, Heidelberg (2000)
13. Sharp, M.: Static Analyses for Java in the Presence of Distributed Components and Large Libraries. PhD thesis, Ohio State University (2007)
14. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using Spark. In: Hedin, G. (ed.) CC 2003 and ETAPS 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)

15. Cousot, P., Cousot, R.: Modular static program analysis. In: Horspool, R.N. (ed.) CC 2002 and ETAPS 2002. LNCS, vol. 2304, pp. 159–178. Springer, Heidelberg (2002)
16. Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: POPL (2008)
17. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural dataflow analysis in the presence of large libraries. In: Mycroft, A., Zeller, A. (eds.) CC 2006 and ETAPS 2006. LNCS, vol. 3923, pp. 2–16. Springer, Heidelberg (2006)
18. Chatterjee, R., Ryder, B.G., Landi, W.: Relevant context inference. In: POPL, pp. 133–146 (1999)
19. Choi, J., Gupta, M., Serrano, M., Sreedhar, V., Midkiff, S.: Escape analysis for Java. In: OOPSLA, pp. 1–19 (1999)
20. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: OOPSLA, pp. 187–206 (1999)
21. Cheng, B., Hwu, W.: Modular interprocedural pointer analysis using access paths. In: PLDI, pp. 57–69 (2000)
22. Ruf, E.: Effective synchronization removal for Java. In: PLDI, pp. 208–218 (2000)
23. Foster, J., Fähndrich, M., Aiken, A.: Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 175–198. Springer, Heidelberg (2000)
24. Liang, D., Harrold, M.J.: Efficient computation of parameterized pointer information for interprocedural analyses. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 279–298. Springer, Heidelberg (2001)
25. Triantafyllis, S., Bridges, M., Raman, E., Ottoni, G., August, D.: A framework for unrestricted whole-program optimization. In: PLDI, pp. 61–71 (2006)
26. Cherem, S., Rugina, R.: A practical effect and escape analysis for building lightweight method summaries. In: Krishnamurthi, S., Odersky, M. (eds.) CC 2007. LNCS, vol. 4420, pp. 172–186. Springer, Heidelberg (2007)
27. Chatterjee, R., Ryder, B.G.: Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-433, Rutgers University (2001)
28. Oxhøj, N., Palsberg, J., Schwartzbach, M.: Making Type Inference Practical. In: Lehrmann Madsen, O. (ed.) ECOOP 1992. LNCS, vol. 615, pp. 329–349. Springer, Heidelberg (1992)
29. Codish, M., Debray, S., Giacobazzi, R.: Compositional analysis of modular logic programs. In: POPL, pp. 451–464 (1993)
30. Flanagan, C., Felleisen, M.: Componential set-based analysis. TOPLAS 21(2), 370–416 (1999)
31. Das, M.: Unification-based pointer analysis with directional assignments. In: PLDI, pp. 35–46 (2000)
32. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA. In: PLDI, pp. 254–263 (2001)
33. Rountev, A., Ryder, B.G.: Points-to and side-effect analyses for programs built with precompiled libraries. In: Wilhelm, R. (ed.) CC 2001 and ETAPS 2001. LNCS, vol. 2027, pp. 20–36. Springer, Heidelberg (2001)
34. Dwyer, M.: Modular flow analysis of concurrent software. In: ASE, pp. 264–273 (1997)
35. Guyer, S., Lin, C.: Optimizing the use of high performance software libraries. In: Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J.F., Pugh, B., Tseng, C.-W. (eds.) LCPC 2000. LNCS, vol. 2017, pp. 227–243. Springer, Heidelberg (2001)

36. Rugina, R., Rinard, M.: Design-driven compilation. In: Wilhelm, R. (ed.) CC 2001 and ETAPS 2001. LNCS, vol. 2027, pp. 150–164. Springer, Heidelberg (2001)
37. Harrold, M.J., Rothermel, G.: Separate computation of alias information for reuse. TSE 22(7), 442–460 (1996)
38. Rountev, A., Ryder, B.G., Landi, W.: Data-flow analysis of program fragments. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, pp. 235–252. Springer, Heidelberg (1999)
39. Sreedhar, V., Burke, M., Choi, J.: A framework for interprocedural optimization in the presence of dynamic class loading. In: PLDI, pp. 196–207 (2000)
40. Ghemawat, S., Randall, K., Scales, D.: Field analysis: Getting useful and low-cost interprocedural information. In: PLDI, pp. 334–344 (2000)
41. Vivien, F., Rinard, M.: Incrementalized pointer and escape analysis. In: PLDI, pp. 35–46 (2001)
42. Tip, F., Sweeney, P., Laffra, C., Eisma, A., Streeter, D.: Practical extraction techniques for Java. TOPLAS 24(6), 625–666 (2002)
43. Rountev, A., Milanova, A., Ryder, B.G.: Fragment class analysis for testing of polymorphism in Java software. TSE 30(6), 372–387 (2004)
44. Rountev, A.: Precise identification of side-effect-free methods in Java. In: ICSM, pp. 82–91 (2004)
45. Xue, J., Nguyen, P.H.: Completeness analysis for incomplete object-oriented programs. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 271–286. Springer, Heidelberg (2005)