2005 International Conference on Compiler Construction, Pages 271 -- 286.

Completeness Analysis for Incomplete Object-Oriented Programs

Jingling Xue and Phung Hua Nguyen

Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales
NSW 2032, Sydney, Australia

Abstract. We introduce a new approach, called completeness analysis, to computing points-to sets for incomplete Java programs such as library modules or applications in the presence of dynamic class loading. One distinctive feature of this work is that the access and modification properties of fields are taken into account. By combining with a whole-program points-to analysis, completeness analysis yields not only the required points-to sets but also determines which points-to sets and call sites are complete (when the pointed-to objects and target methods are statically resolvable) or not. Such a compositional approach yields more precise points-to sets than those computed by the points-to analysis alone. In addition, our technique also determines (for the first time) which objects may be incompletely detectable, i.e., may be missing in some statically computed points-to sets. We provide experimental evidence to demonstrate that better analysis precision in benchmarks is obtained when the field access and modification properties are exploited. In particular, we are able to find significantly more complete and mono call sites in an incomplete program, which is useful in devirtualisation and inlining. Our analysis is simple since it is flow- and context-insensitive and achieves these improvements at reasonably small analysis costs.

1 Introduction

For object-oriented languages such as Java and C♯, points-to analysis finds many applications in compilers and software engineering. However, most existing points-to analysis methods [9, 13] require the whole program to be available. Their inadequacies are being recognised as modern applications rely increasingly more on component programming and software libraries. When applied to components or library modules alone, whole-program methods may yield incomplete points-to sets (i.e. the ones that may not contain all the pointed-to objects at run time), and consequently, incomplete call sites (i.e., the ones whose sets of target methods resolved statically may not contain all methods invoked at run time). In addition, these methods cannot tell whether or not a points-to set or call site is complete or not, making their results hardly useful. The situation is further aggravated by some features in Java and C♯. Due to the presence of

dynamic class loading and/or native methods, an application written in these languages may be incomplete in its entirety at analysis time.

This work addresses the problem of computing the points-to sets for an incomplete object-oriented program and also determining at the same time whether these sets and call sites are complete or not. There are some existing methods that are proposed (intentionally or otherwise) to solve this problem. The extant analysis (EA) [16] is developed mainly to support devirtualisation and inlining. For that reason, EA takes as input the points-to sets for a program and produces as output the set of runtime types in the analysed program for each reference. As a by-product of this process, one can deduce that the point-to set of a reference is complete iff all its runtime types are guaranteed to be known in the analysed program. Several techniques reported in [20, 21] combine points-to analysis and escape analysis [2, 4]. Their results can be used to determine the completeness of points-to sets and call sites.

A common problem with these earlier efforts is that they do not exploit the access and modification properties of fields although similar properties for methods are considered somewhat. As a result, they do not provide sufficient precision about the completeness of points-to sets and call sites. Given an incomplete program, applying whole-program analysis alone is inadequate because the reciprocal modification effects between the program and unknown code are not accounted for. In the presence of unknown code, some points-to sets are inevitably incomplete. As a result, some alias relations cannot be determined based on the computed points-to sets alone. In order to obtain better analysis precision, we believe that the reciprocal modification effects between an incomplete program (i.e., the analysed code) and the unknown (i.e., unanalysed) code must be taken into account as accurately as possible. In this work, we exploit the access and modification properties of fields (static or instance) to improve analysis precision. To substantiate this claim, we present a flow- and context-insensitive analysis, called *completeness analysis (CA)*, for incomplete Java programs and demonstrate its benefits in increasing analysis precision in benchmark programs.

The contributions of this work are summarised as follows:

- We present a new compositional approach to conducting points-to analysis for incomplete Java program. By composing with a whole-program points-to analysis, we obtain not only the points-to sets for the program but also determine which point-to sets and call sites are complete or not.
- We exploit the access and modification properties of fields in our analysis. To the best of our knowledge, this is the first work using these properties to compute the points-to sets for incomplete object-oriented programs. As a result, our points-to sets are more precise than those computed by the points-to analysis alone.
- We introduce for the first time the notion of object detectability, which played an important role in completeness analysis. A compile-time object created in an analysed program is *incompletely detectable* if it may be pointed to by a reference at run time but is missing in its statically computed points-to set and *(completely) detectable* otherwise. Since the access and modification

properties of fields are considered, object detectability is different from object reachability used in escape analysis [2, 4, 20, 21]. If an object escapes from a function or thread, so are all the reachable objects. If an object is incompletely detectable, a directly or indirectly reachable object may be either completely or incompletely detectable.
- We have implemented our complete analysis in Soot [19]. When the access and modification properties of fields are exploited, we obtain better analysis precision in benchmarks, in particular, significantly more mono call sites, i.e., more opportunities for devirtualisation and inlining.

The rest of this paper is organised as follows. Section 2 introduces the language model used. Section 3 presents our completeness analysis. Section 4 discusses our experimental results. Section 5 reviews the related work. Section 6 concludes the paper and discusses some future work.

2 Language Model

For simplicity, we describe our approach for a subset of Java with the features most relevant to points-to analysis. The points-to analysis for a Java program is carried out in an intermediate representation (IR) of the program. As our approach is flow- and context-insensitive, IR consists of only the seven kinds of statements listed in Table 1. Furthermore, the features such as multi-inheritance interface and multi-threading do not pose any problems. Of the seven statements, the first and last are explained below and the other five are self-explanatory.

Syntax	Semantics
$\ell = \texttt{new}\ C$	Object Creation
$\ell = r$	Assignment
$\ell = C.f$	Static Field Load
$C.f = r$	Static Field Store
$\ell = r.f$	Instance Field Load
$\ell.f = r$	Instance Field Store
$op(a_0, \ldots, a_n, \ell)$	Call Site

Table 1. Instruction set for the IR on which points-to analysis is conducted, where ℓ and r are reference variables, f a field, op a method name, and C a class name.

In Java, objects can be created either explicitly via `new` or implicitly, say, via a Java reflection method `newInstance`. In the latter case, the object creation statement can be replaced by $\ell = \texttt{new}\ C$ if C is detected statically to be the class name of the implicitly created object. Otherwise, the object creation statement used is $\ell = \texttt{new}\ \mathit{Unknown}$, where *Unknown* can be any class in the analysed program or any new class that may be loaded dynamically at run time.

For notational convenience, each method is denoted $op(p_0, \ldots, p_n, r)$, where $p_0, \ldots, p_n$ are its $n+1$ formal input parameters and r its formal output parameter. As is clear in Table 1, the return statements in a method are not explicitly

represented. Instead, every return statement in a method is replaced by an assignment to the formal output parameter of that method. Correspondingly, a call site has the form $op(a_0, \ldots, a_n, \ell)$, where $a_0, \ldots, a_n$ are the $n+1$ actual input parameters and ℓ is the actual output parameter. If op is an instance method, then a_0 denotes the receiver of the call. Otherwise, op is static and can be conveniently regarded as an instance method if a_0 is set to be the name of the class in which op is declared. In Java, parameter passing is call by value.

Accesses to arrays are handled similarly to instance field accesses by introducing a special field, say, *sf*. We do not distinguish accesses to different components of an array. For example, `x[i]` and `x[j]` are both represented by `x.`*sf*.

The term *fixed call site* is used to denote (1) an `invokestatic`, (2) an `invokespecial`, (3) a call site whose (unique) target is declared to be `final` or in a `final` class, or (4) a sealed call site [22]. All other kinds of call sites are called *non-fixed*. In a fixed call site, all target methods that may be invoked are known at compile time. This is obvious in the first three cases. The target methods of a sealed call site are confined to be in the underlying sealed package.

The term *reference* is used to denote all kinds of accesses such as variable accesses, static field accesses and instance field accesses.

The semantics of the following field and method modifiers in Java are exploited in our analysis: `private`, `protected`, `public` and `final`. In the absence of the first three modifiers, the *default* (i.e., *package*) access is assumed.

3 Completeness Analysis for Incomplete Programs

The whole-program points-to analysis requires the entire program, i.e. all its classes and methods, to be available at analysis time. An incomplete program includes only a subset of these classes. In addition, some methods in a class (e.g., native methods) may be unavailable to participate in static analysis.

3.1 Incomplete Programs

Definition 1 An incomplete program, F, is a triple $F = \langle \mathbb{L}_F, \mathbb{M}_F, \mathbb{F}_F \rangle$, where $\mathbb{L}_F$ is the set of classes in F, $\mathbb{M}_F$ the set of methods in $\mathbb{L}_F$ and $\mathbb{F}_F$ the set of fields declared in $\mathbb{L}_F$ such that (1) all classes in $\mathbb{L}_F$ except the root class `java.lang.Object` have all their superclasses in $\mathbb{L}_F$, (2) there is not a reference in $\mathbb{M}_F$ whose type is not in $\mathbb{L}_F$, (3) there is not a read/write to a field not in $\mathbb{F}_F$, and finally, (4) there is not an access (i.e., a call) to a method not in $\mathbb{M}_F$.

According to this definition, our work is applicable to library modules or applications supporting native methods and dynamic class loading.

We use $IM_F \subseteq \mathbb{M}_F$ to denote the set of all *analysed* methods whose code is available for static analysis and define $EM_F = \mathbb{M}_F \setminus IM_F$. Although the methods in EM_F are *unanalysed*, their signatures are always available by Definition 1.

$\mathbb{S}_F$ denotes the set of statements and $\emptyset_F$ the set of compile-time objects created in IM_F, respectively. Let V_F be the set of references in $\mathbb{S}_F$.

Let $\mathbb{U}_F$ symbolise the unknown code, i.e., the code in EM_F and outside F.

Let us define the (in)accessibility of the fields and methods in IM_F with respect to $\mathbb{U}_F$. A method in IM_F is *accessible* if it can be invoked in $\mathbb{U}_F$ and *inaccessible* otherwise. A field in F is *accessible* if it can be accessed by some statements in $\mathbb{U}_F$ and *inaccessible* otherwise. For example, a `private` field is inaccessible if there does not exist any unanalysed method in the class in which the field is declared. $\mathbb{F}_F^i$ denotes the set of all inaccessible fields in F.

Let $\mathbb{F}_F^f$ be the set of all `final` fields in F. In Java, `final` fields are initialised only once in the initialisers or constructors in analysed methods.

3.2 Applying Whole-Program Analysis to F

We use an Andersen-style analysis [1, 9] because of its reasonable precision and efficiency. Let $PT_F(\ell)$ be the points-to-set of a reference ℓ in V_F. By convention, we assume that $PT_F(\ell) = \emptyset$ if $\ell \notin V_F$. The points-to analysis requires an approximation of the call graph for F. A call graph is the relation $\mathbb{C}_F \subseteq \mathbb{S}_F \times \mathbb{M}_F$ such that $(s, op) \in \mathbb{C}_F$ iff s is a call site statement and op is a method that may be a target of the call site. We use CHA (Class Hierarchy Analysis) [5] to construct such a call graph. A more precise alternative [17] is to construct the call graph on-the-fly as the points-to sets of call site receivers are being computed. However, the improved precision may not justify the computational cost [9].

An analysed program F consists of only the seven different kinds of statements given in Table 1. The rules for computing the points-to sets of F are:

Rule P1 *If* $\exists\ s : [\ell = \texttt{new}\ C] \in \mathbb{S}_F$, *then* $\{o_s\} \subseteq PT_F(\ell)$.

Rule P2 *If* $\exists\ s : [\ell = r] \in \mathbb{S}_F$, *then* $PT_F(r) \subseteq PT_F(\ell)$.

Rule P3 *If* $\exists\ s : [\ell = C.f] \in \mathbb{S}_F$ *then* $PT_F(C.f) \subseteq PT_F(\ell)$.

Rule P4 *If* $\exists\ s : [C.f = r] \in \mathbb{S}_F$ *then* $PT_F(r) \subseteq PT_F(C.f)$.

Rule P5 *If* $\exists\ s : [\ell = r.f] \in \mathbb{S}_F$ *then* $PT_F(r.f) \subseteq PT_F(\ell)$.

Rule P6 *If* $\exists\ s : [\ell.f = r] \in \mathbb{S}_F \wedge\ \exists\ \ell'.f \in V_F$ *s.t.* $PT_F(\ell) \cap PT_F(\ell') \neq \emptyset$, *i.e.,* ℓ *and* ℓ' *are aliases (with nonempty points-to sets), then* $PT_F(r) \subseteq PT_F(\ell'.f)$

Rule P7 *If* $\exists\ s : [op(a_0, \ldots, a_n, \ell)] \in \mathbb{S}_F \wedge \exists\ op(p_0, \ldots, p_n, r) \in \mathbb{M}_F$ *s.t.* $(s, op) \in \mathbb{C}_F$, *then* $PT_F(a_0) \subseteq PT_F(p_0), \ldots, PT_F(a_n) \subseteq PT_F(p_n)$ *and* $PT_F(r) \subseteq PT_F(\ell)$.

The points-to analysis for F consists of solving the constraints for all its statements to determine the points-to sets of all references in the program.

3.3 Inadequacies of Whole-Program Analysis

When a whole-program points-to analysis technique is applied to an incomplete program F, the following two assumptions are conventionally made:

- All methods in EM_F are considered to have an empty body.

– The points-to sets of all formal input parameters of all methods that are accessible (in the unknown code $\mathbb{U}_F$) are initialised to be empty.

Due to the lack of knowledge about $\mathbb{U}_F$, a reference (inside F or outside) is said to be incomplete if its statically computed points-to set may not contain some object pointed to at run time – such an object can be created either inside F or outside. In addition, a call site in F may be incomplete when the set of target methods that are statically resolved may not contain a method that is invoked at run time – such a method is declared in the unknown code outside F. (The methods in EM_F cannot be missing since their declarations are available.)

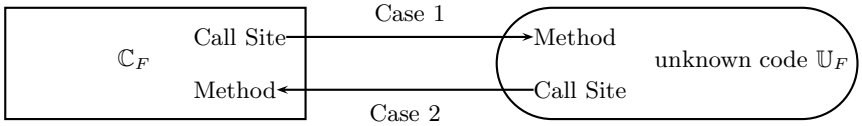

Fig. 1. Two kinds of troublesome missing caller-callee relations in $\mathbb{C}_F$.

There are several reasons why a points-to sets can be incomplete. First, while the call graph $\mathbb{C}_F$ constructed using CHA over-approximates all caller-callee relations within F, some relations that happen during program execution can be missing if F is incomplete. As illustrated in Figure 1, there are two kinds of troublesome missing caller-callee relations in $\mathbb{C}_F$. In Case 1, the objects passed to an unknown method in $\mathbb{U}_F$ can be used in an unknown way. In addition, the type of the returned object is unknown. In Case 2, the situation is reversed. Second, static fields can be accessed by the unknown code $\mathbb{U}_F$. Third, due to the first two reasons, instance fields can also be accessed indirectly in $\mathbb{U}_F$.

Finally, if some points-to sets are incomplete, the alias relations captured by Rule P6 may also be incomplete. This is because ℓ and ℓ' can still be aliases even if $PT_F(\ell) \cap PT_F(\ell') = \emptyset$. So Rule P6 needs to be augmented later by Rule C6.

3.4 Completeness Analysis (CA)

In the previous section, we argued that whole-program analysis is inadequate if the analysed program is incomplete. In this section, we present a technique, called *completeness analysis (CA)*, to detect which points-to sets and call sites may be incomplete and which compile-time objects may be missing (i.e., incompletely detectable) in some points-to sets. Our approach is compositional. By combining with a whole-program points-to analysis technique, our completeness analysis also produces at the same time the points-to sets with better precision (Theorem 1).

Our approach is flow- and context-insensitive. If an analysed program F is not incomplete, Rules P1 – P7 given in Section 3.2 are sufficient. Otherwise, we rely on Rules C1 – C9, which are introduced in Section 3.4, to carry out the so-called

completeness analysis. By applying both sets of rules to an incomplete program F and solving the derived constraints for all the statements in F iteratively, the desired results are found as a fixed point to these constraints.

Completeness; Detectability; Aliases Let $\mathcal{P}(F)$ be the set of all possible programs that include F as a subset. Let $W \in \mathcal{P}(F)$. Let $\mathbb{M}_W$ be the set of methods in the program W and V_W the set of the reference variables in $\mathbb{M}_W$. The following two concepts are defined conceptually (but not physically constructed). Let $PT_W(\ell)$ be the points-to set of $\ell \in V_W$ observed during program execution. Let $\mathbb{C}_W$ be the call graph of W also observed during program execution.

To determine which objects may be missing in some points-to sets and which references may have such points-to sets, the following notions are introduced.

Definition 2 An object $o \in \emptyset_F$ is *incompletely detectable* if $\exists\ W \in \mathcal{P}(F)$ such that

- $\exists\ \ell \in V_F : o \in PT_W(\ell) \setminus PT_F(\ell)$, or
- $\exists\ \ell \in V_W \setminus V_F : o \in PT_W(\ell)$.

and *(completely) detectable* otherwise. A reference $\ell \in V_F$ is *incomplete* if $\exists\ W \in \mathcal{P}(F)$ such that $PT_W(\ell) \setminus PT_F(\ell) \neq \emptyset$ and *complete* otherwise. Every reference $\ell \in V_W \setminus V_F$ in every program $W \in \mathcal{P}(F)$ is *incomplete*, i.e., every reference $\ell \notin V_F$ is *incomplete*.

Recall the convention that $PT_F(\ell) = \emptyset$ if $\ell \notin V_F$. Therefore, an object in $\emptyset_F$ is incompletely detectable if it may be pointed to by a reference (inside F or outside) at run time but is missing in its statically computed points-to set. A reference (inside F or outside) is incomplete if it may point to an object at run time such that the object is missing in its points-to set. Note that a complete reference may or may not include incompletely detectable objects in its points-to set.

To find out the missing caller-callee pairs (s, op) in the call graph $\mathbb{C}_F \subseteq \mathbb{S}_F \times \mathbb{M}_F$ (built statically), the notion of incomplete call site is introduced below. Essentially, a call site in F is incomplete if its set of statically resolved target methods does not include a method that may be invoked at run time.

Definition 3 If $\exists\ W \in \mathcal{P}(F) \wedge \exists\ s \in \mathbb{S}_F \wedge \exists\ op \in \mathbb{M}_W$ s.t. $(s, op) \in (\mathbb{C}_W \setminus \mathbb{C}_F)$, then s is said to be an *incomplete* call site and *complete* otherwise.

We discussed earlier that the points-to sets computed for an incomplete program is insufficient to determine all the alias relations. The notions of reference completeness and object detectability are used below to provide an over-approximation of all missing aliases.

Definition 4 Let ℓ and ℓ' be two references in $W \in \mathcal{P}(F)$. Both are aliases, denoted *C-Alias*(ℓ, ℓ'), if ℓ is incomplete and ℓ' is either incomplete or complete with its points-to set containing at least one incompletely detectable object, or vice versa.

According to the above definition, *C-Alias*(ℓ, ℓ) is true iff ℓ is incomplete.

Rules Let O^i, R^i and S^i be the set of incompletely detectable objects, incomplete references and incomplete call sites in an incomplete program F, respectively. Rules C1 – C9 for computing these sets are introduced below. In each rule, the statements or field accesses (among others) to which the rule is applied is indicated. There are no extra rules for object creation statements $\ell = \texttt{new}\ C$, where C is a known class in F, and for assignment statements since they are covered by Rules P1 and P2.

Rule C1 ($\mathbf{s}$:$[\ell = \texttt{new}\ \mathit{Unknown}]$) $o_s \in O^i$, where o_s is the object created at s.

In the following two rules for static fields, the corresponding access and modification properties are used to determine whether they are applicable.

Rule C2 ($C.f = r$) If $f \notin \mathbb{F}^i_F$, then $PT_F(r) \subseteq O^i$.

As $f \notin \mathbb{F}^i_F$, there may exist a static load $\ell = C.f$ in the unknown code $\mathbb{U}_F$, where $\ell \notin V_F$, i.e., $\ell \in V_W \setminus V_F$ for some $W \in \mathcal{P}(F)$. By Rule P4, $C.f$ will point to the objects pointed to by r. But these objects may be assigned to ℓ in $\mathbb{U}_F$. By Definition 2, the objects that r points to are marked as incompletely detectable.

Rule C3 ($C.f$) If $f \notin (\mathbb{F}^i_F \cup \mathbb{F}^f_F)$, then $C.f \in R^i$.

If $f \notin (\mathbb{F}^i_F \cup \mathbb{F}^f_F)$, there may exist a static store $C.f = r$ in the unknown code $\mathbb{U}_F$, where $r \notin V_F$. The objects pointed to by r may not appear in the points-to set of $C.f$ when F is analysed. By Definition 2, $C.f$ is incomplete.

If the access and modification properties of instance fields were ignored, the following two rules would be sufficient for handling instance field loads and stores. We discuss them first in order to motivate Rules C4 – C6 used in our analysis.

Rule S1 ($\ell.f$) *If* $\ell \in R^i \vee PT_F(\ell) \cap O^i \neq \emptyset$, *then* $\ell.f \in R^i$.

Rule S2 ($\ell.f = r$) *If* $\ell \in R^i \vee PT_F(\ell) \cap O^i \neq \emptyset$, *then* $PT_F(r) \subseteq O^i$.

If the access and modification properties of f are ignored, we must assume conservatively the existence of an instance field access $\ell'.f$ in the unknown code $\mathbb{U}_F$, where $\ell' \notin V_F$. If $\ell \in R^i \vee PT_F(\ell) \cap O^i \neq \emptyset$, then *C-Alias*$(\ell, \ell')$ may hold, i.e., ℓ and ℓ' are potentially aliases. However, when F is analysed as a whole program, $PT_F(\ell) \cap PT_F(\ell') = \emptyset$ is possible. In this case, Rule P6 will not be applied. We need Rule S1 for the following reason. If there exists a store $\ell'.f = r$ in $\mathbb{U}_F$, the pointed-to objects by r may not belong to the points-to set of $\ell.f$. By Definition 2, $\ell.f$ is incomplete. We need Rule S2, because if there exists a load into $\ell'.f$ in $\mathbb{U}_F$, then all the objects pointed by r may be incompletely detectable.

By exploiting the access and modification properties of instance fields, we have relaxed the assumption about the always existence of of a field access $\ell'.f$ in the unknown code $\mathbb{U}_F$. Rules S1 and S2 are replaced by Rules C4 – C6.

Rule C4 ($\ell.f$) If $f \notin (\mathbb{F}^i_F \cup \mathbb{F}^f_F) \wedge (\ell \in R^i \vee PT_F(\ell) \cap O^i \neq \emptyset)$, then $\ell.f \in R^i$.

Rule C5 ($\ell.f = r$) If $f \notin \mathbb{F}^i_F \wedge (\ell \in R^i \vee PT_F(\ell) \cap O^i \neq \emptyset)$, then $PT_F(r) \subseteq O^i$.

Rule C6 ($\ell.f = r$) If $f \in (\mathbb{F}^i_F \cup \mathbb{F}^f_F) \wedge \exists\ \ell'.f \in V_F$ s.t. *C-Alias*(ℓ, ℓ') holds, then $PT_F(r) \subseteq PT_F(\ell'.f)$

Rule C4 is refined from Rule S1 since it is applied only when there may be an instance field store $\ell'.f = r$ in the unknown code $\mathbb{U}_F$. Rule C5 is refined from Rule S2 since it is applied only when there may be an instance field access $\ell'.f$ in $\mathbb{U}_F$. In this case, the objects pointed to by $\ell'.f$ may be incompletely detectable since they can be assigned to a reference in $\mathbb{U}_F$. Rule C6 is applied only when $\ell'.f \in V_F$ holds. In this case, there cannot be any store of the form $\ell'.f = r$ in $\mathbb{U}_F$. If ℓ and ℓ' are aliases, then a store to $\ell'.f$ is also a store to $\ell.f$ in disguise. Rule C6 is the rule in completeness analysis that enables better points-to sets to be computed. In this rule, $\ell = \ell'$ is possible. So the rule will be applied if ℓ is incomplete, since by Definition 4, *C-Alias*(ℓ, ℓ) is true iff ℓ is incomplete.

Rule C7 ($s\!:\![op(a_0, \ldots, a_n, \ell)]$ **for Case 1 in Figure 1)**. There are three parts:

1. Suppose s is a non-fixed call site. If $a_0 \in R^i \vee (PT_F(a_0)$ includes an instance of *Unknown*), then $s \in S^i$.
2. If $s \in S^i$, then $PT_F(a_1), \ldots, PT_F(a_n) \subseteq O^i$ and $\ell \in R^i$.
3. If there exists $(s, op) \in \mathbb{C}_F$ such that $op \in EM_F$, then
   (a) $PT_F(a_1), \ldots, PT_F(a_n) \subseteq O^i$ and $\ell \in R^i$, and
   (b) $A_0 \subseteq O^i$, where A_0 is the set of all receiver objects in $PT_F(a_0)$ on which op is invoked at s (A_0 can be statically determined from $PT_F(a_0)$).

Part 1 determines whether a call site is incomplete or not. As discussed in Section 2, a fixed call site is complete since its set of target methods can be statically resolved. Under the stated conditions, an overriding method in a class outside F may be invokable at s. Such a caller-callee relation is not available in the call graph $\mathbb{C}_F$. Thus, s is incomplete by Definition 3. Clearly, s is incomplete if a_0 is incomplete or complete but may point to an instance of *Unknown*.

Part 2 applies to a call site at which an unknown method op outside F may be invoked. The objects pointed to by $a_1, \ldots, a_n$ may be missing in the points-to sets of the corresponding formal input parameters of op, and thus, are incompletely detectable. The receiver a_0 is excluded since it is the incompleteness of a_0 rather than the nature of its pointed-to objects that causes op to be invoked at s. The actual output parameter ℓ is incomplete since its points-to set may not include the object returned by the unknown method op.

Part 3 applies to a call site at which an unknown method op in EM_F may be invoked. Therefore, Part 3(a) is exactly the same as Part 2. In Part 3(b), the receiver objects that cause op to be invoked are marked as incompletely detectable since they may be assigned to some unknown references in op.

Rule C8 ($op(p_0, p_1, \ldots, p_n, r) \in \mathbb{M}_F$ **for Case 2 in Figure 1)**. If op is accessible (i.e., invokable) in $\mathbb{U}_F$, then $p_0, , \ldots, p_n \in R^i$ and $PT_F(r) \subseteq O^i$.

If op is accessible, there may exist a call site $s : [op(a_0, \ldots, a_n, \ell)]$ in $\mathbb{U}_F$. The effect of the assignments due to parameter and result passing cannot be considered when F is analysed. Thus, the points-to sets of p_i and ℓ cannot contain the objects in the points-to sets of a_i and r, respectively. Hence, $p_0, \ldots, p_n$ are incomplete and all objects pointed to by r are incompletely detectable.

Rule C9 ($PT_F(r) \subseteq PT_F(r')$ Created by Rules P1 – P7 and C6) If $r \in R^i$, then $r' \in R^i$.

The incompleteness of points-to sets is propagated during the points-to analysis. If r is incomplete, a missing object in its points-to set is also missing in the points-to set of r' when $PT_F(r) \subseteq PT_F(r')$. So r' is incomplete by Definition 2.

As explained in Section 3.4, we combine Rules P1 – P7 and C1 – C9 to compute not only the points-to sets of all references but also the information about the completeness of references and call sites as well as object detectability.

Thanks to Rule C6, such a compositional approach enables better points-to sets to be computed. Essentially, a reference may be accurately identified as being complete even though it is incomplete if Rule C6 is not used.

Theorem 1. *Let r be a reference in F. Let $PT_F(r)$ be the points-to set of r computed according to CA. Let $PT'_F(r)$ be computed according to CA′, i.e., a version of CA in which C4 – C6 are replaced by S1 and S2. The following two statements are true: (a) if r is complete in CA′, then r is also complete in CA, and in addition, $PT_F(r) = PT'_F(r)$, (b) if r is incomplete in CA′, then r may be complete or incomplete in CA, and in addition, $PT_F(r) \supseteq PT'_F(r)$.*

4 Experiments

In this section, we provide experimental evidence that completeness analysis can yield better precision when the field access and modification properties are exploited. We have implemented our completeness analysis in Soot [19], a bytecode to bytecode optimiser. In Soot, only whole-program analyses and optimisations are supported. A preprocessing translator converts Java bytecode into a three-address IR called *Jimple*. The points-to sets for an analysed program are computed using the points-to analysis pass in Soot [9]. We have implemented our completeness analysis by composing it with this existing points-to analysis.

Due to the space limitation, we discuss briefly how we have handled some other Java language features not present in Table 1. Java exceptions are dealt with as follows. All formal input parameters of a `catch` statement are initialised to be incomplete. All objects that may be thrown by a `throw` statement are marked as incompletely detectable objects. Reflection methods are treated as native ones with some extra rules. For example, all fields that may be accessed by `get` or `set` are considered as accessible fields. We do not address the Java class reloading since it may potentially modify code on-the-fly and so could affect our assumptions about the analysed program. We also assume that all native methods respect the access and modification properties of fields and methods.

In our experiments, three approaches are compared: CA, CA0 and EA. CA denotes our complete analysis technique. CA0 is the version of CA when the

field modifiers are ignored. Precisely, the following changes are made to our rules. Rules C4 – C6 are replaced by Rules S1 and S2 and Rule C2 – C3 by:

Rule S3 $(C.f = r)$ $PT_F(r) \subseteq O^i$.

Rule S4 $(C.f)$ If $f \notin \mathbb{F}_F^f$, then $C.f \in R^i$.

The extant analysis[1] (EA) [16] can be used for completeness analysis even though it was originally designed for inlining and devirtualisation. EA is chosen because it is applicable to incomplete Java programs and can also be carried out based on the same kind of points-to analysis, i.e., flow- and context-insensitive Andersen-style points-to analysis. In EA, an extant reference is complete while a non-extant reference is incomplete. A call site is complete if its receiver is complete and incomplete otherwise. EA cannot handle object detectability since their extant or non-extant objects can be completely or incompletely detectable.

4.1 Benchmarks

Table 2 gives some statistics about the 12 benchmarks used in our experiments. The first seven are from SPECjvm98, `jbb` is from SPECjbb2000, `jlex` is a Java scanner generator from Princeton University, `jtar` is GNU's `tar` ported to Java (version 1.21), `jtb` is a Java tree builder from Purdue University (version 1.2.2), and finally, `soot` (version 2.0.1) is the Java bytecode-to-bytecode optimiser [19], in which our completeness analysis is implemented.

In our experiments, the analysed program for a benchmark consists of all classes in the application and the classes in Java library reachable statically from the application. The analysis starts with the methods in these classes that may be invoked from outside and continues to analyse the methods that may be reachable statically from these methods. All packages in a benchmark are assumed to be sealed [18]. For each benchmark (including application and library code), Columns 2 – 4 give the total number of its classes, methods and fields and Columns 5 – 8 summarise the access and modification information about its fields. These statistics show convincingly the existence of opportunities for completeness analysis to exploit the field modifiers for better analysis precision.

4.2 Analysis Precision

Table 3 compares CA, CA0 and EA in finding complete points-to sets, call sites and detectable objects in the application part of a benchmark. In all the benchmarks, CA is more precise than CA0, which is more precise than EA.

CA performs better than CA0 because CA fully exploits the field modifiers in Rules C2 – C6 while CA0 considers only the `final` modifier in Rule S4. As shown in Table 2, a benchmark typically has a significant number of fields that are inaccessible by the unknown code and/or that embrace the `final` modifier.

[1] We do not make the optimistic assumptions as in [16] and ignore all fixed call sites (defined in Section 2) in Tables 3 and 4 since they can all be resolved statically.

Benchmark	Classes	Methods	Fields				
			Total	Inaccessible		Final	
				Static	Instance	Static	Instance
compress	2059	21563	5245	730	1647	1865	491
jess	2201	22226	5314	732	1699	1862	488
db	2051	21563	5229	727	1634	1862	488
javac	2225	22764	5491	730	1799	1928	488
mpegaudio	2104	21896	5364	770	1724	1896	488
mtrt	2073	21699	5265	728	1669	1862	488
jack	2104	21844	5301	731	1666	1864	490
jbb	2158	22677	5736	913	1888	1989	491
jlex	652	6345	1571	325	662	492	142
jtar	2132	22092	5560	911	1746	2035	495
jtb	785	7926	2012	361	723	540	141
soot	2459	20062	3842	374	2290	644	491
total	23003	232657	55930	8032	19147	18839	5181

Table 2. Java Applications

Benchmark	Points-to sets				Objects			Non-fixed call sites			
	Total	Complete			Total	Completely detectable		Total	Complete		
		EA	CA0	CA		CA0	CA		EA	CA0	CA
compress	205	4	46	106	24	5	23	11	3	3	3
jess	4685	152	1176	1679	458	37	78	677	23	92	235
db	353	29	91	164	23	8	13	140	36	38	84
javac	9437	123	1604	2045	808	44	109	1932	40	143	288
mpegaudio	2876	4	1435	2404	1040	1	1013	37	0	0	3
mtrt	1330	75	284	516	128	22	35	868	118	156	195
jack	2848	49	938	1469	218	16	41	851	13	98	498
jbb	10222	522	2341	3389	577	166	266	2521	152	222	629
jlex	2532	1633	2038	2169	184	77	91	553	452	484	486
jtar	3123	273	1177	1770	272	45	73	483	66	197	303
jtb	11780	176	2888	3192	820	25	77	2676	253	609	772
soot	81762	2501	13011	21369	5631	484	1080	25987	1080	2313	6176
total	131153	5541	27029	40272	10183	930	2899	36736	2236	4355	9672

Table 3. Benefits from exploiting field access and modification modifiers.

Taking advantage of their existence has resulted in more accurate analysis. Compared to CA0, CA has found 49.0% more complete points-to sets, 122.1% more complete call sites and 211.7% more detectable objects overall.

In EA, all the field modifiers are ignored. CA0 performs better than EA mainly due to the fact that CA0 distinguishes completely from incompletely detectable objects while EA does not. As a result, CA0 has succeeded in classifying many non-extant references as complete references. The exploitation of the `final` modifier in Rule S4 contributes about 4.0% and 0.1% to the improved precisions of CA0 in determining complete points-to sets and call sites, respectively. The concept of object detectability has other applications. For example, it

Benchmark	EA	CA0	CA		
				Increase over EA(%)	Increase over CA0 (%)
compress	3	3	3	0.0	0.0
jess	23	92	235	921.7	155.4
db	36	38	83	130.6	118.4
javac	40	143	243	507.5	69.9
mpegaudio	0	0	3	*n/a*	*n/a*
mtrt	118	156	195	65.3	25.0
jack	10	95	489	4790.0	414.7
jbb	152	220	623	309.9	183.2
jlex	452	484	484	7.1	0.0
jtar	66	197	303	359.1	53.8
jtb	253	609	772	205.1	26.8
soot	1080	2295	5928	448.9	158.3
total	2233	4332	9361	319.2	116.1

Table 4. A comparison of analysis techniques in determining mono call sites.

has helped us in developing an interprocedural side-effect analysis for incomplete programs, which cannot be discussed here due to the space limitation.

The knowledge about whether a call site is complete or not can be exploited in a number of ways. A complete call site is a call site whose targets are guaranteed to be in the analysed program. Some compiler optimisations can be applied to complete call sites. A complete call site that has a unique target can be devirtualised or inlined without any test (Section 4.3). In addition, some complete call sites such as `invokeinterface` can be virtualised or annotated to eliminate unnecessary dynamic type checks associated with them.

4.3 Mono Call Sites

A call site is a *mono* call site if it has a unique target method. These call sites can be devirtualised or inlined safely without any runtime tests. Table 4 compares CA, CA0 and EA in detecting the mono call sites from among the set of the non-fixed call sites in the application part of each benchmark. CA has detected significantly more mono call sites than CA0 and EA. CA improves EA by 319.2% overall. By exploiting the field modifiers, CA performs as well as CA0 in `compress` and `jlex` and outperforms CA0 in all the remaining benchmarks, resulting in a total increase of 116.1% in analysis precision.

4.4 Analysis Costs

Our experiments are conducted on a 2.4GHz Intel Xeon PC with 2GB memory. Figure 2 gives both the time and memory overheads of the points-to analysis and completeness analysis combined relative to the points-to analysis alone. The analysis time for `jlex` has decreased slightly and that for `jtb` remains unchanged.

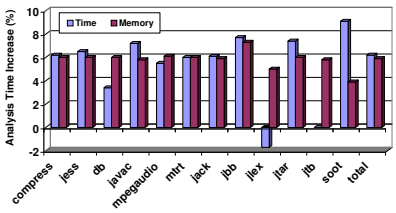

Fig. 2. Analysis costs of completeness analysis relative to the points-to analysis alone.

The analysis times for the remaining benchmarks range from 3.4% (for `db`) to 9.1% (for `soot`). The overall time increase for all the benchmarks is 6.2%. Due to Rule C6, the number of iterations required for constructing some point-set sets can be reduced. The memory overheads for all the benchmarks are small, ranging from 3.9% and 7.3%. The overall memory increase is only 5.9%.

5 Related work

Many points-to analysis techniques exist for object-oriented programs but there is little work done when these programs are incomplete. However, points-to analysis for incomplete programs in imperative languages has been studied [7, 15].

Rountev et. al. [14] study points-to analysis for incomplete Java programs in order to detect receiver types. Their approach works by creating *placeholders* that serve as representatives for unknown code. A limitation of this work is that dynamic class loading is not permitted. Chatterjee et al. [3] present a points-to analysis for library modules in order to find def-use relations. The analysis evaluates a parameterised points-to solution for each method and propagates conservative assumptions about the clients of the library in a top-down manner. A limitation of this approach is that it does not examine the effects of threads.

Extant analysis [16] is designed for the purposes of specialising Java programs in the presence of dynamic class loading. The technique partitions the references of a program into two categories: *(unconditionally) extant references* when they only point to the objects whose runtime types are in the analysed program and *conditionally extant (i.e., non-extant) references* otherwise. Our experimental results show that our completeness analysis yields more precise information about the completeness of points-to sets and call sites.

Escape analysis [2, 4, 20, 21] detects the objects that never escape out of a method or thread. An object escapes a method if its lifetime may exceed the lifetime of that method. An object that does not escape a method can be possibly allocated on the method's stack frame. If an object does not escape a thread, no other threads can access the object. The synchronisation operations associated with the object can be eliminated. For these reasons, if o is an escaped object, so will the objects pointed to by $o.f$. This facilitates the above two optimisations. However, in completeness analysis, if o is incompletely detectable, the objects pointed to by $o.f$ may be completely or incompletely detectable. Therefore, an incompletely detectable object is an escaped object but the converse is not true. So our object detectability analysis is different from escape analysis.

Some dynamic points-to analysis techniques for Java [8, 10, 12] restrict themselves only to the classes loaded during program execution. They do not determine the completeness of points-to sets. As a result, the analysis and optimisation techniques that make use of the points-to information may require runtime invalidation and recompilation mechanisms, which can hurt performance.

None of these above approaches exploit the access and modification properties of fields when computing points-to sets. These properties are, however, exploited in other kinds of analyses. Immutability analysis [11] is a technique for detecting mutability of fields and classes in a Java program. Field analysis [6] exploits the declared access restrictions placed on fields in order to determine such useful properties of these fields as `exact_type`, `nonnull`, `may_leak` and `only_init`.

6 Conclusion

In this paper, we describe a framework for points-to analysis and optimisation for incomplete object-oriented programs. As an analysed program is incomplete, some of its points-to sets and call sites may be incomplete. We present an completeness analysis technique combined with a whole-program points-to analysis to determine which points-to sets (call sites) may be incomplete in the sense that their pointed-to objects (target methods) are not statically resolvable. We introduce the notion of object detectability and show how such an information can be obtained as part of the completeness analysis. To the best of our knowledge, this is the first work that exploits the field access and modification properties in performing completeness analysis for incomplete object-oriented Java programs. We demonstrate by experiments that such an exploitation leads to better analysis precision. Our approach is compositional, which enables better points-to sets to be computed than those obtained when the points-to analysis is applied alone.

In this paper, completeness analysis is combined with a flow- and context-insensitive points-to analysis. One future work is to extend our approach to accommodate other kinds of points-to analyses. Another future work is to exploit type-based alias analysis to improve the precision of the results.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
2. Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 25–37, January 1998.
3. Ramkrishna Chatterjee and Barbara G.Ryder. Data-flow-based testing of object-oriented libraries. Technical Report 433, Rutgers University, 2001.
4. Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In *14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 1–19, November 1999.

5. Jeffrey Dean, David Grove, and Craig Chamber. Optimization of object-oriented programs using static class hierarchy analysis. In *5th European Conference on Object-Oriented Programming*, volume 952, pages 77–101. Springer, Aug. 1995.
6. Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.
7. Mary Jean Harrold and Gregg Rothermel. Separate computation of alias information for reuse. *IEEE Transaction on Software Engineering*, 22(7):442–460, 1996.
8. Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *18th European Conference on Object-Oriented Programming*, June 2004.
9. Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.
10. Igor Pechtchanski and Vivek Sarkar. Dynamic optimistic interprocedural analysis: a framework and an application. In *16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2001.
11. S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *Proceedings of CASCON 2000*, 2000.
12. Feng Qian and Laurie Hendren. Towards dynamic interprocedural analysis in JVMs. In *3rd ACM SIGPLAN Symposium on Virtual Machine Research and Technology*, May 2004.
13. Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java based on annotated constraints. Technical Report DCS-TR-424, Rutgers University, November 2000.
14. Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in java software. In *25th International Conference on Software Engineering*, May 2003.
15. Atanas Rountev and Barbare G. Ryder. Practical points-to analysis for programs built with libraries. Technical Report 410, Rutgers University, February 2000.
16. Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 196–207, June 2000.
17. M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, University Passau, November 2000.
18. Sun Microsystems. Java 2 software development kit version 1.2.2, July 1999.
19. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot: a java optimization framework. `http://www.sable.mcgill.ca/soot`, 1999.
20. Frédéric Vivien and Martin C. Rinard. Incrementalized pointer and escape analysis. In *ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 35–46, June 2001.
21. John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 187–206, November 1999.
22. Ayal Zaks, Vitaly Feldman, and Nava Aizikowitz. Sealed calls in Java packages. In *15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2000.