Enabling Static Analysis for Partial Java Programs

Barthélémy Dagenais Laurie Hendren

McGill University, Montréal, Québec, Canada

[bart,hendren]@cs.mcgill.ca

Abstract

Software engineering tools often deal with the source code of programs retrieved from the web or source code repositories. Typically, these tools only have access to a subset of a program's source code (one file or a subset of files) which makes it difficult to build a complete and typed intermediate representation (IR). Indeed, for incomplete object-oriented programs, it is not always possible to completely disambiguate the syntactic constructs and to recover the declared type of certain expressions because the declaration of many types and class members are not accessible.

We present a framework that performs partial type inference and uses heuristics to recover the declared type of expressions and resolve ambiguities in partial Java programs. Our framework produces a complete and typed IR suitable for further static analysis. We have implemented this framework and used it in an empirical study on four large open source systems which shows that our system recovers most declared types with a low error rate, even when only one class is accessible.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Experimentation

Keywords Partial Programs, Type Inference, Java

1. Introduction and Motivation

Static program analysis is an important tool for software engineering research: techniques such as bug detection [5] and feature location [21] heavily depend on static analyses to model a program's behavior and structure.

Compiler frameworks, with which many static analyses were developed, usually assume that the complete program is available (either as source code or as a high-level binaries such as Java .class files), even if only part of that program is to be analyzed. When the complete program is available, it is straightforward for the compiler to build a correct, typed, and complete intermediate representation (IR) for the part of the program to be analyzed.

However, some methodologies used by software engineering techniques preclude the access to complete programs. Indeed, source code retrieved from software versioning systems [8], web repositories [19], or bug reports [7], is typically difficult to compile: source folders and libraries required to build a snippet of code may not be known or accessible, and the correct versions of those code artifacts may be impossible to automatically determine (e.g., which version of Log4J is needed to compile Foo.java 1.4?).

The goal of this paper is to provide techniques that produce complete and typed intermediate representations for the source code of partial Java programs, even when only part of the program is accessible. The unavailability of many class declarations leads to two main challenges: (1) dealing with syntactic ambiguities and (2) determining the correct types for expressions such as field accesses and method calls.

Syntactic ambiguities arise when classes and members for other parts of the program are not available. For example, consider the statement `E.dothat()`. Without the declaration for `E`, it is not possible to determine if `E` is a class or a field. If `E` is a missing class, it could be a call to a static method `dothat()` which is a member of the missing class `E`. Otherwise, if `E` is a missing field, then this is a virtual method call, with receiver `E`.

Typing problems arise when a compiler or tool does not have access to the complete type hierarchy and the signatures of fields or methods in classes that are directly or indirectly referenced by the program under analysis. A Java compiler usually creates an intermediate representation (IR) such as an abstract syntax tree, annotating the IR with the appropriate types, based on type declarations. For example, given the following complete Java code snippet, the compiler would use the declaration of class `A` at line 1 to find that the declared type of the expression `a.p1` at line 9 is `String`. The compiler would also use the declaration of class `A` to find that the method called at line 10 is the method `A.add(Object)` declared at line 3.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

```
1  class A {
2    String p1;
3    void add(Object o) {}
4  }
5
6  class B {
7    void main() {
8      A a = new A();
9      a.p1 = "hello";
10     a.add(a.p1);
11   }
12 }
```

Software engineering tools that operate on partial programs may not have access to the complete program and all of its declared types. For example, assume that a tool had access only to the source code for class `B` (and not `A`). In this case, the tool cannot find the declared type of some expressions in the incomplete program (e.g., what is the declared type of `a.p1`?). To deal with this problem, these tools often fall back to syntactic analysis, which provides limited information. For example, if only class `B` is available, a syntactic analysis will conclude that a method named *add* with *one* parameter is called at line 10. This lack of precise type information is a problem for tools, like PARSEWeb [19] and SemDiff [8], that analyze partial programs to recommend to programmers method calls from arbitrary frameworks. These tools require the types of the receiver and the formal parameters in order for their recommendations to be useful (e.g., telling the user to call a method named *add* is not helpful if many classes declare such method). PARSEWeb and SemDiff have thus begun to perform partial type inference to get more information from the source code of incomplete programs. For example, if only the declaration of class `B` was accessible, they would conclude that at line 10, the method `A.add(String)` is called by looking at the assignment of field `p1`. This information, although not strictly correct (there is no method `A.add(String)`, but there is a method `A.add(Object)`), is more precise than the one provided by syntactic analysis.

Software engineering techniques usually tolerate a certain level of imprecision and errors, as measured by precision and recall, so they can benefit from information that is often more precise, but potentially incorrect. Thus, in designing our approach we traded some guarantees on correctness for increased precision. This also implies that our approach is not suitable for situations where a sound analysis is required, such as in program optimization.

In this paper, we propose a technique, *Partial Program Analysis* (PPA), which builds a typed IR of incomplete Java programs' source code. Our technique recovers the declared types by performing partial type inference and resolves syntactic ambiguities inherent to incomplete programs using heuristics. Although it is impossible to guarantee that the generated IR is correct with respect to the result one would get given the complete program, we aim to generate an IR which: (1) obeys the type constraints available in the partial program under analysis, (2) does not introduce unknown program constructs, and (3) is suitable for use with other static analysis tools.

We implemented this approach in a prototype using Soot, a static analysis framework [20], and Polyglot [15], an extensible compiler framework. Currently, our prototype transforms Java source code from an incomplete program into a typed abstract syntax tree (AST) and into Jimple, a typed three-address intermediate representation, because those representations are suitable for most static analyses. We used an early version of the PPA prototype in SemDiff to analyze the evolution of method calls in code retrieved from software repositories and to recommend precise method invocation to adapt client programs that broke during framework evolution [8]. Our positive experiences with that project gave us some confidence that PPA will be useful for developing other software engineering analyses and tools.[1]

To validate to what extent our proposed PPA technique produces useful results, we performed a quantitative study on four open source programs, three of them from the DaCapo benchmark suite [18], for three common scenarios. We found that even for the hardest scenario, when the source code of only one class is available, partial program analysis could generate an intermediate representation that was on average 91% identical to the intermediate representation of the same class analyzed with the whole program.

The contributions of this paper include: (1) the PPA techniques that allow us to analyze incomplete Java programs, and that deal with both syntactic ambiguities and typing problems, (2) the implementation of a tool based on PPA that produces an IR and AST representation of an incomplete program, and (3) an empirical evaluation of our approach.

In the rest of this paper, we first introduce our problem and terminology in more detail in Section 2. We then describe our approach to solving the typing problems in Section 3 and the ambiguous syntax problems in Section 4. We put it all together in Section 5, where we describe our overall algorithm. We then report on the results of the empirical evaluation of our technique (Section 6). Finally, we discuss the related work in Section 7 and conclude in Section 8.

2. Partial Object-Oriented Programs

We consider a partial program to be *a subset of a program's source files*. This definition is suitable for current software engineering tools that can get as input complete source files and that require more precise information than what syntactic analysis can provide.[2] In a source file, we associate a *type fact* with all references to *declared types*. For example, there are six type facts associated with line 6 in Fig-

[1] The PPA implementation is available at http://www.sable.mcgill.ca/ppa.

[2] This definition of partial programs could even be relaxed to include any snippet of well-formed code. Although our approach does not rely on a complete source file, the parser implementation that we currently use does.

```
1  class D {
2    int field1;
3    void main() {
4      A varA = new A();
5      String s = "hello";
6      field1 = m1(s);
7      varA.field3 = s;
8      varA.field3 = B.field3;
9      Object o = new String("hello");
10   }
11   short m1(Object o) {return 0;}
12 }
```

Figure 1. A partial program

ure 1: the declared type of field1's container (`D`), the declared type of field1 (`int`), the declared return type of method m1 (`short`), the declared type of method m1's target (`D`), the declared type of m1's formal parameter (`Object`) and the declared type of m1's actual parameter (`String`). To simplify our presentation, we will use the term *type* to refer to a declared type (as opposed to a runtime type) in the remainder of this paper. For example, at line 9, the declared type of the variable `o` is `Object`, but the type of this variable during runtime is `String`.

Given a partial program, the challenge is to recover as many correct type facts as possible without having access to the rest of the program, i.e., referenced source files, binaries, or dependencies such as libraries. For example, by analyzing only class `D`, we can infer two type facts at line 7: (1) the type of the container holding `field3` is `A` or one of its ancestors and (2) the type of `field3` is a `String` or one of its ancestors. Furthermore, we know at line 7 that the type of `varA` is `A` because we have access to its declaration in method `main`. In the remainder of this paper, we will use static fields in our examples (e.g., `B.field3` at line 8) instead of instance fields to reduce the size of the examples: our analysis considers each syntactically different field access as a distinct field, even if they share the same name. For example, at line 8, PPA considers that there are two distinct fields `field3`: the first is attached to the local variable `varA` and the second is attached to the type `B`. In the text, we will also always refer to the field name without the qualifier when it is unambiguous.

Having access to only a subset of the source files forces us to make an important assumption when performing partial program analysis:

Compilable Program Assumption: The source files of a partial program compile without any error given the required dependencies.

This is a reasonable assumption for code extracted from software versioning systems or web repositories because a popular convention is to only commit source files if they compile. The absence of class declarations makes it impossible to detect type-related errors such as calling a non-existing method so we cannot assess whether the code is compilable or not. The compilable program assumption is thus necessary to infer type facts: in potentially uncompilable source code, every inference made on type usage could be wrong.

```
1  class E {
2    void main() {
3      A.field1 = new B("hello");
4      Collection coll = A.field2;
5      A.field1.m2().m3();
6    }
7  }
```

Figure 2. Inferring type facts

In this paper, we will focus on Java 1.4 which does not include features such as generics and autoboxing, and we will also assume that the user has access to standard Java types, e.g., `java.lang.Object`, either in a binary or source format. Again, this is a reasonable assumption because those classes are required to execute any Java program.

3. Recovering Types in Partial Programs

When analyzing a partial Java program, it is possible to infer type facts by looking at how a type is used in the program. For example, in Figure 2, we see that the program assigns an instance of class `B` to `field1` at line 3. Because of the Java type system, we know that `field1` must be `B` or one of its ancestors.

We define the operator $dt(x)$ which returns the declared type t of the Java expression x. For example, at line 4, $dt(\texttt{coll}) = \texttt{Collection}$. We also define the *subtype* operators, $t1 <: t2$, which means that $t1$ is a subtype of $t2$, i.e., it is either $t2$ or a descendant of $t2$. The *supertype* operator, $t1 :> t2$, means that $t1$ is either $t2$ or an ancestor of $t2$. For the last two operators, we consider class extension and interface implementation and extension (through the `extends` and `implements` keywords) to be the only generators of subtypes and supertypes. We use the *related type* operator, $t1 \sim t2$ when we know that two types are related, i.e, one is a subtype of the other or they share a common ancestor or descendant.[3] Finally, we define the operator $target(x)$ which returns the target's type t of a method x or the container's type of a field x. For example, $target(\texttt{field1}) :> \texttt{A}$. Because partial programs are imprecise, a type t can be either precise (e.g., $dt(\texttt{coll}) = \texttt{Collection}$) or imprecise (e.g., $dt(\texttt{field1}) :> \texttt{B}$).

When we try to infer a type fact, it sometimes happens that we cannot recover any information at all. We define the following data structures to handle these cases:

[3] In Java, because all reference types are a subtype of `java.lang.Object`, all reference types are related to each other. The related type operator can still be used to distinguish reference types from primitives as they are not related.

The `unknown` type is used as a placeholder for any type that is referenced, but not explicitly named. For example, in the following call chain, `m2().m3()`, the return type of the method `m3` is `unknown`, which denotes either a Java primitive, a Java class or `void`.

To fully qualify any type which has an ambiguous Fully Qualified Name (FQN), we define the package `p-unknown`. This package is necessary to distinguish a type located in the default (empty) package from a type that is located in an unknown package. The fully qualified name of the `unknown` type is thus `p-unknown.unknown` even if for the sake of brevity, we will always use the short name `unknown`.

Finally, we define a *type fact* as being a record containing the following attributes: (1) a Java expression x (e.g., a reference to a field), (2) the type t of the expression before the inference, and (3) the type t of the expression after the inference. For example, if we just found that the unknown field `field1` was a supertype of class `B`, we would have inferred the following type fact: {`field1`, `unknown`, :> `B`}.

3.1 Type Inference Strategies

To infer type facts, we rely on several strategies based on the Java programming language type system. These inference strategies are sound in the sense that the real declared types will always respect the constraints of the inferred type facts. For example, if a strategy infers that the type of an expression is a subtype of `java.util.List`, the real type is guaranteed to have this property. Although the inference strategies can generate imprecise type facts such as {`field1`, `unknown`, <: `java.lang.Object`}, we found during our evaluation that they generally recover the real type.

Table 1 shows the intuition behind the inference strategies that we devised. The formal inference rules for each of those strategies are presented in Appendix A.

3.2 Inferring Method Bindings

When an expression's type has been inferred (e.g., using one of our inference strategies), it is sometimes possible to determine a method binding that is ambiguous from a purely syntactic point of view. Consider the call to method `m1` at line 4 in Figure 3. Because there are two declarations of a method `m1` with one parameter, it is not possible to decide which method is called when looking only at the syntax of the program. On the other hand, if we perform some type inference, we know that *dt*(`field1`) :> `B` at line 3, and thus, we are certain that we call the method `m1(B)` declared at line 10. Once we know the exact method binding, we can then use our *method binding* inference strategy. Unfortunately, there are cases like line 5 where we cannot identify the correct method binding because the method is overloaded and the declared type of the parameter is unknown.

A class can also potentially overload a method declared in a supertype. For example, in class `H`, we cannot soundly infer that the method called at line 17 is the one declared at line 20. Indeed, we know from line 16 that *dt*(`field3`) :> `C`. Suppose that *dt*(`field3`) = `Object` (which respects the type fact we inferred) and that the supertype of `I` defines a method `m1(Object)`. It follows that the method called at line 17, is `I.m1(Object)` and not `H.m1(C)`. Determining a method binding is thus an undecidable problem because of overloaded methods.

```
1  class G {
2    void main() {
3      A.field1 = new B();
4      A.field4 = m1(A.field1);
5      A.field5 = m1(A.field2);
6      B.m2(2);
7      B.m2(A.field2);
8    }
9
10   D m1(B b) { ... }
11   E m1(int i) { ... }
12 }
13
14 class H extends I {
15   void main() {
16     A.field3 = new C();
17     m1(A.field3);
18   }
19
20   void m1(C c) { ... }
21 }
```

Figure 3. Method binding

3.3 Inferring Type Members

Until now, we focused on inferring the type of an expression, but, at the same time, we need to infer the existence and types of members (i.e., fields and methods). For example, in Figure 3, we inferred at line 3 that there is a field named `field1` that is declared in the type `A` or one of its supertypes. When analyzing a complete program, we could check whether these members exist and are accessible. Because we only have access to class `G`, we must rely on our assumption that the underlying code compiles and that both members are accessible from the context of the calling method `G.main`.

More specifically, when PPA encounters a reference to a type whose declaration is not available, it creates an internal representation of the type. If a member is accessed from this type, we add the member to the generated type declaration. For example, at line 3 in Figure 3, PPA would generate the fact that class `A` has a field called `field1` whose type is :> `B`.

In adding missing members, fields and methods are treated differently. Since fields cannot be overloaded, we only generate a missing field once and reuse this one if another occurrence of the same field occurs. However, since methods can be overloaded, generated methods cannot be reused like generated fields. For example, at line 6, we can infer that there is a method called `m2` that is in one of the

Inference Strategy	Example	Explanation
Assignment	B. field1 = "Hello_World"; C c = B. field2 ;	The type of an unknown expression on the right-hand side is the subtype of a known left-hand side expression's type and vice-versa, e.g., {field1, unknown, :> java.lang.String} and {field2, unknown, <: C}.

Return	**int** m1() {
	return B.method2();
	}
		The type of an unknown return expression is the subtype of the method's declared return type, e.g., {method2,unknown, <: int}.

Method binding	**void** main() {
	B. field3 = me(B. field4);
	}
	
	D m3(E p1) {...}
		If we know the exact method binding, the types of the actual parameters are subtypes of the formal parameters' types, and the expression to which the method is assigned to is a supertype of the method's return type, e.g., {field3, unknown, :> D} and {field4, unknown, <: E}.

Condition	**if** (B.method4()) {
	...
	}
		An expression used as a condition must resolve to a boolean, e.g., {method4, unknown, boolean}.

Binary and unary operators	**int** i = B. field7 − 10;	Depending on the operands' types and the expected return type of a binary or unary expression, it might be possible to infer the primitive type of an expression by taking into account implicit type promotion, e.g., {field7, unknown, <: int}.
Array	B. field10 = B. field8 [B. field9];	When an unknown expression is used as an array index, we can infer that it is a subtype of int. For example, at line 14, we can infer that {field9, unknown, <: int}. When an unknown expression is accessed using an array index, we can infer that the type of the expression is an array, e.g., {field8, unknown, unknown[]}.

Switch	**switch**(B. field11) {
	...
	}
		Switch expressions enable us to infer that the operand is a subtype of the int primitive, e.g., {field11, unknown, <: int}.

Conditional	B. field14 = B. field12 ? 'c' : B. field13 ; **int** i = B. field12 ? 'c' : B. field16 ;	Conditional expressions, represented by the ternary operator ? can be used in two fashions. First, if their return type is not know, we can at least infer that the two last operands' type must be related, e.g., {field13, unknown, ∼ char}. Indeed, if dt(field14) = long, field13's type can be either a subtype or a supertype of char: we can thus only say that the two types are related. When the return type is known, we know that the last two operands must be a subtype of the return type, e.g., {field16, unknown, <: int}.

Table 1. Type Inference Strategies

supertypes of B and that takes as a parameter a supertype of int. At line 7, we cannot safely reuse this fact to infer the type of the actual parameter field2 because there might be another method called m2 with a different parameter type. We thus infer that there is a method called m2 that takes a parameter of type unknown.

Finally, inferred type members can be refined by type inference. For example, if we later find that dt(field2) <: C, we will add a method m2(C) to B.

3.4 Combining Type Inference

Sometimes, we can infer two type facts related to the same expression. For example, in Figure 4, at lines 3 and 4, we infer that dt(field3) <: Object and dt(field3) <: String. By definition of a subtype, it is clear that dt(field3) <: String because String <: Object. We say that the two inferred type facts are *converging* and we only keep the most precise type fact (<: String). On the other hand, the type facts that we infer at lines 5 and 6 are *erroneous*: dt(field4) :> Object and dt(field4) <: String cannot be true at the same time. Erroneous type facts contradict our compilable program assumption, but this is one of the few cases where we can detect and report a compilation error. Finally, the two last type facts at lines 7 and 8, dt(field5) <: B and dt(field5) <: C, are *conflicting*: it is not possible to decide which of the two type facts is the most precise because three type hierarchies can explain the code of lines 7 and 8[4]:

1. B <: C, so dt(field5) <: B
2. C <: B, so dt(field5) <: C
3. There exists a type P which is a common descendant of B and C (either B or C must be an interface). In that case, neither type fact is more precise.

In the case of the third possibility, even if we knew the whole type hierarchy of B and C, it would still be impossible to determine the type of field5 because there might be more than one common descendant P.

[4] The converse is also true if we have the two following type facts: dt(field5) :> B and dt(field5) :> C.

```
1  class F {
2    void main() {
3      Object o1 = A.field3;
4      String s1 = A.field3;
5      A.field4 = new Object();
6      String s2 = A.field4;
7      B b = A.field5;
8      C c = A.field5;
9    }
10 }
```

Figure 4. Combining type facts

```
1  class Y {
2    int m1() {
3      A a1 = Z.field1;
4      Z.field1 = Z.field2;
5      A a2 = Z.field3;
6      Z.field4 = Z.field3;
7    }
8  }
```

Figure 5. Conflicting type direction

When we encounter two conflicting type facts, we first try to select the safest one, where we determine that a type fact is safer than another using the total ordering: `unknown` $<$ `missing` $<$ `super missing` $<$ `full`. Each member of this ordering is defined as follows:

If the type of a fact is *unknown*, it is less safe than a fact whose type is known but whose declaration is *missing* (e.g., we know that $dt(x) =$ `B`, but we do not have access to the declaration of `B`). A known type with a missing declaration is less safe than a known type whose declaration is accessible but not all of its *supertypes* (e.g., we have access to the declaration of `B`, but one of its supertype's declaration is missing). Finally, the safest type, *full*, means that we have access to its declaration and the declaration of all of its supertypes. If the two type facts are equally safe, we keep the first type fact that we inferred. The rationale behind this scheme is that we only keep types that allow us to work with safer (i.e., known) types, which is generally more precise than just keeping the first inferred type fact and ignoring further type facts.

Another alternative strategy would be to treat all related type facts as constraints and try to solve the constraints to obtain the most precise type fact. However, during our early experimentation with PPA, we observed that we generally produced either one type fact or multiple conflicting type facts (e.g., `field5` in Figure 4), which forced us to make an arbitrary choice. Therefore, we had no reason to think that a more complex type combination scheme involving constraint solving would produce more accurate results.

Finally, when combining type facts, the direction of the types, whether they are subtypes or supertypes, might conflict. In Figure 5, we can produce this *inference chain* at lines 3 and 4: $dt($`field2`$) <:$ $dt($`field1`$) <:$ `A`. It is thus clear that $dt($`field2`$) <:$ `A` by transitivity. On the other hand, the directions of the types at lines 5 and 6 conflict: $dt($`field4`$) :>$ $dt($`field3`$) <:$ `A`. We can thus only say that $dt($`field4`$) \sim$ `A` or in other words, that there is a path in the type hierarchy that links `field4` with `A`.[5]

```
1  package ca.mcgill;
2
3  import ppa.*;
4  import java.util.*;
5  import soot.Unit;
6
7  class A {
8    void main() {
9      C c = B.getC();
10     Collection coll = B.getCollection();
11     coll.doThis();
12     Unit u = B.getUnit();
13     ppa.internal.D d = B.getD();
14   }
15 }
```

Figure 6. Ambiguous fully qualified name

4. Ambiguous Syntax in Partial Programs

The programming language syntax is a source of imprecision: Table 2 shows the main constructs that are ambiguous in partial Java programs.

When we encounter such ambiguous syntax constructs, we can either (1) create an unknown node in the AST representation, or (2) use an heuristic that guesses the real construct. The first strategy is sound, in the sense that it doesn't introduce a potentially wrong construct. However, it potentially introduces many unknown parts of the code, losing useful parts of the program. Furthermore, it breaks the compatibility with client tools, which assume only valid Java constructs. We thus relied on the use of heuristics that can produce wrong, but potentially more precise results.

Fully Qualified Name. When we encounter a reference to a simple type name (e.g., `String`), we use the following heuristic to find the FQN of the ambiguous type:

1. If the ambiguous type is fully qualified, we use that FQN (e.g., `ppa.internal.D` in Figure 6).
2. If there is an explicit import statement that ends with the ambiguous type name, we use the FQN specified in the import statement (e.g., `soot.Unit` in Figure 6).

[5] Unfortunately, in Java, this is true for any two given reference types because every type is a subtype of `Object` so there is always a path from one type to another type that passes by `Object`. In a language like C++ which does not have this concept of a universal supertype, related types would have a more precise meaning.

Syntax ambiguity	Example	Explanation
Fully Qualified Name (FQN)	Figure 6	The FQN of a type cannot always be soundly inferred in a partial program because a programmer can use the `import *` construct. For example, at line 9 in Figure 6, the FQN of `C` can either be: `C` (in the default package), `ca.mcgill.C` (in the package of `A`), or `ppa.C` (because of `import ppa.*`). Additionally, we cannot soundly infer the FQN of a type contained in a known package (e.g., `java.util`). For example, at line 10, the `Collection` type might be contained either in `java.util` or in `ca.mcgill`. Line 11 gives a hint that the latter FQN is the good one since the `java.util.Collection` type does not declare the method `doThis`.
Package or Class?	Line 13 in Figure 6	It is not always possible to discriminate the part in the FQN that relates to a type from the part that relates to the package. For example, at line 13, variable `d` might be of type `D` or of type `internal.D` (an internal class).

Field or Class?	**class** B **extends** C {
	void main() {
	E.doThat();
	E = **new** F();
	}
	}
		An expression such as `E` in the first line of the `main` method can be either a field or a class. In the former case, we infer that $target$(`doThat`) = `unknown` and in the latter case, we infer that $target$(`doThat`) :> `E`.
		It is sometimes possible to resolve this ambiguity by looking at other lines of code (such as the assignment) that provide hint that the expression is a field.

This or Container?	**class** G **extends** H {
	public void main() {
	I i = **new** I() {
	public void m1() {
	f1 = 2;
	...}
		It is not always possible to soundly infer the container of a particular member in an internal class because a reference to `this` or to the `container` of an internal class is implicit in Java. For example, it is not clear whether $target$(`f1`) :> `I` or if $target$(`f1`) :> `H`.

Overloaded operators	**class** J **extends** K {
	void main() {
	int i = 2 − f2;
	String s = "Hello"+(f3+2)+"World";
	}
	}
		Some operators are overloaded by the Java language. For example, it is not clear whether the + operator is the addition operator or the String concatenation operator in the `main` method. In the latter case, because a String can be concatenated with an arbitrary type instance or a primitive, it is still not possible to soundly infer the type of the field `f3`.

Table 2. Ambiguous syntax constructs in Java

3. If we have access to the packages imported using a wildcard import statement (e.g., `import java.util.*`) and we find a type whose name is the same as the ambiguous type name, we use that FQN (e.g., `java.util.Collection` in Figure 6). If later on we realize that this type is *not adequate* (e.g., we are calling a method that is not declared in this type), we rely on the last two heuristics to determine its FQN.
4. If there is no wildcard import statement, we append the name of the ambiguous type to the package of the analyzed type (e.g., `ca.mgill.C`).
5. If there is at least one wildcard import statement, we append the name of the ambiguous type to the unknown package (e.g., `p-unknown.C` in Figure 6).

Rules #3 and #4 can lead to wrong fully qualified names because the type might be declared in the default package. We expect most programs to avoid defining types in the default package because this practice is discouraged and often impractical.

Package or Class? We always consider the last part of a fully qualified name (after the last dot) to be the simple name of the type and the rest of the FQN to be the package. This can be a false assumption if the FQN refers to an internal type. A false assumption has no impact on the FQN (it is the same no matter if the type is internal) but it changes the type of node in the AST representation. Thus, if at a later point we find that the initialization can only refer to an internal type, we modify its AST representation.

Another strategy would be to use the Java naming convention (a type name and a package name should respectively start with an uppercase and lowercase character) to determine which part of the FQN is the package and which part is the type. We would still need to tune this heuristic on a per project basis.

Field or Class? We consider any ambiguous reference (e.g., `E.doThat()`) to be a static method call from a class. If we find a hint contradicting that assumption (such as the instantiation of the ambiguous reference), we change the AST node accordingly. Like the previous heuristic, we could also use the Java naming convention.

This or Container? Most of the time, an ambiguous reference to `this` or to the container of internal types is im-

possible to resolve. We thus chose to always replace such ambiguous references by a reference to `this`.

Overloaded operators. When we encounter an overloaded operator such as `+` or `&`, we always consider that the type of the operands is `unknown`. We use the binary operator inference strategy to decide the type of the operands when it is possible.

5. PPA Algorithm

Figure 7 shows an overview of the algorithm which consists of three passes. Although the general techniques introduced in this paper could be used in other systems, we implemented our approach using Polyglot [15] and Soot [20]. Polyglot is an extensible compiler that creates an AST representation of a source file by applying various passes such as disambiguation, type checking and exception checking. Soot uses Polyglot as a frontend to parse Java source files and then transforms the AST into a three-address intermediate representation called Jimple that can be used to perform data flow analysis. Our algorithm mainly extends Polyglot and works at the AST level. We first review the three main passes of the algorithm and then discuss the different modes in which the algorithm can be executed, its termination property, and its time complexity.

Seed pass. The seed pass is performed while Polyglot builds the AST of a source file. First, Polyglot tries to disambiguate each AST node (e.g., it determines if the expression is a field reference, a method call, a local variable reference, etc.). We modified Polyglot so it infers the missing type members as described in Section 3.3 and uses the heuristics described in Section 4 to resolve ambiguous syntax constructs.

Once the AST nodes are disambiguated, PPA visits each node and applies the inference strategy (presented in Appendix A) that corresponds to the type of the visited AST node. For example, at line 4 in Figure 8, PPA can use the assignment inference strategy to infer the following type fact: $\{$`field1`, `unknown`, $<:$ `A`$\}$.

We visit each node of the AST in postfix order because the type of the parent node is often determined by the children's types. During the visit, we generate type facts and append them to a worklist. When two type facts refer to the same expression (e.g., we can infer two type facts at line 4 and 6 that are related to the field `Z.field1`), we merge them according to the combination strategy we presented in Section 3.4.

During the seed pass, we only *infer* type facts and the `unknown` type is assigned to all unknown expressions. If a complete and correct program was available, there would be no unknown types at this point. However, partial programs often have some unknown types after the seed pass. For example, in Figure 8, $dt($`field1`$) =$ `unknown`.

Type inference pass. Once the AST is built, we can use the type facts that we inferred to modify the nodes of the AST (called *Make node safer* in Figure 7). When modifying the type of a node, we keep the complete type fact in memory, but we can only assign a simple type to a node to simplify the usage of the AST. For example, if we have the following type fact, $\{$`field1`, `unknown`, $<:$ `A`$\}$, we modify the declared type of the field nodes at lines 4, 5, 6, and 8 to be equal to `A`.

Finally, when we modify a node, it is possible that we can infer a new type fact. For example, at line 5, when we modify the assignment node, we can infer that $\{$`field2`, `unknown`, $<:$ `A`$\}$ using the assignment inference strategy. The inferred type facts are appended and merged into the worklist.

Method binding pass. When we build the AST and infer type facts, we can encounter ambiguous method calls. For example, at line 9, we do not know which `println` method is called: it is an overloaded method. During the seed pass and type inference pass, we only select a method binding if there is no ambiguity to make sure that we do not introduce potentially erroneous or conflicting type facts.

The method binding pass basically *forces* the compiler to select the first possible declaration of method calls that remain ambiguous. Once the declaration is selected, this enables the inference of new type facts that are appended and merged into the worklist. The worklist is then processed like in the type inference pass. For example, if we executed this pass on the program listed in Figure 8, we would find that two method calls remain ambiguous: the call to `println` at line 9 and the call to `m2` at line 10 (because `Y` extends `X`, the call might refer to a method declared in `X`). By forcing the selection of a method declaration, we would conclude that the method `println(boolean)` is called at line 9 and `m2(C)` at line 10, which would lead to the inference of the two following type facts: $\{$`field4`, `unknown`, `boolean`$\}$ and $\{$`field5`, `unknown`, $<:$ `C`$\}$.

Results. Once the three passes are completed, PPA produces a typed abstract syntax tree. Each node in the AST contains three kinds of information: (1) a type, (2) whether the declaration of the type is available (i.e., the source is accessible) or was generated by PPA, and (3) whether the type is unsound. A type can be unsound if it was inferred using one of our syntax heuristics or in the case of Java, if it was inferred through the related type operator ($\sim$). As seen in Section 3.4, types can be also be obtained from an inference chain: if one of the type in the chain is unsound, the rest of the chain is also considered to be unsound. Finally, the Soot framework translates the abstract syntax tree into a typed three-address intermediate representation.

Modes of execution. There are three main parameters that can be adjusted when using PPA.

The first parameter concerns the input of the analysis. Depending on the availability of source files, PPA can be performed on one Java source file at a time or on a set of source files. In the latter case, the worklist containing the inferred type facts is *shared* among all source files and the type infer-

ence and the method binding passes are only executed once the seed pass has been performed on each source file. This enables the sharing of inferred type facts which can lead to more precise inference, but it can also propagate imprecise type facts (see Appendix B for an example).

The second parameter allows the user to disable type inference, effectively preventing the execution of the type inference and the method binding pass. When type inference is disabled, all unknown expressions are assigned to the `unknown` type. PPA still performs type member inference and uses our heuristics to resolve ambiguous syntax constructs because those are needed to build the AST.

Finally, the third parameter enables the user to disable the method binding pass, preventing the selection of arbitrary method bindings.

In Section 6 we use these parameters to examine the effectiveness of our approach in different scenarios and to measure the added benefits of enabling the type inference and method binding passes.

```
// Seed pass
for each node in AST do
  Disambiguate node
  Infer type facts
  Put and merge type facts into worklist
end for

// Type inference pass
while worklist is not empty do
  for each node impacted by type fact do
    Make node safer
    Infer type facts
    Put and merge type facts into worklist
  end for
end while

// Method binding pass
for each ambiguous method call do
    Select the first possible call binding
    Infer type facts
    Put and merge type facts into worklist
end for
while worklist is not empty do
  for each node impacted by type fact do
    Make node safer
    Infer type facts
    Put and merge type facts into worklist
  end for
end while
```

Figure 7. Partial program analysis algorithm

```
1  class Y extends X {
2    int m1() {
3      System.out. println (Z. field1 );
4      A a1 = Z. field1 ;
5      Z. field1 = Z. field2 ;
6      B b1 = Z. field1 ;
7      Z. field2 = Z. field3 ;
8      A.m1(Z.field1 );
9      System.out. println (Z. field4 );
10     m2(Z.field5 );
11   }
12
13   void m2(C param1) {...}
14 }
```

Figure 8. Sample program

Termination. Our algorithm is ensured to always terminate. First, the number of AST nodes and type facts in a given program is finite, so the first pass is always sure to complete. The second pass also always completes because the number of times a type fact related to a particular expression can be inferred is finite. As explained in Section 3.4, we only infer a new type fact related to an expression if (1) it converges or (2) it conflicts with a previous type fact and is safer than a previous type fact. The number of converging type facts that we can infer on an expression is bounded by the depth of the type hierarchy and the number of conflicting type facts that we can infer on a particular expression is bounded by 4 (from unknown to full). Finally, in the third pass, we select the binding of ambiguous method calls, which has a finite number.

Complexity. To analyze the time complexity of our algorithm, we consider each pass individually. The complexity of the algorithm is bounded by n, the number of AST nodes in all source files, f_a, the number of type facts in all source files, r, the maximum number of nodes referring to a type fact in all source files, f_n, the maximum number of type facts that can be inferred on a node (typically the maximum number of parameters in a method call), m, the maximum number of ambiguous method calls, k, the constant time required to perform operations on a node such as disambiguation, modification or method call binding selection, and h, the maximum depth of the program's type hierarchy. We consider that the selection of a method binding takes a constant time because we always select the first binding. We can express the complexity of each pass with the following formulas:

$$\textbf{Seed pass} = nf_nk = O(nf_n)$$

$$\textbf{Type inference pass} = h \times f_a r f_n k = O(h f_a r f_n)$$

$$\textbf{M. binding pass} = mf_nk + h \times f_a r f_n k = O(mf_n + hf_a r f_n)$$

Although the cost to process the worklist is potentially high ($O(hf_af_nr)$), several factors reduce the time complexity in practice. We found during our evaluation of partial program analysis that $h < 4$ because most type facts related to the same expression were conflicting, and when they were converging, they always converged fast. The number of nodes impacted by a type fact, r, was also small: it was

on average equal to 1.57 and always below 213. Finally, the number of inferred type facts per node was low: $f_n < 4$.

6. Evaluation

To validate the performance of partial program analysis under various circumstances, we performed an empirical study on four open-source systems. We were mostly interested in evaluating the following criteria:

1. The quality of the results obtained by PPA as measured by the number of correct and erroneous type facts.
2. The impact of the input (i.e., size of the partial program) on PPA precision.
3. The contribution of the various inference strategies in producing more precise results.

6.1 Experimental Design

We performed partial program analysis on every single class, including anonymous and internal classes, of four open-source systems. Table 3 shows the target systems along with their version, the number of classes and the number of source lines of code (SLOC) they have. We selected these systems because they are relatively complex, their version history was available, they could be compiled with Java 1.4, and because the programming language and software engineering communities frequently analyze those programs. Indeed, the first three systems, Lucene [3], JFreeChart [1], and Jython [2] are part of the DaCapo benchmark suite [18]. Because the three first systems are self-contained, i.e., they do not require any other library outside the Java standard library, we selected a fourth system, Spring [4], which depends on 90 external jar files to compile. This was to increase the external validity of our evaluation by analyzing various kinds of Java programs.

In general, we wanted to assess the quality of the results obtained by partial program analysis against the results obtained when the complete program is available. To perform this comparison, we executed PPA on each class separately, without any other classes in the target system, and obtained an intermediate representation of each class in the form of a Jimple file. We also transformed every class of the complete target system into Jimple. We thus obtained two Jimple representations for each class, one from PPA and the other from the complete system, that we could compare. The following example shows two Jimple statements, the first one from the partial program, the second one from the complete program.

```
1: i = virtualinvoke
    $r1.<p-unknown.unknown: int length()>();
2: i = virtualinvoke
    $r1.<java.lang.String: int length()>();
```

When comparing the type facts referenced by two statements, there are four possible outcomes:

correct The two types are the same. For example, the return type of the method `length` is `int` in both statements.

Target	Version	# Classes	SLOC
Lucene	2.2.0	371	23937
JFreeChart	1.0.9	561	81538
Jython	2.2.1	995	83763
Spring	2.5.1	2011	98938

Table 3. Target systems

unknown The type of the partial program is `unknown`, which means that PPA could not infer anything about this type. This is the case of the method's target in the first statement.

hierarchy correct The type in the partial program is a supertype or a subtype, depending on the type direction in the type fact, of the type in the complete program. For example, if $dt(\texttt{\$r1}) = \texttt{CharSequence}$ at line 1 and $dt(\texttt{\$r1}) = \texttt{String}$ at line 2, we say that the two types are *hierarchy correct*.

erroneous All other cases. Erroneous types can be inferred when we combine conflicting type facts or when we use certain syntax heuristics.

We only compared the short name of the types. The ability to infer the fully qualified name of a type solely depends on the project coding convention: if a project such as Lucene or Jython allows the usage of wildcard import statements, most inferred types will have a `p-unknown` package. Because a short name, given the context in which it is used, is often sufficient to uniquely identify a type, we preferred to classify as correct, types with an unknown package that matched the short name of a real type.

Finally, we chose to compare the Jimple intermediate representation of the partial program and the complete program because (1) this is the typical representation used to perform data flow analysis, (2) this provides a reasonable estimate of the results we would obtain if we performed the comparison at the AST or bytecode level (the transformation from Jimple to AST or bytecode is more straightforward than the transformation from AST to bytecode), and (3) there are fewer statement types in Jimple than node types in a Polyglot AST which makes the comparison easier and more robust.

6.2 Quality

We executed our implementation of partial program analysis on one class at a time without its dependencies, for all classes in our four target systems. Table 4 shows the results of PPA. There are three main sections in the table corresponding to the three configurations we used to execute PPA: (1) our baseline configuration (type inference and method binding disabled) [6], (2) type inference enabled and method

[6] Since our tool must build a properly constructed Polyglot AST in order to continue processing an entire class file, this is the minimal configuration we can enable. It uses the declared types that are available inside the class under analysis, plus syntax heuristics (Section 4) and it infers missing type

	Outcome	Lucene	JFreeChart	Jython	Spring
baseline	% correct	89.20	89.23	81.20	87.16
	% unknown	8.23	9.02	13.88	8.01
	% h. correct	0.29	1.12	1.91	1.12
	% erroneous	2.29	0.63	3.01	3.71
inf. no bind.	% correct	93.48	94.12	87.94	90.78
	% unknown	3.52	3.89	6.63	4.30
	% h. correct	0.34	1.16	2.36	1.20
	% erroneous	2.67	0.83	3.07	3.71
inf. bind.	% correct	93.80	94.40	88.22	90.97
	% unknown	2.46	3.56	6.21	4.07
	% h. correct	0.38	1.19	2.44	1.24
	% erroneous	2.71	0.85	3.13	3.72
	Total Facts	87706	250155	312907	325641

Table 4. Partial program analysis results

binding disabled, and (3) type inference and method binding enabled. For each of the configurations, the percentage of type facts in the Jimple IR that correspond to one of the four possible comparison outcomes is indicated below the target system. For example, with the baseline configuration, 89.20% of the type facts recovered by PPA were correct and 2.29% of the type facts were erroneous. The last line reports the total number of type facts in each complete system. For example, there were 250155 type facts in JFreeChart.

Our baseline configuration recovered most of the type facts in the partial programs (up to 89.23% in JFreeChart). Thus, combining the declared types available for the class under analysis with the syntax heuristics and type member inference works reasonably well. However, this baseline configuration can be improved upon, and the results indicate that type inference provides most of the remaining improvement. In the best case (Jython), type inference enabled the recovery of 6.7% of correct type facts. Forcing method bindings had a much smaller impact on the precision of the results because in the best case (Lucene), it recovered only 0.32% of correct type facts.

Syntax heuristics were the largest contributor of erroneous type facts. In the worst case (Spring), 3.71% of the inferred type facts were erroneous because of the syntax heuristics. These errors are effectively unavoidable because most of the syntax construct ambiguities represent undecidable problems. Still, as future work, we could validate the assumptions behind our syntax heuristics on more systems to ensure that they are representative and minimize the potential for erroneous type facts.

The number of unknown type facts decreased significantly when we enabled type inference and method binding. Table 5 shows the distribution of the unknown types once we enabled these two parameters. On average, the type inference and method binding passes correctly recovered 52% of the types that were previously unknown. On average, only

members (Section 3.3).

Outcome	Lucene	JFreeChart	Jython	Spring
% correct	55.92	57.33	50.61	47.62
% h. correct	1.08	0.79	3.78	1.46
% erroneous	5.17	2.37	0.86	0.15
% unknown	37.83	39.50	44.75	50.77

Table 5. Unknown types distribution after the type inference and method binding passes

Target	From	To	# Revisions	# Classes
Lucene	149000	616506	1017	4800
JFreeChart	1	712	185	924
Jython	1	4011	1267	20609
Spring	2003-08-01	2008-02-24	7299	31101

Table 6. Target systems versions

1% of the unknown types were erroneously inferred by these two passes. This provides evidence that performing type inference and method binding is desirable.

Hierarchy correct type facts only accounted for a small portion of the total type facts. This suggests that even if our heuristics and type inference strategies are theoretically imprecise (i.e., we often infer that an the type of an expression is subtype or a supertype of a type `T`), in practice, they often recover the exact type. The small number of hierarchy correct and erroneous type facts introduced by type inference and method binding also indicates that conflicting type facts do not represent a serious threat to the precision of the results.

6.3 Analysis Input

Partial program analysis can be performed on one class or on a set of classes. Having access to multiple type declarations can potentially improve the precision of the analysis. Because the accessibility to source files may vary from one technique to the other, we devised three scenarios that are representative of current software engineering techniques. The first scenario assumes that the user of partial program analysis only has access to one class: this is the same scenario as the previous section. The second scenario assumes that the user has access to one class and all classes that are directly referenced by this class. Approaches that mine code from web repositories would typically have access to a subset of the direct dependencies. The third scenario assumes that the user mines version histories and has thus access to all files that were modified in the same change set.

To evaluate the second scenario, we took each class in a target system and computed their direct dependencies using the complete target system. For each class, we provided the source files containing the class and the direct dependencies to our tool, but we did not provide any dependencies that were contained in a jar file. We then compared the inferred type facts from the class in the partial program with the type

		Outcome	Lucene	JFreeChart	Jython	Spring
single	*baseline*	% correct	89.20	89.23	81.20	87.16
		% erroneous	2.29	0.63	3.01	3.71
	inf.	% correct	93.80	94.40	88.22	90.97
		% erroneous	2.70	0.85	3.13	3.72
dep	*baseline*	% correct	99.30	95.97	98.13	93.31
		% erroneous	0.33	0.23	0.27	2.64
	inf.	% correct	99.56	98.39	98.92	95.00
		% erroneous	0.26	0.29	0.29	2.67
cs	*baseline*	% correct	89.52	89.54	86.92	86.13
		% erroneous	2.00	0.56	2.97	4.91
	inf.	% correct	93.88	94.82	90.68	90.34
		% erroneous	2.62	0.72	3.24	4.91

Table 7. Partial program analysis inputs

	Strategy	Lucene	JFreeChart	Jython	Spring
single inf. no bind.	% Assign.	45.59	61.06	36.73	38.37
	% Return	13.41	6.10	52.93	31.12
	% Method	0.66	0.72	0.39	0.35
	% Condition	8.73	5.10	1.38	16.50
	% Binary	22.01	17.64	6.06	7.12
	% Unary	3.48	8.75	0.91	3.55
	Total facts	4571	6700	41521	14462
single inf. bind.	% Assign.	41.95	57.31	29.52	36.29
	% Return	12.25	5.73	42.53	29.42
	% Method	8.55	6.64	19.93	5.72
	% Condition	7.97	4.79	1.11	15.60
	% Binary	20.28	16.73	4.87	6.77
	% Unary	3.18	8.21	0.73	3.36
	Total facts	5006	7138	51675	15297

Table 8. Inference strategies results

facts from the class in the complete program, but we did not compare the type facts inferred in the direct dependencies.

For the third scenario, we first retrieved the change sets, i.e., files that were committed together, from the Subversion repositories of Lucene, JFreeChart and Jython and we recovered the change sets from the CVS repository of Spring using a standard change set inference technique [22]. For each change set, we computed the list of classes that (1) were changed or modified, and (2) still existed in the current version of the program. We obtained a collection of class sets taken from the *current version* of the program that we provided as input to our tool. For each change set, we compared the type facts inferred in all classes in the change set with the type facts from the same classes in the complete program. Table 6 shows the range of versions we mined for each target system, the number of change sets (revisions) containing Java source files related to the target system and the total number of classes that we analyzed.

For each of the three scenarios, we executed PPA with the three configurations used in Section 6.2: (1) baseline configuration, (2) type inference enabled and method binding disabled, and (3) type inference and method binding enabled. Table 7 shows the results of our analysis. The three main sections represent the three scenarios we evaluated: single class (*single*), one class with all direct dependencies (*dep*), and all classes in the same change set (*cs*). For each section, we report the percentage of correct and erroneous type facts for the first (*baseline*) and third configurations (*inf.*) of PPA.

In all cases, we omitted the results of the second configuration because there was no significant difference between it and the third configuration. The results for the first configuration are the same as Table 4.

Including the direct dependencies greatly increased the percentage of correct type facts PPA could infer. This high precision actually left little room for improvement from type inference. On average, 96% of the type facts were correct in the baseline configuration when including all direct dependencies as opposed to 86% correct type facts in our baseline configuration of single class analysis. We obtained fewer correct type facts when analyzing Spring because a subset of the direct dependencies was contained in jar files which were not supplied to the compiler. These results suggest that, when possible, retrieving a subset of the dependencies might be highly beneficial since adding the dependencies had a greater impact than type inference. Finally, because certain members were declared in an ancestor and were thus not accessible, we still inferred erroneous and unknown type facts.

Analyzing all classes in a change set did not significantly improve the precision of our results. On average, only 34% of the direct dependencies of a class were in the same change set. Usually, even if two related classes are in the same change set, the improvement might be minimal if we the one class only access a few members in the other class. Still, further analysis of the change sets results are required. For example, some files are changed more often than others: if the files that are frequently changed are also the ones that gives the best (or the worst) results when analyzed by our tool, the results will be highly biased toward these files. This could explain the decrease of precision for Spring (90.97% for single class analysis versus 90.34% for change set analysis).

6.4 Inference Strategies

Since we showed that type inference was beneficial, we were interested in analyzing the contribution of each inference strategy we devised and presented in Section 3.1. This information can be used to determine which strategies are worth implementing if PPA needs to be implemented in an existing technique. For each type fact that was processed in the worklist (see Figure 7), we recorded the inference strategy that caused its insertion in the worklist which provided a good estimation of the contribution of each strategy.

Table 8 shows the percentage of type facts that each of the six most popular inference strategies generated: the other inference strategies had a negligible contribution. Because

the ordering and the proportion of the inference strategies were similar for each input scenario, we only report the results when we analyzed one class at a time. The upper part of the table shows the proportion of each inference strategy when performing type inference without method binding and the lower part shows the results when performing type inference with method binding. The last line in each part indicates the number of type facts that were processed in the worklist. For example, when performing type inference and method binding on Lucene, the assignment inference strategy generated 41.95% of the type facts.

The assignment inference strategy was the largest contributor of type facts in all target systems except Jython. The return and binary inference strategies came second in two target systems each. Those three strategies contributed to 90% of the type facts when disabling method binding and 76% when enabling method binding. Because a strategy can also trigger the use of another strategy (e.g., we find the type of a field using the assignment strategy and then we use the method binding strategy because a method uses this field as a parameter), we were interested in the inference chains produced by our approach. We found the average inference chain length to be 1.02, meaning that generally, an inference strategy *does not* trigger the use of another strategy. We also found in a manual investigation of the inference chains for each inference strategy that there is no significant correlation between any two inference strategies.

6.5 Threats to Validity

The external validity of this study is limited by the fact that we only studied four programs. Because three of these programs are self-contained, i.e., they do not require external libraries, we studied the Spring Framework which requires 90 jar files to increase the scope of our evaluation. Our four target systems have a large number of lines of code and their purposes are different enough to be representative of many Java programs. The fact that the results were also relatively stable across all four programs suggests that results obtained with different systems would be similar.

Our unit of measurement to evaluate the precision of PPA was the number of correct type facts in the Jimple intermediate representation. This unit is a good indicator for techniques that use PPA to retrieve static type information (e.g., SemDiff), but it is not sufficient to evaluate the usefulness of our approach for client static analyses such as call graph generation and points-to analysis that use the IR produced by PPA. These client static analyses typically require a higher precision than the software engineering tools that currently use PPA. Researchers in both our groups and others will be able to do these sorts of experiments now that PPA is fully implemented and publicly available.

The parser that PPA uses takes as input well-formed Java source files. Hence, it was not possible to evaluate our approach against code snippets even if PPA does not require complete source files. Although it is possible that analyzing only code snippets could decrease the precision of our approach, we found during early experimentation that type inference did not often cross the method boundaries, so the performance of PPA should not be dramatically impacted.

Finally, when we analyzed the change sets, we only used the latest version of the classes for each change set as opposed to using the version of these classes at the time of the change set. We expect the results of our analysis to be representative because the set of direct dependencies should be relatively stable during the lifetime of a class.

7. Related Work

Static analysis tools typically assume that a complete and correct representation of the program is available. Our work is quite different in that we assume that only part of the program is available. To the best of our knowledge, we know no other research project, except the prototype used in PARSEWeb [19], that performs type inference and resolves syntactic ambiguities with such constraints. Unfortunately, it is not possible to provide a full evaluation of the PARSEWeb's prototype because it is not publicly available and the paper only describes two inference strategies the prototype used. Those strategies are equivalent to our return and method binding inference strategies.

Still, there are several techniques that deal with the parsing of incomplete programs. Two such examples include *fuzzy parsers* [13] which extract high level structures out of incomplete or syntactically incorrect programs, and *island grammars* [14] which parse snippets of code into islands (recognizable constructs of interest) and water (remaining parts). Knapen et. al. presented an approach for parsing C++ programs when missing some header files [12]. Their motivation was similar to ours, since they wanted to deal with situations where not all the code was available. They developed various semantic tests and heuristics, similar to the syntax heuristics we used and described in Section 4, to determine the nature of an ambiguous syntactic constructs. As opposed to PPA, this C++ parser does not try to infer the declared type of an AST node (like the inference strategies presented in Section 3) and it creates an unknown node when it cannot soundly determine the nature of an expression.

The programming system generator (PSG) enabled the creation of development environments that could handle the parsing and analysis of incomplete programs [6]. For example, the environment could list the possible types of an undeclared variable according to its context. The computation of the possible types was encoded into a grammar written by the PSG users: the inference rules had thus to be manually defined and only simple inheritance schemes (e.g., Pascal) were supported. Another difference is that the environment generated by PSG was only *suggesting* possible types: it was not making any definitive choice like PPA.

Finally, modern Integrated Development Environments (IDE) often include tools to execute or analyze snippets of

code. For example, the Java parser [7] in Eclipse is able to generate an Abstract Syntax Tree for incomplete programs, but it does not try to resolve syntax ambiguities and it does not provide any typing information when the declaration of a type is missing. The Scrapbook editor[8] tries to execute any snippet of code even if it is not included in a Java class. Again, this tool reports an error if it encounters an undeclared type.

In terms of inferring declared types for Java, Gagnon et. al. solved a related problem of finding declared types of local variables when starting from Java bytecode [10]. Although their approach also used type constraints to assign declared types, their setting is quite different since they have access to the complete program and type hierarchy, and there are no syntactic ambiguities in bytecode.

Other work, less directly related, includes static analyses techniques that aim to analyze only part of a program. These are often designed for software engineering applications, where it is too expensive to analyze the whole program. These techniques use an intermediate representation generated from the *complete program* where all type declarations are accessible. Examples of these techniques include *partial data flow analysis* [9,11] which uses a demand-driven approach to analyze only the relevant part of a whole program and *fragment analysis* [16, 17] which does a full analysis on a given fragment of the program, using summary information for the remainder.

8. Conclusions and Future Work

We presented *Partial Program Analysis*, a technique that builds a typed abstract syntax tree and a typed three-address intermediate representation that software engineering tools can use to get more precise type information than what syntactic analysis traditionally provides. We covered the two main challenges when analyzing partial programs in Java, the ambiguous language constructs and the determination of declared types, and we proposed type inference strategies and heuristics to solve these problems.

We performed an empirical study on four open source programs and found that, on average, Partial Program Analysis could uncover 91.2% of correct type facts when analyzing one class at a time and only produced 2.7% of erroneous type facts. This high precision suggests that partial program analysis is a viable approach to enable useful static analysis on incomplete Java programs.

Finally, the current implementation of our prototype is available online at http://www.sable.mcgill.ca/ppa.

Acknowledgements

The authors thank Eric Bodden and Ekwa Duala-Ekoko for their valuable comments on the paper. This project was supported by the Natural Sciences and Engineering Research Council of Canada.

[7] www.eclipse.org/jdt/core/index.php

[8] www.eclipsezone.com/eclipse/forums/t61137.html

References

[1] JFreeChart. `http://www.object-refinery.com/jfreechart/`.

[2] Jython. `http://www.jython.org`.

[3] Lucene. `http://lucene.apache.org`.

[4] Spring Framework. `http://www.springframework.org`.

[5] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 805–806, 2007.

[6] Rolf Bahlke and Gregor Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst.*, 8(4):547–576, 1986.

[7] Nicolas Bettenburg, Rahul Premraj, and Thomas Zimmermann. Extracting structural information from bug reports. In *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories*, pages 27–30, 2008.

[8] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 481–490, 2008.

[9] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.*, 19(6):992–1030, 1997.

[10] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *Static Analysis Symposium*, pages 199–219, 2000.

[11] Rajiv Gupta and Mary Lou Soffa. A framework for partial data flow analysis. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 4–13, 1994.

[12] Gregory Knapen, Bruno Laguë, Michel Dagenais, and Ettore Merlo. Parsing C++ Despite Missing Declarations. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, page 114, 1999.

[13] Rainer Koppler. A systematic approach to fuzzy parsing. *Softw. Pract. Exper.*, 27(6):637–649, 1997.

[14] Leon Moonen. Generating robust parsers using island grammars. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering*, page 13, 2001.

[15] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proc. of the 12th International Conference on Compiler Construction*, pages 138–152, 2003.

[16] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in

java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, 2004.

[17] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–252, 1999.

[18] Stephen M. Blackburn et al. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[19] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, 2007.

[20] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[21] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, 2006.

[22] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

Appendix A

Table 9 presents the inference rules that PPA applies in a partial program. The first column indicates the inference rule name, the second column shows the AST node templates to which the inference rule is applicable. The third column lists the type facts that are inferred.

In an AST node template, arbitrary Java expressions are represented by `w`, `x`, `y`, and `z`, and types are represented by t, $t1$, and $t2$. The operator *baseType* returns the base type of an array (e.g., *baseType*(`int[]`) = `int`). The operator *safest* returns the safest type, or the first type if both are as safe, as defined in Section 3.4. Although type facts are generated for each AST node for which we do not have access to the declared type, the type facts are only added to the worklist if they meet the safety criterion presented in Section 3.4. Finally, for the sake of brevity, we only present the relevant subset of the unary and binary operators.

```
1  // File A.java
2  class A {
3    void main() {
4      C.f1 = "hello";
5      C.f1 = C.f2;
6    }
7  }
8
9  // File B.java
10 class B {
11   void main() {
12     C.m1(C.f2);
13   }
14 }
```

Figure 9. A partial program

Appendix B

When analyzing multiple source files, errors and imprecisions can be propagated. For example, suppose that the classes `A` and `B` presented in Figure 9 are provided as input to PPA. The field `f2` is shared by the two classes. In class A, PPA finds that $dt($`f2`$) \sim$ `String` because $dt($`f2`$) <:$ $dt($`f1`$) :>$ `String` (inference chain produced at lines 4 and 5). PPA then concludes that at line 12, the method `C.m1(String)` is called. Unfortunately, this is not true since $dt($`f2`$)$ could be of any reference type (e.g., `List`) and the parameter of method `m1` could end up being "far" from the type `String`.

If the two files had been analyzed separately, PPA would have concluded that at line 12, the method `C.m1(Unknown)` was called. This is still imprecise, but not as misleading as the previous inference.

Inference Strategies	**Code**	**Generated Type Fact**
Assignment	`x = y;`	$\{\texttt{x}, dt(\texttt{x}), :> dt(\texttt{y})\}$ $\{\texttt{y}, dt(\texttt{y}), <: dt(\texttt{x})\}$

Return	t m1() {
	...
	return y;
	}
		$\{\texttt{y}, dt(\texttt{y}), <: t\}$

Method Binding	**void** m1() {
	...
	x = m2(y);
	}
	
	$t1$ m2($t2$ param1) {
	...
	}
		$\{\texttt{x}, dt(\texttt{x}), <: t1\}$
		$\{\texttt{y}, dt(\texttt{y}), <: t2\}$
		We assume that m2 declared after m1 is the binding selected by PPA.

Condition	**for** (y; x; z) **while** (x) **if** (x) x ? y : z	$\{\texttt{x}, dt(\texttt{x}), \texttt{boolean}\}$
Unary	x++ !y	$\{\texttt{x}, dt(\texttt{x}), \sim \texttt{int}\}$ $\{\texttt{y}, dt(\texttt{y}), \texttt{boolean}\}$
Binary	x + y w & z w | z	$\{\texttt{x}, dt(\texttt{x}), \sim dt(\texttt{y})\}$ $\{\texttt{y}, dt(\texttt{y}), \sim dt(\texttt{x})\}$ $\{\texttt{w}, dt(\texttt{w}), \sim dt(\texttt{z})\}$ $\{\texttt{z}, dt(\texttt{z}), \sim dt(\texttt{w})\}$ or $\{\texttt{w}, dt(\texttt{w}), \texttt{boolean}\}$ $\{\texttt{z}, dt(\texttt{z}), \texttt{boolean}\}$ if the expected return type of `&` or `|` is a boolean.
Array	x[y]	$\{\texttt{y}, dt(\texttt{y}), <: \texttt{int}\ \}$ $\{\texttt{x}, dt(\texttt{x}), dt(\texttt{x[y]})[]\ \}$ $\{\texttt{x[y]}, dt(\texttt{x[y]}), baseType(\texttt{x})\ \}$
Switch	**switch**(x)	$\{\texttt{x}, dt(\texttt{x}), <: \texttt{int}\}$
Conditional	x ? y : z;	$\{\texttt{y}, dt(\texttt{y}), \sim \texttt{z}\}$ $\{\texttt{z}, dt(\texttt{z}), \sim \texttt{y}\}$ $\{\texttt{x ? y : z}, dt(\texttt{x ? y : z}), safest(dt(\texttt{y}), dt(\texttt{z}))\}$ or $\{\texttt{y}, dt(\texttt{y}), <: t\}$ $\{\texttt{z}, dt(\texttt{z}), <: t\}$ if the expected type t of the conditional is known.

Table 9. Type Inference Rules