McGill University
School of Computer Science
Sable Research Group

Partial Program Analysis

Sable Technical Report No. 2007-6

Barthélémy Dagenais
Software Evolution Research Group
McGill University
Montréal, Québec, Canada

September 14, 2007

Contents

Abstract

Software engineers often need to perform static analysis on a subset of a program source code. Unfortunately, static analysis frameworks usually fail to complete their analyses because the definitions of some types used within a partial program are unavailable and the complete program type hierarchy cannot be reconstructed. We propose a technique, Partial Program Analysis, to generate and infer type facts in incomplete Java programs to allow static analysis frameworks to complete their analyses. We describe the various levels of inference soundness our technique provides, and then, we cover the main algorithm and type inference strategies we use. We conclude with a detailed case study showing how our technique can provide more precise type facts than standard parsers and abstract syntax tree generators.

1 Motivation

Software engineering researchers often perform simple static analyses such as computing and following use-def chains and building call graphs. Sometimes, however, they only have access to a subset of the program source code and their analyses are thus greatly hindered.

For example, researchers that mine software repositories [8] are typically interested only in the files that were modified between two revisions of a software. Because they need to perform their analysis on every single revision of the system, the complexity of their analysis must be proportional to the size of the change and not to the size of the program, if they want their work to complete in a reasonable time. Moreover, they usually do not have access to the compiled program corresponding to each revision.

With strongly typed languages such as Java, most parsers and compilers fail to reconstruct the complete type hierarchy in the presence of partial program source code and thus complain and report an error. This limitation greatly reduces the amount and the quality of the analyses that can be performed on partial programs even though:

1. It should be possible to perform some static analyses such as locals use-def chains without having a complete type hierarchy because such information is typically not required by these analyses.
2. Facts about a type can be inferred just by looking at how the partial program uses the type.

We call the ability to statically analyze a subset of a program source code *Partial Program Analysis* or PPA. In the following sections, we present the ideas behind this analysis in Java and its implementation in the Soot [6] static analysis framework. To lighten the writing, we will refer to any parser, compiler or framework such as Soot as *analysis framework*.

2 Related Work

Since the Java type system has been proved to be sound [1], most analysis frameworks stay away from partial program analysis because it is inherently unsound. Still, some research projects related to program fragments and partial type systems address similar problems.

In [5], researchers wanted to evaluate the test coverage of all receiver classes and target methods at polymorphic call sites. To compute this coverage, they needed to perform whole-program analyses such as Rapid Type Analysis (RTA) and points-to analysis, but for performance reasons, they wanted to cover only a fraction of a program. Thus, they devised a general scheme where they create a main method in each potentially interesting class that calls every method and copy every field of the class. Then, they use these main methods as entry points for their whole-program analyses. One prerequisite of their approach is that they must have access to the definition of any types required by their input classes. Since our work addresses the cases where the users do not control the input classes and do not have access to the definition of all types referenced in the input, this approach cannot be used.

Work also have been done on performing type inference using local information [4] or partial evaluation [7], but in all cases, the prerequisite is the access to the type definition.

3 Partial Programs

Partial Program Analysis aims at helping an analysis framework to reconstruct a complete type hierarchy when receiving as input a subset of a program source code. For this to be realizable, we must make the following assumption:

A1: Given a subset of a program P's source code, we assume that program P's complete source code can compile without any error.

Indeed, there is no point in generating and inferring type facts if the way types are used in the partial program, our only source of information, is flawed. We argue this is a reasonable assumption, especially when mining software repositories: a widely accepted convention is to commit code to the repository only if the developer's workspace compile without an error. If we assume that the full program compiles, partial program analysis can then be divided into two problems:

P1: What type facts need to be generated so the analysis framework can reconstruct the complete type hierarchy and complete its analysis?

Type facts are needed almost everywhere in a Java program. All references to a type by its name are discussed in the Java Language Specification [2] (JLS), section 4.11. Such references include import statements at the beginning of a Java file, extend clauses, variable declarations or catch statements, to name only a few.

Unfortunately, referring to a type by its name is only one instance where an analysis framework needs to interact with the type system. Here is a non exhaustive list of situations where type facts can also be required:

- When the analysis framework encounters a method call made on an object, the framework needs to access the definition of this method to produce the correct binding.
- When a method is accessed in a static way, the analysis framework might check if the method is declared to be static.
- When an object is passed as a parameter to a known method, the analysis framework might want to ensure that the object is a subclass of the formal parameter type.
- In a catch statement, the analysis framework might want to ensure that the referenced type is a subtype of the `Throwable` type.
- When an object is assigned to another object, the analysis framework might want to check if the implicit cast is valid.

Since we assume the program compiles, we need to generate the required type facts, such as method declarations, and ensure that all checks implemented by the analysis framework pass.

P2: What type facts can we infer once the partial program has been parsed and checked?

Generating the fact that an unknown type `t` contains a certain method `m()` because it is called in the partial program is only one side of the problem. What can we say about the unknown type's hierarchy? If a known method is overloaded, can we determine which one is called? Can we infer the type of an unknown field? Can we infer the return type of an unknown method?

This problem is generally known as type inference. But as opposed to complete programs written in a strongly type language we, cannot use standard constraint-based type inference because most

of the type facts are expected to be missing. It follows that we can perform two kinds of type facts inference:

Sound type fact inference
We expect this kind of inference to be very rare. For example if an `int` is assigned to an unknown field and the unknown field is assigned to an `int`, we can then soundly infer that the unknown field is an `int`. But if there is only one assignment, the inference becomes unsound because it could be a short or a long (or even a `java.lang.Integer`) depending on the side of the assignment.

Unsound type fact inference
We expect this kind of inference to be more common. Still, we can differentiate the unsound type fact inference results in three categories:

1. Correct hierarchy-related inference
This kind of inference will always be correct, but the inferred type might not be the one defined in the missing source code (i.e, formal type). Let us consider those three examples:

Example 1:

```
super.age = 2;
```

In this example, if the definition of the super class is unknown, we can infer that `super.age` is a primitive. When we generate the final code, we need at some point to choose a primitive. The obvious choice would be in this case an `int`, but it is possible that it is in fact a `long`. We say that the inference is correct because this is definitively a primitive and this is not a `boolean`, but the real type might be in the hierarchy of the inferred type (in this case, a super type of `int`).

Example 2:

```
Bird b1 = ...
Bird b2 = ...
b1.singFor(b2);
```

In this example, we can infer that the type `Bird` declares a method `singFor()` that accepts a `Bird` type as input. Again, we say that the inference is correct (indeed, there is one method accessible from the `Bird` type that accepts a parameter that can be a bird), but the `singFor()` method might be declared in a super type of `Bird` and it could accept a super type of `Bird` as a parameter (hierarchy-related).

Example 3:

```
Bird b1 = unknownObject.unknownMethod();
unknownObject.unknownMethod().eat();
```

In this example, we first infer that `unknownMethod` returns an object in the hierarchy of `Bird`. In the second statement, we infer that the method `eat()` is called on an object with a type in the hierarchy of `Bird`. In other words, we will generate the fact that the method `Bird.eat()` is called. Strictly speaking, this might be wrong since it is possible that `unknownMethod()` returns a subclass of `Bird` (e.g., `BlueJet`) and that the `eat()` method is only declared within the subclass. However, the `eat()` method is still declared in a type within the descendants of `Bird`, so we conclude that this is a correct hierarchy-related inference.

2. Unknown types inference
Sometimes, we expect to be unable to generate any fact about a type. Let us consider field `f1` in this example:

```
unknownObject.f1 = unknownObject.unknownMethod(...);
```

In this case, where everything is unknown, we will simply generate the fact that the type of `f1` is unknown.

3. Possibly incorrect inference
Finally, we might want to be aggressive when doing type inference even if this could lead to incorrect type facts. See Section 6-E for an example.

When writing the final output, the type inference soundness level should be clear.

4 Partial Analysis

In this section, we cover the main strategies we devised to solve the two problems presented in Section 3. We also briefly present the technical framework into which we implemented those strategies.

4.1 Named and Anonymous Unknown Types

When analyzing a partial program, the analysis framework can come across two kinds of unknown types, i.e., types for which we do not have the definition:

Example:

1. `Bird bird = new Bird();`
2. `System.out.println(bird.property1);`

If we know the name of a particular type but do not have access to its definition as it is the case with `Bird` in the first line, we call it a *Named Unknown Type* since we at least know its short name (and maybe its fully qualified name as we will see in the next subsection). We also know that it is not a primitive nor the reference version of a primitive since those are restricted and cannot be extended as specified by the JLS. Finally, we also know that the type is not an array or a subclass of a final class. For example, it cannot substitute a `String`.

If we do not know the name of a particular type as it is the case with `property1` in the second line, we call it an *Anonymous Unknown Type*. In this particular case, it can be anything: a primitive, an array, a `String`, a Named Unknown Type, etc. Since the analysis framework needs a fully qualified name even for unknown types, we chose to use `MMAGICPPACKAGE.MAGICCLASS` as the name of such types. This is the only type name that is generated in our analysis: even if we could generate other intermediate types, this would not be practical if the user of our analysis expects to get real types from the analyzed program.

4.2 Generating type facts

The main strategy when generating type facts is to try to be as efficient and permissive as possible by doing as little type inference as possible.

4.2.1 Type definition

The first kind of facts that we need to generate for the analysis framework is the type's fully qualified name. As we will see in the following example, we can encounter four different situations:

```
package bar;

import foo.*;
import bar.baz.Bird;
```

```
class BlueJet extends Bird {
  private test.Dog dog;
  private Animal a;
...
```

First, when the analysis framework encounters the `Bird` class name, we infer that its fully qualified name is `bar.baz.Bird` because there is an *explicit import statement*.

Second, when the analysis framework encounters the `Dog` class name, we infer that its fully qualified name is `test.Dog` because its fully qualified name is used in the declaration.

Third, when the analysis framework encounters the `Animal` class name, we cannot infer its fully qualified name because it could be:
`Animal` (default package)
`foo.Animal` (because of the import all statement)
`bar.Animal` (it would be in the current package)

In this particular case we will generate the fact that the fully qualified name of `Animal` is `MMAGICP-PACKAGE.Animal` telling the analysis framework that we could not make a sound assumption.

Fourth, if the `import foo.*` statement was not part of the above example, we could have generated the fact that the fully qualified name of Animal was in fact `bar.Animal`. Even if this would be an unsound assumption (Animal could be in the default package), this default behavior is desirable since explicitly importing a class in the same package is not a common practice. For example, Eclipse removes this kind of explicit imports when auto-organizing import statements. Moreover, using a class in the same package is probably more frequent then using a class in the default package. In all cases, the default behavior should be configurable.

4.2.2 Type Members

When dealing with unknown types, the analysis framework often needs to get facts about their members (fields and methods). Let us consider the following example, assuming that we do not have the definition of the class `Bird`:

```
1: Bird b = new Bird(''Twitibird TM'');
2: System.out.println(b.age);
3: b.singWith(new Bird(''BigBird TM'',2),''Happy Birthday'');
```

Here, we need to generate the following facts:

1. The `Bird` type has at least two constructor declarations, one that takes as input a `String` (unsound but hierarchy related inference on line 1) and one that accepts one `Bird` and one `String` (unsound but hierarchy related inference on line 3).
2. The `Bird` type has at least one method declaration named `singWith()` that takes as input a `Bird` and a `String` (unsound but hierarchy related inference on line 3). The return type of `singWith()` is an anonymous unknown type.
3. The `Bird` type has a field named `age` which type is an anonymous unknown type (unsound and unknown inference on line 2).

Basically, we create the members as the analysis framework encounters them and we use only the type facts that are at hand. The return type of an unknown field and an unknown method is always an anonymous unknown type. Thus, if the same unknown method is used elsewhere with the same parameters' type and is assigned to a variable, we do not need to regenerate the method declaration because we already allowed any return types.

4.2.3 Caveats

Unfortunately, the syntax of the Java programming language can be ambiguous and make type facts generation a hard problem. Let us consider the following example:

```
1: import foo.Bar.Animal;
2:
3: class Bird extends Animal {
4:
5:   public method m1(...) {
6:     Property.doThis();
7:   }
8:
9: }
```

The first import statement illustrates a problem that was unfortunately always present in the previous examples: is `foo.Bar` a class, meaning that `Animal` is a static inner class or is `foo.Bar` a package? If we do not have access to the different packages or to the `Bar` class definition, we need to make a guess. One strategy would be to use the Java naming convention (which precludes the use of upper case in the case of package) to determine the nature of `foo.Bar`, but not all Java programs follow it. Another strategy is to always take the guess that it is a package until proved otherwise (for example, by looking at the way the class is instantiated or if the `Bar` class is used or instantiated itself). In our solution, we preferred the latter since in our experience, the use of inner class outside of its declaring class is rarer than not following the Java naming convention. In any case, the fully qualified name of the type will be the same.

The second problem illustrated at line 6 can dramatically change the facts that we generate. Indeed, is `doThis()` a static method of the `Property` class or is `Property` a field of the super class `Animal`? Both alternatives are possible because (1) the `Property` class is not explicitly imported, and (2) the current type extends a named unknown type. Again, we can rely on the Java naming convention or simply choose to treat the method as a static one until proved otherwise (for example, if `Property` is used in an assignment or as a method parameter). We chose the latter to be consistent with the previous heuristic.

4.3 Inferring type facts

4.3.1 Type inference Strategies

There are a lot of different statements, operations and requirements in the Java programming languages that can be used to infer type facts. The following two examples are relevant examples of the type inference strategies that we implemented:

a) Assignment
If one side of an assignment is unknown, we can use the type of the other operand to do the inference. The known operand side is important because it defines a type constraint. Let us consider the following example:

```
1: int temp = bird.age;
2: bird.name = ‘‘Twitibird TM’’;
```

In the first line, we know that the type of age must be less or equals to an `int`: it can be a `short` or an `int`. In the second line, we know that the type of name must be greater or equals to a `String`: it can be any ancestors of `String` or the `String` type itself. An assignment can also be used to infer the return type of an unknown method.

b) Parameter binding
The parameter types of a method can be used either to infer the type of unknown parameters or to infer the correct binding for an unknown method. Let us consider the following example:

```
1: ...
2: method1(2,bird.age);
3: ...
4:
5: private void method1(int p1, int p2)
6: private void method1(String p1, String p2)
```

From the call to `method1` at line 2, we can infer two type facts. First, because the first parameter is an `int`, only the method at line 5 can be called. Second, this method declares that the second parameter is an `int`, so the unknown field `age` must be an `int`.

4.3.2 Combining Strategies

Inference strategies can be combined as illustrated by the next example:

```
1: ...
2: int temp = method1(bird.age, bird.number);
3: ...
4:
5: private int method1(int p1, int p2) {}
6: private String method1(String p1, String p2) {}
```

Here, we first use the assignment inference strategy to determine that the return type of `method1` must be equals or less than an `int`. Only the first declaration of `method1` fulfills this requirement. Then, we use the parameter binding inference strategy to determine the type of the `age` and `number` unknown fields.

4.3.3 Merging type inference facts

When applying type inference strategies, we can gather conflicting facts about the same type and thus, encounter one of those three situations:

Decidable conflict
This is the case when type constraints can be resolved as in the following example:

```
1: Object o = bird.name;
2: String s = bird.name;
```

Here, we get the two following constraints: `name`'s type must be less or equals to an `Object` and less or equals to a `String`. Since the `String` type satisfies both constraints, we infer that `String` is the type of the `name` field.

Undecidable conflict
This is the case when we cannot get enough facts about the involved types to resolve the constraints.

```
1: dog.friend = new Bird();
2: dog.friend = new Cat();
```

In this example, we do not have access to the definition of `Bird` and `Cat` types. Thus, even if we know that they are part of the same hierarchy because of the two assignments, we do not know if one is a subclass of the other, who their first common ancestor is or what interface they share. There are three strategies that we can use to deal with this situation. The first strategy involves keeping in memory those various constraints (that the `friend` field must be a subclass of `Bird` and `Cat`). A second strategy would be to declare that the `friend` field is unknown. Finally, another strategy would be to keep the first or the last constraint and ignore the other, until a stronger inference can be made. We chose to implement the last solution because we hypothesize that a partial program will not provide sufficient facts to use the various constraints gathered and that, in the end, we will need to make the same choice as if we did not collect those various constraints.

Erroneous conflict
This is the case where constraints cannot be resolved because the underlying code would not compile (contradicting our main assumption). Here is such an example:

```
1: bird.name = new Object();
2: String s = bird.name;
```

In the first line, `bird.name` cannot be a `String` even if in the second line, it must be a `String`. If we encounter this case, we report an error.

4.4 Implementation details

4.4.1 Extending Polyglot

We implemented our solution in the Soot static analysis framework [6] and more precisely in the Polyglot compiler framework [3]. Soot uses Polyglot to build abstract syntax trees from Java source files and then, transforms those trees into Jimple, a 3-address intermediate representation. Unfortunately, when providing Polyglot with incomplete programs, exceptions are thrown because the type hierarchy cannot be completed. Indeed, Polyglot aims for full compliance with the Java Language Specification and needs to perform various validations involving type checking.

Because of this requirement, the first part of the problem, generating type facts, needed to be implemented into Polyglot to ensure that it would build the abstract syntax trees without throwing exceptions. To achieve this goal, we modified the two following classes:

SourceResolver
This class is responsible for finding and parsing referenced classes. Typically, Polyglot makes multiple requests to SourceResolver when it encounters an unknown type. For example, if the class `foo.bar.Baz` is imported, Polyglot will make three requests in the following order: one for `foo`, one for `foo.bar` and one for `foo.bar.Baz` (in the case that `Baz` is an internal class). Each of these requests can throw an exception, but only the last one can make Polyglot crash. We thus needed to modify the SourceResolver API to know when the request was the last one and act accordingly by creating the requested type.

TypeSystem
This class is responsible for providing method or field instance according to certain requests. For example, when a method call is parsed, Polyglot will make a request to the TypeSystem to know which method should be bound according to the parameter types and the target class. We needed to modify this class to generate the methods and fields of unknown types.

For the second part of the problem, inferring type facts, we had the choice of implementing the solution at the AST level in Polyglot or at the Jimple level in Soot. The latter has the advantage of simplifying the analysis since we would deal with shorter and simpler statements (no call chains for example). On the other hand, once new facts about a type are discovered, we need to modify the code representation accordingly. We argue that this would not be efficient at the Jimple level because of the intermediate variables introduced in the 3-address intermediate representation. Type inference is hence done at the AST level.

4.4.2 The algorithm

Here is our algorithm implementing those two sides of the solution (generating type facts and type facts inference):

- While building ASTs
  - Generate type facts.
  - Put and merge inferred type facts into the worklist.
- First pass - Type inference
  - Mark and index unsafe nodes.
  - While the worklist is not empty
    - Make nodes safer.
    - Put and merge new inferred type facts into worklist.
- Optional - Second pass - Method binding and inference
  - Mark and index unsafe nodes.
  - Bind all unsafe methods, put and merge new inferred type facts into worklist.
  - While the worklist is not empty

* Make nodes safer.
* Put and merge new inferred type facts into worklist.

The first part of the algorithm is performed while parsing the Java source code and building the abstract syntax trees. At this stage, most unknown methods and fields refer to our magic type. The facts that we gather about the unknown types (e.g., there is a call to a method `eat()` on an object of type `Bird`) are put in a worklist and will be the seed of the type inference step.

Once all ASTs are built, we visit all nodes to mark and index the ones that are *unsafe*. A node, such as a method call, is considered to be unsafe if it refers to an unknown type (anonymous or not). A node can be made unsafe because of more than one element. For example, if a method contains two parameters of an unknown type, the method is marked as unsafe and is indexed twice. Then, the type facts in the worklist are processed. Every nodes indexed by a type in the worklist is reprocessed (by the "make nodes safer" operation). If, while transforming a node new type facts are inferred, the new data is added and merged into the worklist.

In the two first stages, it is possible that a method call might be bound to multiple method declarations. Let us consider this example:

```
1: System.out.println(unknown.field1);
2: method1(unknown.field1);
3: ...
4: private void method1(boolean b) {}
5: private void method1(Bird b) {}
```

In this example, the method `println` is overloaded and thus, `field1` could be of any type (a `boolean`, an `Object`, a `String`, etc.). Deciding the method call binding too early might produce incorrect inference that could cascade. It follows that we defer the binding of these kinds of method calls until no other inference can be done. Then, in the second inference pass, we arbitrarily choose one binding and infer type facts from it.

The algorithm can be run in three modes:

Mode 1: Generate-only
In this mode, we only parse the source code and do not infer type facts. This is the fastest mode, but also the least precise.

Mode 2: Isolated type system
In this mode, we try to infer type facts but we do not share these facts across classes that are parsed by Polyglot. This mode is expected to be slower than the previous one, but also more precise. Another advantage of this mode is that it restricts aggressive but possibly wrong type inferences to the class being analyzed, preventing false facts to pollute other classes.

Mode 3: Shared type system
In this mode, we share any inferred facts with all parsed classes. This mode is expected to the slowest one but also the most precise depending on the application or the context in which partial program analysis is used. For example, when mining software repositories, files that are changed are often related which might significantly increase the amount of type inference we can do.

4.4.3 An extensible implementation

In our implementation of partial program analysis, we decoupled the reporting of type facts from the rest of the algorithm so new type inference strategies could be added easily: the PPA framework is responsible for merging this new facts and sending it to the nodes that might be made safer. The framework is also responsible for generating and keeping the node indexes up to date: indeed, when new facts about a type are processed, we want to visit only the nodes that might refer to this type. Once the nodes are made “safer”, the indexes often need to be regenerated since the nodes now refer to other the new type.

5 Experimental Framework

Partial program analysis must be validated against three different variables:

Robustness
The main motivation behind partial program analysis is to make an analysis framework accepts an incomplete program without “crashing”. Our implementation should then be able to accept any incomplete program as long as it compiled in its complete version.

Correctness
We devised multiple type inference strategies with different degrees of soundness. We need to ensure that our implementation respects those strategies and the intended soundness.

Precision and performance tradeoff
Because we propose different modes, we need to evaluate the tradeoffs between performance and precision.

To evaluate correctness, we wrote a series of case studies, i.e., incomplete Java programs where we associated for each line and each mode an expected result. We discuss those case studies in the next section.

To evaluate robustness and precision, we will apply partial program analysis on all the revisions of a Java program using a repository mining framework. More precisely, we intend to apply the three modes of this analysis on all revisions of the Java Development Environment in the Eclipse platform and report the number of nodes that were made more precise in each case. We will also select a random sample of the revisions to evaluate the correctness of the type inference by comparing the types and bindings of the partial program analysis with the types and bindings of the compiled program at those versions.

6 Results

A
In the three modes, two constructors should be generated for the named unknown type `package1.package2.Animal`, one with no parameter and one with a `String`.

B
Since the `Dog` type is not explicitly imported, the fully qualified name of this named unknown type should be `MAGICPACKAGE.Dog`.

C
In mode 2 (isolated type inference), there is no way that the type for `super.age` could be inferred. Thus, when producing the Jimple code, the first possible method binding is chosen (which should be `println(boolean)`).

In mode 3 (shared type inference), the type of `super.age` should be an int (because of line 9 in BlueJet.java). The correct binding for the println method should then be `println(int)`.

D
`super.oldAge` should be an int in mode 3 since an `int` (`super.age`) is assigned to this unknown field (assignment inference strategy).

E
Because of the assignment inference strategy, the `doSomething` method should return a `String`. Since there is only one known definition that fulfills this requirement, the first parameter must be a `String`. Thus, the super.name unknown field is a `String` (parameter binding inference strategy). This inference is possibly incorrect because the `Application` class is extending the unknown class `Bird`: it might be possible that Bird defines another method that returns a `String` and that accepts a different parameter.

F
Because of the assignment inference strategy, we can conclude that the unknown field `timeToFlee` is an `int`.

G
In this case, we should generate the fact that the `Dog` class defines a method called `getSpeed`. In mode 2 and 3, we should also infer that the method return type is a `double` using the assignment inference strategy.

H
This case is a good example of type merging. The `nickname` unknown field should be a `String` because of the second assignment.

I
The analysis framework should generate the fact that the `Dog` class defines a method called chase that accepts one parameter. In mode 2 and 3, we should also indicate that the method accepts a `String` as a parameter.

J
Again, because of the assignment inference strategy, the analysis framework should determine that the return type of `doSomething` should be a `String`. Using the parameter binding inference strategy, it should then infer that the `fullName` unknown field is a `String` and the child unknown field is a `Bird`. Like E, this inference is possibly incorrect.

K

Because of the assignment inference strategy, the analysis framework should determine that the return type of `doSomething` is an int. Using the parameter binding inference strategy, it should then infer that the `getAge` method should return an int and the second call chain should return an int. This means that a method `getDecay()` returning an anonymous unknown type should be created for the type Dog and a method `getInt()` returning an int should be created for this unknown type.

L

Because of the assignment inference strategy, the type of the `age` unknown field should be an `int`. Note that `BlueJet` and `Application` are both subtypes of the `Bird` class and `age` is a field of the `Bird class`. Thus, in mode 3, both classes refer to the same field, `age`, as an `int`.

7 Conclusion

The main contributions of this project include both the partial program analysis at the conceptual level and its extensible implementation. We clearly defined the theoretical and implementation problems associated to incomplete programs and devised solutions to overcome these issues. Because the whole problem of partial program analysis is unsound and undecidable, we categorized our type inference strategies according to their different level of soundness. This allows PPA to be used in various contexts.

As future work, we identified several areas for improvement. First, we would like to increase the robustness our implementation of PPA, i.e., ensure that it can parse any incomplete program. We would also like to continue our work on inference strategies. Finally, we would like to leverage the Soot tagging facility to provide type inference information (e.g. level of soundness) on statements that were made “safer”.

A Annex A - Java Source Files

```
package package1;

import package10.*;
import package1.package2.Animal;
import package3.Bird;
import java.io.PrintStream;

public class Application extends Bird {
  public static void main(String[] args) {
    // A- Generation of two constructors
    Animal animal = new Animal();
    Animal animal2 = new Animal(args[0]);
    System.out.println(animal.toString());
    System.out.println(animal2.toString());
  }

  // B- MAGICPACKAGE.Dog
  public void flee(Dog dog) {
    PrintStream printer = System.out;

    // C- println(int) if mode 3
    // C- println(unknown) if mode 2
    printer.println(super.age);

    // D- oldAge should be an int in mode3
    super.oldAge = super.age;

    // E- doSomething(String)
    String salutations = doSomething(super.name);
    printer.println(salutations);

    // F- super.timeToFlee = int
    super.timeToFlee = doSomething(10);

    // G- getSpeed should return a double
    double speed = dog.getSpeed();

    // H- super.nickName must be String
    Object obj = super.nickname;
    String s = super.nickname;

    // I- chase(String)
    dog.chase(super.nickname);

    // J- doSomething(String,Bird)
    s = doSomething(super.fullName,super.child);
```

```
    // K- getAge() should return int, getDecay() should return magic class
    // getInt should return an int!
    int var = doSomething(dog.getAge(),dog.getDecay().getInt());
  }

  public String doSomething(String name) {
    return "Hello " + name;
  }

  public int doSomething (int age) {
    return age * 10;
  }

  public String doSomething(String name, Bird child) {
    return name + " is the parent of " + child.toString();
  }

  public int doSomething(int age, int increment) {
    return age + increment;
  }
}
```

```
package package1.package2;

import package3.Bird;

public class BlueJet extends Bird {
  public void decay() {
    // L - super.age is an int
    int ageTemp = super.age;
    ageTemp = ageTemp + 10;
    super.age = ageTemp;
  }
}
```

B Annex B - Jimple Source Files

```
// package1.Application.jimple - MODE 1
public class package1.Application extends package3.Bird
{
  public static void main(java.lang.String[])
  {
    java.lang.String[] args;
    package1.package2.Animal animal, $r0, animal2, $r1;
    java.lang.String $r2, $r4, $r6;
    java.io.PrintStream $r3, $r5;
    args := @parameter0: java.lang.String[];
    $r0 = new package1.package2.Animal;
    specialinvoke $r0.<package1.package2.Animal: void <init>()>();
    animal = $r0;
    $r1 = new package1.package2.Animal;
    $r2 = args[0];
    specialinvoke $r1.<package1.package2.Animal: void
<init>(java.lang.String)>($r2);
    animal2 = $r1;
    $r3 = <java.lang.System: java.io.PrintStream out>;
    $r4 = virtualinvoke animal.<java.lang.Object: java.lang.String
toString()>();
    virtualinvoke $r3.<java.io.PrintStream: void
println(java.lang.String)>($r4);
    $r5 = <java.lang.System: java.io.PrintStream out>;
    $r6 = virtualinvoke animal2.<java.lang.Object: java.lang.String
toString()>();
    virtualinvoke $r5.<java.io.PrintStream: void
println(java.lang.String)>($r6);
    return;
  }

  public void flee(MMAGICPPACKAGE.Dog)
  {
    package1.Application this;
    MMAGICPPACKAGE.Dog dog;
    java.io.PrintStream printer;
    MMAGICPPACKAGE.MagicClass $r0, $r1, $r2, $r3, $r5, $r6, $r7, $r8, $r9;
    java.lang.String salutations, s;
    int $i0;
    java.lang.Object obj;
    this := @this: package1.Application;
    dog := @parameter0: MMAGICPPACKAGE.Dog;
    printer = <java.lang.System: java.io.PrintStream out>;
    $r0 = this.<package3.Bird: MMAGICPPACKAGE.MagicClass age>;
    virtualinvoke printer.<java.io.PrintStream: void
println(boolean)>($r0);
```

```
    $r1 = this.<package3.Bird: MMAGICPPACKAGE.MagicClass age>;
    this.<package3.Bird: MMAGICPPACKAGE.MagicClass oldAge> = $r1;
    $r2 = this.<package3.Bird: MMAGICPPACKAGE.MagicClass name>;
    salutations = virtualinvoke this.<package1.Application:
java.lang.String doSomething(java.lang.String)>($r2);
    virtualinvoke printer.<java.io.PrintStream: void
println(java.lang.String)>(salutations);
    $i0 = virtualinvoke this.<package1.Application: int
doSomething(int)>(10);
    this.<package3.Bird: MMAGICPPACKAGE.MagicClass timeToFlee> = $i0;
    virtualinvoke dog.<MMAGICPPACKAGE.Dog: MMAGICPPACKAGE.MagicClass
getSpeed()>();
    obj = this.<package3.Bird: MMAGICPPACKAGE.MagicClass nickname>;
    s = this.<package3.Bird: MMAGICPPACKAGE.MagicClass nickname>;
    $r3 = this.<package3.Bird: MMAGICPPACKAGE.MagicClass nickname>;
    virtualinvoke dog.<MMAGICPPACKAGE.Dog: MMAGICPPACKAGE.MagicClass
chase(MMAGICPPACKAGE.MagicClass)>($r3);
    $r5 = this.<package3.Bird: MMAGICPPACKAGE.MagicClass fullName>;
    $r6 = this.<package3.Bird: MMAGICPPACKAGE.MagicClass child>;
    virtualinvoke this.<package1.Application: java.lang.String
doSomething(java.lang.String,package3.Bird)>($r5, $r6);
    $r7 = virtualinvoke dog.<MMAGICPPACKAGE.Dog: MMAGICPPACKAGE.MagicClass
getAge()>();
    $r8 = virtualinvoke dog.<MMAGICPPACKAGE.Dog: MMAGICPPACKAGE.MagicClass
getDecay()>();
    $r9 = virtualinvoke $r8.<MMAGICPPACKAGE.MagicClass:
MMAGICPPACKAGE.MagicClass getInt()>();
    virtualinvoke this.<package1.Application: java.lang.String
doSomething(java.lang.String,package3.Bird)>($r7, $r9);
    return;
  }

  public java.lang.String doSomething(java.lang.String)
  {
    package1.Application this;
    java.lang.String name, $r3;
    java.lang.StringBuffer $r0, $r1, $r2;
    this := @this: package1.Application;
    name := @parameter0: java.lang.String;
    $r0 = new java.lang.StringBuffer;
    specialinvoke $r0.<java.lang.StringBuffer: void <init>()>();
    $r1 = virtualinvoke $r0.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>("Hello ");
    $r2 = virtualinvoke $r1.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>(name);
    $r3 = virtualinvoke $r2.<java.lang.StringBuffer: java.lang.String
toString()>();
    return $r3;
```

```
  }

  public int doSomething(int)
  {
    package1.Application this;
    int age, $i0;

    this := @this: package1.Application;
    age := @parameter0: int;
    $i0 = age * 10;
    return $i0;
  }

  public java.lang.String doSomething(java.lang.String, package3.Bird)
  {
    package1.Application this;
    java.lang.String name, $r3, $r5;
    package3.Bird child;
    java.lang.StringBuffer $r0, $r1, $r2, $r4;
    this := @this: package1.Application;
    name := @parameter0: java.lang.String;
    child := @parameter1: package3.Bird;
    $r0 = new java.lang.StringBuffer;
    specialinvoke $r0.<java.lang.StringBuffer: void <init>()>();
    $r1 = virtualinvoke $r0.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>(name);
    $r2 = virtualinvoke $r1.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>(" is the parent of ");
    $r3 = virtualinvoke child.<java.lang.Object: java.lang.String
toString()>();
    $r4 = virtualinvoke $r2.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>($r3);
    $r5 = virtualinvoke $r4.<java.lang.StringBuffer: java.lang.String
toString()>();
    return $r5;
  }

  public int doSomething(int, int)
  {
    package1.Application this;
    int age, increment, $i0;

    this := @this: package1.Application;
    age := @parameter0: int;
    increment := @parameter1: int;
    $i0 = age + increment;
    return $i0;
  }
```

```
  public void <init>()
  {
    package1.Application this;

    this := @this: package1.Application;
    specialinvoke this.<package3.Bird: void <init>()>();
    return;
  }
}
```

```
// package1.package2.BlueJet.jimple - MODE 1
public class package1.package2.BlueJet extends package3.Bird
{
  public void decay()
  {
    package1.package2.BlueJet this;
    int ageTemp;
    this := @this: package1.package2.BlueJet;
    ageTemp = this.<package3.Bird: MMAGICPPACKAGE.MagicClass age>;
    ageTemp = ageTemp + 10;
    this.<package3.Bird: MMAGICPPACKAGE.MagicClass age> = ageTemp;
    return;
  }

  public void <init>()
  {
    package1.package2.BlueJet this;
    this := @this: package1.package2.BlueJet;
    specialinvoke this.<package3.Bird: void <init>()>();
    return;
  }
}
```

```
// package1.Application.jimple - MODE 2
public class package1.Application extends package3.Bird
{
  public static void main(java.lang.String[])
  {
    java.lang.String[] args;
    package1.package2.Animal animal, $r0, animal2, $r1;
    java.lang.String $r2, $r4, $r6;
    java.io.PrintStream $r3, $r5;
    args := @parameter0: java.lang.String[];
    $r0 = new package1.package2.Animal;
    specialinvoke $r0.<package1.package2.Animal: void <init>()>();
    animal = $r0;
    $r1 = new package1.package2.Animal;
    $r2 = args[0];
    specialinvoke $r1.<package1.package2.Animal: void
<init>(java.lang.String)>($r2);
    animal2 = $r1;
    $r3 = <java.lang.System: java.io.PrintStream out>;
    $r4 = virtualinvoke animal.<java.lang.Object: java.lang.String
toString()>();
    virtualinvoke $r3.<java.io.PrintStream: void
println(java.lang.String)>($r4);
    $r5 = <java.lang.System: java.io.PrintStream out>;
    $r6 = virtualinvoke animal2.<java.lang.Object: java.lang.String
toString()>();
    virtualinvoke $r5.<java.io.PrintStream: void
println(java.lang.String)>($r6);
    return;
  }

  public void flee(MMAGICPPACKAGE.Dog)
  {
    package1.Application this;
    MMAGICPPACKAGE.Dog dog;
    java.io.PrintStream printer;
    MMAGICPPACKAGE.MagicClass $r0, $r1, $r7;
    java.lang.String salutations, $r2, s, $r3, $r5;
    int $i0, $i1, $i2;
    java.lang.Object obj;
    package3.Bird $r6;
    this := @this: package1.Application;
    dog := @parameter0: MMAGICPPACKAGE.Dog;
    printer = <java.lang.System: java.io.PrintStream out>;
    $r0 = this.<package3.Bird: MMAGICPPACKAGE.MagicClass age>;
    virtualinvoke printer.<java.io.PrintStream: void
println(boolean)>($r0);
```

```
    $r1 = this.<package3.Bird: MMAGICPPACKAGE.MagicClass age>;
    this.<package3.Bird: MMAGICPPACKAGE.MagicClass oldAge> = $r1;
    $r2 = this.<package3.Bird: java.lang.String name>;
    salutations = virtualinvoke this.<package1.Application:
java.lang.String doSomething(java.lang.String)>($r2);
    virtualinvoke printer.<java.io.PrintStream: void
println(java.lang.String)>(salutations);
    $i0 = virtualinvoke this.<package1.Application: int
doSomething(int)>(10);
    this.<package3.Bird: int timeToFlee> = $i0;
    virtualinvoke dog.<MMAGICPPACKAGE.Dog: double getSpeed()>();
    obj = this.<package3.Bird: java.lang.String nickname>;
    s = this.<package3.Bird: java.lang.String nickname>;
    $r3 = this.<package3.Bird: java.lang.String nickname>;
    virtualinvoke dog.<MMAGICPPACKAGE.Dog: MMAGICPPACKAGE.MagicClass
chase(java.lang.String)>($r3);
    $r5 = this.<package3.Bird: java.lang.String fullName>;
    $r6 = this.<package3.Bird: package3.Bird child>;
    virtualinvoke this.<package1.Application: java.lang.String
doSomething(java.lang.String,package3.Bird)>($r5, $r6);
    $i1 = virtualinvoke dog.<MMAGICPPACKAGE.Dog: int getAge()>();
    $r7 = virtualinvoke dog.<MMAGICPPACKAGE.Dog: MMAGICPPACKAGE.MagicClass
getDecay()>();
    $i2 = virtualinvoke $r7.<MMAGICPPACKAGE.MagicClass: int getInt()>();
    virtualinvoke this.<package1.Application: int
doSomething(int,int)>($i1, $i2);
    return;
  }

  public java.lang.String doSomething(java.lang.String)
  {
    package1.Application this;
    java.lang.String name, $r3;
    java.lang.StringBuffer $r0, $r1, $r2;
    this := @this: package1.Application;
    name := @parameter0: java.lang.String;
    $r0 = new java.lang.StringBuffer;
    specialinvoke $r0.<java.lang.StringBuffer: void <init>()>();
    $r1 = virtualinvoke $r0.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>("Hello ");
    $r2 = virtualinvoke $r1.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>(name);
    $r3 = virtualinvoke $r2.<java.lang.StringBuffer: java.lang.String
toString()>();
    return $r3;
  }

  public int doSomething(int)
```

```
  {
    package1.Application this;
    int age, $i0;

    this := @this: package1.Application;
    age := @parameter0: int;
    $i0 = age * 10;
    return $i0;
  }

  public java.lang.String doSomething(java.lang.String, package3.Bird)
  {
    package1.Application this;
    java.lang.String name, $r3, $r5;
    package3.Bird child;
    java.lang.StringBuffer $r0, $r1, $r2, $r4;
    this := @this: package1.Application;
    name := @parameter0: java.lang.String;
    child := @parameter1: package3.Bird;
    $r0 = new java.lang.StringBuffer;
    specialinvoke $r0.<java.lang.StringBuffer: void <init>()>();
    $r1 = virtualinvoke $r0.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>(name);
    $r2 = virtualinvoke $r1.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>(" is the parent of ");
    $r3 = virtualinvoke child.<java.lang.Object: java.lang.String
toString()>();
    $r4 = virtualinvoke $r2.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>($r3);
    $r5 = virtualinvoke $r4.<java.lang.StringBuffer: java.lang.String
toString()>();
    return $r5;
  }

  public int doSomething(int, int)
  {
    package1.Application this;
    int age, increment, $i0;

    this := @this: package1.Application;
    age := @parameter0: int;
    increment := @parameter1: int;
    $i0 = age + increment;
    return $i0;
  }

  public void <init>()
  {
```

```
    package1.Application this;
    this := @this: package1.Application;
    specialinvoke this.<package3.Bird: void <init>()>();
    return;
  }
}
```

```
// package1.package2.BlueJet.jimple - MODE 2
public class package1.package2.BlueJet extends package3.Bird
{
  public void decay()
  {
    package1.package2.BlueJet this;
    int ageTemp;

    this := @this: package1.package2.BlueJet;
    ageTemp = this.<package3.Bird: int age>;
    ageTemp = ageTemp + 10;
    this.<package3.Bird: int age> = ageTemp;
    return;
  }

  public void <init>()
  {
    package1.package2.BlueJet this;
    this := @this: package1.package2.BlueJet;

    specialinvoke this.<package3.Bird: void <init>()>();
    return;
  }
}
```

```
// package1.Application.jimple - MODE 3
public class package1.Application extends package3.Bird
{
  public static void main(java.lang.String[])
  {
     java.lang.String[] args;
     package1.package2.Animal animal, $r0, animal2, $r1;
     java.lang.String $r2, $r4, $r6;
     java.io.PrintStream $r3, $r5;
     args := @parameter0: java.lang.String[];
     $r0 = new package1.package2.Animal;
     specialinvoke $r0.<package1.package2.Animal: void <init>()>();
     animal = $r0;
     $r1 = new package1.package2.Animal;
     $r2 = args[0];
     specialinvoke $r1.<package1.package2.Animal: void
<init>(java.lang.String)>($r2);
     animal2 = $r1;
     $r3 = <java.lang.System: java.io.PrintStream out>;
     $r4 = virtualinvoke animal.<java.lang.Object: java.lang.String
toString()>();
     virtualinvoke $r3.<java.io.PrintStream: void
println(java.lang.String)>($r4);
     $r5 = <java.lang.System: java.io.PrintStream out>;
     $r6 = virtualinvoke animal2.<java.lang.Object: java.lang.String
toString()>();
     virtualinvoke $r5.<java.io.PrintStream: void
println(java.lang.String)>($r6);
     return;
  }

  public void flee(MMAGICPPACKAGE.Dog)
  {
     package1.Application this;
     MMAGICPPACKAGE.Dog dog;
     java.io.PrintStream printer;
     int $i0, $i1, $i2, $i3, $i4;
     java.lang.String salutations, $r0, s, $r1, $r3;
     java.lang.Object obj;
     package3.Bird $r4;
     MMAGICPPACKAGE.MagicClass $r5;
     this := @this: package1.Application;
     dog := @parameter0: MMAGICPPACKAGE.Dog;
     printer = <java.lang.System: java.io.PrintStream out>;
     $i0 = this.<package3.Bird: int age>;
     virtualinvoke printer.<java.io.PrintStream: void println(int)>($i0);
     $i1 = this.<package3.Bird: int age>;
     this.<package3.Bird: int oldAge> = $i1;
```

```
     $r0 = this.<package3.Bird: java.lang.String name>;
     salutations = virtualinvoke this.<package1.Application:
     java.lang.String doSomething(java.lang.String)>($r0);
     virtualinvoke printer.<java.io.PrintStream: void
println(java.lang.String)>(salutations);
     $i2 = virtualinvoke this.<package1.Application: int
doSomething(int)>(10);
     this.<package3.Bird: int timeToFlee> = $i2;
     virtualinvoke dog.<MMAGICPPACKAGE.Dog: double getSpeed()>();
     obj = this.<package3.Bird: java.lang.String nickname>;
     s = this.<package3.Bird: java.lang.String nickname>;
     $r1 = this.<package3.Bird: java.lang.String nickname>;
     virtualinvoke dog.<MMAGICPPACKAGE.Dog: MMAGICPPACKAGE.MagicClass
chase(java.lang.String)>($r1);
     $r3 = this.<package3.Bird: java.lang.String fullName>;
     $r4 = this.<package3.Bird: package3.Bird child>;
     virtualinvoke this.<package1.Application: java.lang.String
doSomething(java.lang.String,package3.Bird)>($r3, $r4);
     $i3 = virtualinvoke dog.<MMAGICPPACKAGE.Dog: int getAge()>();
     $r5 = virtualinvoke dog.<MMAGICPPACKAGE.Dog: MMAGICPPACKAGE.MagicClass
getDecay()>();
     $i4 = virtualinvoke $r5.<MMAGICPPACKAGE.MagicClass: int getInt()>();
     virtualinvoke this.<package1.Application: int
doSomething(int,int)>($i3, $i4);
     return;
  }

  public java.lang.String doSomething(java.lang.String)
  {
     package1.Application this;
     java.lang.String name, $r3;
     java.lang.StringBuffer $r0, $r1, $r2;

     this := @this: package1.Application;
     name := @parameter0: java.lang.String;
     $r0 = new java.lang.StringBuffer;
     specialinvoke $r0.<java.lang.StringBuffer: void <init>()>();
     $r1 = virtualinvoke $r0.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>("Hello ");
     $r2 = virtualinvoke $r1.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>(name);
     $r3 = virtualinvoke $r2.<java.lang.StringBuffer: java.lang.String
toString()>();
     return $r3;
  }

  public int doSomething(int)
  {
```

```
    package1.Application this;
    int age, $i0;

    this := @this: package1.Application;
    age := @parameter0: int;
    $i0 = age * 10;
    return $i0;
  }

  public java.lang.String doSomething(java.lang.String, package3.Bird)
  {
    package1.Application this;
    java.lang.String name, $r3, $r5;
    package3.Bird child;
    java.lang.StringBuffer $r0, $r1, $r2, $r4;

    this := @this: package1.Application;
    name := @parameter0: java.lang.String;
    child := @parameter1: package3.Bird;
    $r0 = new java.lang.StringBuffer;
    specialinvoke $r0.<java.lang.StringBuffer: void <init>()>();
    $r1 = virtualinvoke $r0.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>(name);
    $r2 = virtualinvoke $r1.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>(" is the parent of ");
    $r3 = virtualinvoke child.<java.lang.Object: java.lang.String
toString()>();
    $r4 = virtualinvoke $r2.<java.lang.StringBuffer: java.lang.StringBuffer
append(java.lang.String)>($r3);
    $r5 = virtualinvoke $r4.<java.lang.StringBuffer: java.lang.String
toString()>();
    return $r5;
  }

  public int doSomething(int, int)
  {
    package1.Application this;
    int age, increment, $i0;

    this := @this: package1.Application;
    age := @parameter0: int;
    increment := @parameter1: int;
    $i0 = age + increment;
    return $i0;
  }

  public void <init>()
  {
```

```
        package1.Application this;

        this := @this: package1.Application;
        specialinvoke this.<package3.Bird: void <init>()>();
        return;
    }
}
```

```
// package1.package2.BlueJet.jimple - MODE 3
public class package1.package2.BlueJet extends package3.Bird
{
  public void decay()
  {
     package1.package2.BlueJet this;
     int ageTemp;

     this := @this: package1.package2.BlueJet;
     ageTemp = this.<package3.Bird: int age>;
     ageTemp = ageTemp + 10;
     this.<package3.Bird: int age> = ageTemp;
     return;
  }

  public void <init>()
  {
     package1.package2.BlueJet this;
     this := @this: package1.package2.BlueJet;
     specialinvoke this.<package3.Bird: void <init>()>();
     return;
  }
}
```

References

[1] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.

[3] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, 2003.

[4] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[5] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. In *Proceedings of the 25th International Conference on Software Engineering*, pages 210–220, 2003.

[6] V. Sundaresan, P. Lam, E. Gagnon, R. Vallée-Rai, L. Hendren, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON*, pages 125–135, 1999.

[7] A. Tomb and C. Flanagan. Automatic type inference via partial evaluation. In *Proceedings of the 7th international conference on Principles and practice of declarative programming*, pages 106–116, 2005.

[8] T. Zimmermann, A. Zeller, P. Weißgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.