OVERVIEW OF DISAMBIGUATING TYPES IN JAVA PARTIAL PROGRAMS

Yong Qi Foo

February 19, 2021

Summary. The insights presented on 5 Feb 2021 are detailed here. A few basic but widely-used inference rules for resolving type requirements are shown here. A primitive and somewhat clumsy (but working) implementation has been developed to resolve type requirements based on those basic rules. That implementation is posted on GitHub. The implementation still lacks many features that will be implemented in the future. More knowledge on how [1] and other works disambiguate types need to be acquired. The Java Language Specification for JDK11 needs to be studied.

1 BACKGROUND

The goal here is to be able to disambiguate Java Partial Programs (PPs) to enable their compilation into Intermediate Representations (IRs) for program analysis. As [1] pointed out, while complete and sound disambiguation of PPs is an undecidable problem, we can often trade correctness for increase precision.

[1] has an implementation that fully disambiguates classes in Java 1.4 (and is now deprecated), and JCoffee [2] has and implementation that diambiguates classes in Java 8, but are unable to disambiguate inner classes, generics, classes / functions as parameters, and lambda expressions. The goal of this entire project is to build on these works to create a program that can perform disambiguation for any Java PP for Java 11.

This is a rather complex problem, and several issues must be solved for the program to work, such as

1. Missing type/attribute/method declarations
2. Exception is never thrown

For now, we will focus on point 1.

2 APPROACH

Several insights surface on analysis of JCoffee [2].

1. Compiler terminates before all possible errors are shown, hence the need for more than one pass.
2. Most (if not all) possible errors can be understood from the source code (such as by doing partial type inference in [1])
3. `UNKNOWN` class generated from JCoffee clutters programs and produces sub-optimal structures during static analysis.

Thus, our approach to resolve missing type declarations will instead model itself after the partial type inference algorithm detailed in [1]. Our approach comprises four steps:

1. Some Java compiler front-end (like Polyglot) to obtain a typed Abstract Syntax Tree (AST)
2. From the typed AST, collect type information, consisting of (1) type definitions and (2) type requirements.
3. Resolve type requirements
4. Build surrounding stubs to complete the PP

Points 2 and 3 will be elaborated here.

3 TYPE DEFINITIONS

Information on what is defined needs to be obtained so that our PP compiler can determine what constructs have already been defined. As such, we will collect information on what we call Type Definitions. These are unambiguous constructs that have been defined in the PP. They consist of fully-defined types, fully-defined attributes, and fully-defined methods.

3.1 FULLY-DEFINED TYPES

These are classes / abstract classes / interfaces that have been declared (and consequently, defined), containing fully-defined attributes and/or methods and their implementations (for concrete classes). For example:

```
class Foo {
    int hello() { return 1; }
}
class Bar {
    A a;
}
```

Foo and Bar are fully-defined types.

3.2 FULLY-DEFINED ATTRIBUTES

These are attributes that have been declared in fully-defined types, where the names and types are given. For example:

```
class Foo {
    Foo f;
}
class Bar {
    A a;
}
```

show that `Foo.f` is a fully-defined attribute because it is contained in `Foo`, which is fully-defined. The type of `Foo.f` also happens to be fully-defined. `Bar.a` is a fully-defined attribute because it is contained in `Bar`, which is fully-defined. Even though the type of `Bar.a` is not fully-defined, it is referenced (its type has a name).

3.3 FULLY-DEFINED METHODS

These are the methods contained in fully-defined types, where the exact method signatures of all variants are provided. The types in the method signatures are all fully-defined, or referenced. For example, in:

```
class Foo {
    int hello() { return i; }
}
class Bar extends C {
    A meow(B b) { return b.woof() }
}
```

`Foo.hello` is a fully-defined method because it is contained in a fully-defined type. The types in the method signature happen to also be fully-defined. `Bar.meow` is also a fully-defined method as it is contained in a fully-defined type. Even though the types in the signature of `Bar.meow` are not fully-defined, they are referenced by an unambiguous name.

4 TYPE REQUIREMENTS

We also need to collect information of the types that are required for the PP to work, but have not been fully defined by the PP. These are what we call Type Requirements, which are constructs that have certain ambiguity to their behaviour, and must be added to the PP for the PP to be compilable. They consist of referenced types, unknown types, referenced attributes and referenced methods.

4.1 REFERENCED TYPES

These are types that have been referenced by a name somewhere in fully-defined types, but are not be fully-defined. These types must exist with its exact given name, but the specifics (attributes and methods and their corresponding types) are not given by the PP. For example, in:

```
class Bar {
    A a;
}
```

`A` is a referenced type because it is referenced by `Bar` (and therefore must exist), but `A` is not defined anywhere.

4.2 UNKNOWN TYPES

These are completely ambiguous types as they are never referenced—these types might actually resolve to a fully-defined or referenced type, or may be a completely separate type that was simply not referenced in the PP. For example, in

```
class Bar {
    A a;
    static void main(String[] args) {
        System.out.println(a.myCoolAttribute.anotherCoolAttribute);
    }
}
```

`a.myCoolAttribute` and `a.myCoolAttribute.anotherCoolAttribute` both resolve to unknown types because they are not declared with any type information anywhere.

4.3 REFERENCED ATTRIBUTES

These are attributes of referenced / unknown types. These attributes must exist in those types, but their types are unknown. For example, in

```
class Bar {
    A a;
    static void main(String[] args) {
        System.out.println(a.myCoolAttribute);
    }
}
```

`A.myCoolAttribute` is an attribute that must exist, but is of some unknown type.

4.4 REFERENCED METHODS

These are methods of referenced / unknown types. These methods must exist in those types, but the method signatures are unknown. For example, in

```
class Bar {
    A a;
    static void main(String[] args) {
        System.out.println(a.myMethod(1));
    }
}
```

`A.myMethod` is a referenced method, where the method signature is unknown.

5 RESOLVING TYPE REQUIREMENTS

Once type information has been collected, the type requirements can be resolved into the types, attributes and methods that will be built for the PP to work. These requirements can be resolved by using information on the operations done on those types. These operations operate on certain rules, which we will use to determine the behaviour of referenced or unknown types. For now, there are three broad categories of rules, being assignment, attribute, and method call rules.

Rule 5.1 (General Assignment) *For some assignment statement* `x = a;` *where* $type(x) = X$ *and* $type(a) = A$*, then* $X :> A$*.*

Corollary 5.1.1 *If* X *is final (non-extendable and by extension, fully-defined) then for the program to be disambiguated, it must be possible for* $X = A$*. Thus, if* A *is unknown, then* $X = A$*. Otherwise, the program cannot be disambiguated.*

Suppose we have the following program:

```
class Foo {
    A a;
    public static void main(String[] args) {
        String s = a.myAttribute; // Line 1
        String s = a;             // Line 2
    }
}
```

Because `java.lang.String` is final, it is clear that Line 1 resolves the type of `a.myAttribute` to be a `String`, therefore, `class A` must look something like:

```
class A {
    String myAttribute;
    // ...
}
```

However, Line 2 is impossible to disambiguate, since `class A` cannot be the `String` type and `class A` cannot extend `String`.

Corollary 5.1.2 *If A is fully-defined, for the program to be disambiguated, there must exist some direct super-type of A that is covariant to X.*

Suppose we have the following program:

```
class Foo extends B, implements C {
    public static void main(String[] args) {
        A a = new Foo();
    }
}
```

For the program to be compilable, $B <: A$ or $C <: A$, which are both possible. However, further suppose we have a new program

```
class Foo {
    public static void main(String[] args) {
        A a = new Foo();
    }
}
```

In this case, this program cannot be disambiguated as class Foo only extends the Object class, and it has been defined that neither the Object class nor the Foo class extend A.

Rule 5.2 (Variable Referencing) *For some expression `x.y`, `y` must be an attribute of `x`.*

In general, even though we can be certain that y is an attribute in $x.y$, we cannot be certain about x. Because the following statement $a.b.c$ could mean 'get the attribute c in the attribute b of instance a of class A', or 'get the attribute c of static inner class b of class a'. Temporarily, we assume that there are no inner classes in our PPs, so we take it to be unambiguous that in $a.b.c$, a has attribute b, which has attribute c.

Corollary 5.2.1 *If x is not declared as a local variable or attribute in the scope of reference, then x itself must be a class or interface, so y must be static.*

Corollary 5.2.2 *if x is an instance of some type X, then X must not be an interface, as interface cannot have attributes.*

Corollary 5.2.3 *For some expression `x.y = a`, $type(x)$ cannot be an interface as interfaces cannot have instance attributes, and static attributes are final in interfaces. Of course, as with Rule 1, $type(x.y) :> type(a)$*

Rule 5.3 (Method calling) *For some $x.m(y)$, there must exist some method in x that accepts any type contravariant to y.*

Corollary 5.3.1 *If x is of some DefinedType where*

$$x.m = \begin{cases} i_1 \to r_1 \\ i_2 \to r_2 \\ \vdots, \\ i_n \to r_n \end{cases}.$$

Then $\exists k, type(y) <: i_k, 1 \leq k \leq n$

Corollary 5.3.2 *If $type(x)$ is referenced or unknown, then for some method call $x.m(y)$, adding a new overloaded method $m :: type(y) \to Unknown$ would be sound.*

Rule 5.4 (Constructors) *If there exists a constructor for type X, then X is a concrete class.*

Overall, there exists many more inference rules, which will eventually be obtained from the Java Language Specification.

6 A PRIMITIVE TYPE RESOLVER

Based on these few rules, very basic partial programs can be disambiguated. Referring to the program developed here called JavaCPP (Java Compiler for Partial Programs) (just a temporary name), the type resolving module can resolve simple partial programs, given the type definitions and requirements. Two examples and their respective outputs from JavaCPP are shown below.

Example 1.1 input PP.

```
class Foo {
    A a;
    public static void main(String[] args) {
        Foo f = new Foo();
        f.a = new B();
        String s = f.a.doX(1).doY(false);
    }
}
```

Example 1.2 output stubs.

```
interface A {
    UnknownType1 doX(int i);
}
class B implements A { }
interface UnknownType1 {
    String doY(boolean b);
}
```

Example 2.1 input PP

```
class Main {
    public static void main(String[] args) {
        int a = new A().doX(1).doY("Hello").doZ(false);
    }
}
```

Example 2.2 output stubs

```
class A {
    UnknownType1 doX(int i) { return null; }
}
interface UnknownType1 {
    UnknownType2 doY(String s);
}
interface UnknownType2 {
    int doZ(boolean b);
}
```

It is very apparent that it is possible for all the UnknownTypes to be collapsed into one of the referenced types (for both PPs, the UnknownTypes can be collapsed into class A), or for Example 2.2, the two UnknownTypes can be collapsed into a single UnknownType. More work needs to be done to investigate these.

References

[1] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *Proceedings of the 23rd ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, 2008.

[2] P. Gupta, N. Mehrotra, and R. Purandare, "Jcoffee: Using compiler feedback to make partial code snippets compilable," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICME)*, 2020.