JCoffee: Using Compiler Feedback to Make Partial Code Snippets Compilable

1st Piyush Gupta  
Dept. of Computer Sc. and Enng.  
IIIT-Delhi  
New Delhi, India  
piyush16066@iiitd.ac.in

2nd Nikita Mehrotra  
Dept. of Computer Sc. and Enng.  
IIIT-Delhi  
New Delhi, India  
nikitam@iiitd.ac.in

3rd Rahul Purandare  
Dept. of Computer Sc. and Enng.  
IIIT-Delhi  
New Delhi, India  
purandare@iiitd.ac.in

Abstract—Static program analysis tools are often required to work with only a small part of a program's source code, either due to the unavailability of the entire program or the lack of need to analyze the complete code. This makes it challenging to use static analysis tools that require a complete and typed intermediate representation (IR). We present JCoffee, a tool that leverages compiler feedback to convert partial Java programs into their compilable counterparts by simulating the presence of missing surrounding code. It works with any well-typed code snippet (class, function, or even an unenclosed group of statements) while making minimal changes to the input code fragment. A demo of the tool is available here: https://youtu.be/O4h2g_nZQIs

Index Terms—Partial programs, Compiler feedback, Static analysis, Type inference, Java

I. INTRODUCTION & MOTIVATION

Static analysis is an indispensable tool to model a program's structure and analyze its behavior. Most static analysis tools work on an Intermediate Representation (IR) of the code, which is a trivial task for the compiler to build when the complete program is available at their disposal. However, when only a subset of the entire code is available, despite its syntactical correctness, static analysis tools refuse to operate on it due to ambiguities and references to undeclared constructs.

We define a partial program (PP) as a non-empty subset of an otherwise complete program (CP), which is unlikely to have any syntactic errors. A PP is incomplete because it may contain references to classes, methods, and variables whose declarations lie in CP but not in PP. Figure 1 illustrates one such code snippet. Given only the code for a Java class `Bar`, we call it a partial program since it is ambiguous to determine the type of variable `lbl`, the parameter type of function `doB(...)`, and the return type of function `doA()`.

The ability to analyze a partial program is advantageous in many scenarios where either the entire program is unavailable, or it is interesting to analyze only a recently-modified snippet of the entire program, such as:

- encountering missing/deprecated dependencies during maintenance of an old project,
- analyzing code snippets from forum threads and documentation files before incorporating them in a codebase,
- independently checking bug fixes before automatic patching, and

```java
class Bar {
    void main() {
        Foo foo = new Foo();
        foo.lbl = "Hello world";
        foo.doB(foo.doA());
    }
}
```

Fig. 1: A Partial program (PP).

- analyzing changes to code in web repositories to understand the evolution of source code.

In order to be able to analyze partial programs, it is essential to overcome the challenges, namely:

i. Resolving syntactical uncertainty that arises due to missing declarations. For example, `Foo.baz()` may denote both a call to a static function `baz()` of a class `Foo`; or a virtual method call to a member function `baz()` of an object `foo` of some class, and

ii. Assigning correct data types for local variables, method return values, and class-level fields to enable IR generation.

While complete and sound disambiguation of a partial program is an undecidable problem, it works in our favor that software engineering techniques can often trade some guarantees on correctness for increased precision [1]. Some of the inferences we make during disambiguation may not be totally sound, but they provide little to no threat to the validity of the use case at hand.

The contributions of this paper are:

- A novel approach for partial program analysis that infers missing code declarations based on the compiler's feedback during compilation in an iterative process. We focus on making little to no changes to the input snippet, but on adding other class/method definitions around it to complete the missing pieces.
- Our implementation - Java COmpiler For FEEdback (JCoffee) that realizes our approach.
- An evaluation of our tool on 9133 partial code snippets. JCoffee successfully disambiguated over 90% of the inputs, with an overall accuracy of 93.3%.
Our approach offers several benefits over existing methods. First, we do not need to propagate inferred information across errors since subsequent compile cycles automatically incorporate it. Second, we support a much broader use case as we do not need to scrape the web or maintain a repository of existing method signatures to match undeclared methods in input code. Lastly, as opposed to a simplified intermediate representation, we build the complete bytecode satisfying the strictest type guarantees of the compiler.

II. ABOUT JCoffee

JCoffee is based on the simple premise that Java compiler error messages are very verbose and, a lot of context needed to fix a given error is already present in its description.

Consider, for example, the PP in Figure 1. The errors generated during its compilation can be fixed by creating the class Foo, its member variable lbl, and functions doA() and doB() with the right signatures. Some of these errors and the corresponding fixed CP is shown in Figure 2. The CP is now used to generate a class file, an intermediate representation (IR), a program dependence graph (PDG), or any other representation required for static analysis.

Interestingly, for each error, the compiler points out the exact location of the cause, the keywords it was expecting, and the keywords it found. Moreover, the errors are reported hierarchically, the fine-grained errors are detected only after the higher level inconsistencies have been resolved.

Thus, in a use-case such as ours, where the syntactic correctness of the code snippet is assured, it would not be wrong to make decisions based solely on the suggestions put forth by the compiler.

Our only assumption apart from a well-typed input code is the availability of standard Java types, such as java.lang.*, or java.io.* during compile time. We view this as a reasonable assumption because they ship with standard compilers and are necessary to run Java programs. The various errors fixed by JCoffee can be classified as:

- Identifier errors
  - cannot find symbol: The commonest error, it is thrown whenever the compiler cannot find the declaration of an identifier (such as package, interface, class, method, constructor, or variable). JCoffee declares a new identifier with the corresponding missing signature.
  - array required but . . . found: This is thrown when there is an attempt to index a variable that has not been declared as an array. JCoffee infers this and modifies the declared type of the variable.

- Computation errors
  - incorrect method, . . . cannot be applied to . . .: This error results from an incompatibility between a method’s call and its declaration. JCoffee adapts the declaration to match the call signature.
  - operator . . . cannot be applied to . . . / incompatible types / inconvertible types: Some operators are only defined for specific types, and JCoffee uses these errors to infer the type of one of the participating operand whenever possible.

- Access to static entities
  - non-static variable/method . . . cannot be referenced from a static context: The ‘static’ modifier states that a variable/method is associated with a class, not individual objects. This helps JCoffee infer that the corresponding variable/method is to be labeled as ‘static’.

- Miscellaneous errors
  - for-each not applicable to expression type: ‘for-each’ is one of the many ways to iterate over an iterable in Java. This error indicates that the expression is an iterable, most likely an array. Repeated occurrences of this error in the same code fragment indicates the multidimensionality of the array.
  - exception . . . is never thrown in body of corresponding try statement: This error occurs when the code snippet is missing the exception declaration, and JCoffee erroneously made a class by the same name on encountering a missing symbol error earlier. This error is then used to declare an exception instead.

(a) Simplified error log.

(b) Fixed complete program. class UNKNOWN instantiates all objects whose type cannot be resolved.

Fig. 2: Errors and their fixes for a partial program.
III. ARCHITECTURE

At its core, JCoffee attempts to fix all the errors pointed out by the compiler in a deterministic manner, by simulating an environment of missing dependencies around the given code snippet. The tool consists of two modules, namely the Error Fixing Module (EFM) and the Intermediate Representation Generator (IRG). It also comprises the Driver Engine (DE) to integrate the two modules. Although the ideas presented are quite generic and may be implemented in any way suitable to the use case, we have implemented DE, EFM and IRG in python. We now discuss each component in detail.

a) Error Fixing Module (EFM): Given a code snippet and a list of compiler-generated error messages, EFM fixes each error by adding class/method/variable declarations or inferring and updating the missing return type/declared type for methods/variables. Once all errors have been handled, EFM returns the modified code.

Algorithm 1 JCoffee

function DE(code, maxSteps)
    code ← preprocess(code)
    for ctr ← 1 to maxSteps do
        nErr, errorMsgs ← compile(code)
        if nErr = 0 then
            IRG(code)
            return true
        end if
        code ← EFM(code, errorMsgs)
    end for
    return false
end function

function EFM(code, errorMsgs)
    modCode ← code
    for err in errorMsgs do
        modCode ← fixError(code, err)
    end for
    return modCode
end function

function IRG(code)
    outFile ← code
    compile(outFile) ▷ Used for static analysis
end function

b) Intermediate Representation Generator (IRG): Once all the compiler-generated errors have been fixed, IRG generates bytecode or the .class files. The bytecode can be used as it is or further intermediate representations may be generated depending on the individual use case.

c) Driver Engine (DE): The first step is to pre-process the input. If the input code snippet is not already a class (or a set of classes), it is encapsulated into a placeholder method and a class structure. DE then invokes javac, the Java compiler, on the input, to get a list of error messages which are sent to EFM. A successful compilation invokes IRG. Otherwise, it loops for a fixed number of iterations, with each EFM output becoming the input for the next invocation.

We outline the pseudocode for each of the components in Algorithm 1. The simplistic and modular design of JCoffee makes it easy to extend it to support a wider range of errors, keeping it compatible with future Java versions.

IV. EVALUATION

We evaluated our implementation of JCoffee to answer the following research questions:
(RQ1): How effective is JCoffee in disambiguating code snippets?
(RQ2): What is the accuracy of JCoffee in preserving the intended semantics of code during disambiguation?
(RQ3): What is the time complexity and overhead of the iterative approach used in JCoffee?

A. Dataset & Environment

We used 9133 partial code snippets from the BigCloneBench [2] dataset released by Svajlenko et al. Ranging from a few statements (3 lines) to functions as large as 920 lines of code, the snippets contained code for frequently used functionalities from open source java projects. As almost all test samples referenced undeclared objects and functions, less than 1% of the snippets were initially compilable.

The experiments were conducted on a server having OpenJDK Java 8, running Ubuntu 19.04 with 12 Intel Xeon E4 cores and 31 GB of memory.

B. RQ1: Effectiveness

JCoffee’s primary goal is to completely resolve all compiler errors to generate bytecode for the given partial code. With some samples producing as many as 291 errors, we had a success rate of just over 90% - JCoffee completely disambiguated 8220 samples out of the 9133 input snippets when allowed to run for up to 10 compiler iterations per input. Further, 94% of the total samples were reduced to 2 or fewer compiler errors, which may then be fixed manually.

Upon further inspection of snippets that JCoffee failed to fix, we found the following recurring constructs:

- inner classes
- Java generics
- classes/functions as parameters
- lambda expressions

We consider this to be a limitation of the current implementation, and plan to fix this in future versions.

C. RQ2: Accuracy

In some cases, the fix for a snippet does not completely capture the intended logic of the PP.

Consider, for example, the statement

dst.transfer(src, 0, src.size());

Here, src and dst are objects of class Channel. While the PP contains their declaration, it does not contain the definition of class Channel. So, JCoffee sets the 3rd parameter
type of `transfer()` and the return type of `size()` as `UNKNOWN`. While it is trivial for a programmer to guess that `size()` likely returns an `int`, JCoffee does not yet use this extra information available in variable & function names.

To evaluate this impact, we randomly selected 30 test snippets, and manually fixed them, considering language constructs wherever possible. We compared this golden standard with the fixes generated by JCoffee for the same partial programs. Our results show that 93.3% of the type inferences were identical, and in merely 6.7% of the cases, a human evaluator could make more precise assumptions.

Hence, one extension of our approach can be to extract linguistic information from identifier names wherever possible and benefit from it.

**D. RQ3: Time complexity & overhead**

The complexity of the algorithm is bounded by $n_s$, the number of statements in the source file, $n_e$, the average number of errors detected per pass, and $p$, the number of compile passes required to disambiguate the code completely. The time complexity can thus be expressed as $O(p \times (n_s + n_e))$.

Out of the fixed files, 91.3% required up to 5 compile iterations, while 99.5% were fixed within seven iterations. On average, it took 1.64s to fix a code snippet. We view this as an acceptable cost for complete disambiguation.

**V. RELATED WORK**

Several works exist that parse incomplete code to support static analyses. To the best of our knowledge, we know of no other research project aiming to generate compilable code for Java, or directly levering compiler feedback.

PPA introduced by Dagenais and Hendren [1] is the closest in terms of research objectives. It predicts unknown bindings for partial programs by performing partial type inference and uses heuristics to resolve ambiguities. However, it is compatible with only Java 1.4, and is now deprecated.

Melo et al. [3] develop PsycheC, with a similar goal as ours, but for the C language. They build a lattice structure for various standard C types to bind variables to them during constraint generation for type inference.

Zhong and Wang [4] propose GRAPA that locates previously released code archives, and extracts resolved types and method signatures to generate System Dependency Graphs (SDG) for partial programs. While suitable for a niche use case of publicly released applications, this approach fails for partial programs in general.

PARSEWEB [6], PRIME [7], and SemDiff [8] work similarly by recommending method calls with matching signatures from arbitrary frameworks or mining specifications for extracted API calls from the code snippet. Approaches such as [9] only try to resolve missing dependencies given explicit import statements, something which is rarely available in partial programs.

Further, many Integrated Development Environments (IDE) such as [10], [11] generate a typed abstract syntax tree for incomplete code snippets. However, they throw errors when they encounter constructs with missing declarations.

**VI. CONCLUSIONS AND FUTURE WORK**

We implemented JCoffee, a tool that makes use of the detailed error information by the compiler to generate bytecode for partial programs by simulating the presence of undeclared constructs. Based on our evaluations on thousands of open source partial code snippets, JCoffee completely removes errors from over 90% of input files, with the modifications being identical to the gold standard in 93.3% of the cases.

In the future, we plan to support complex Java 8 mechanisms and add capabilities to extract information from object names for a more precise type-inference. Also, since our approach is general, it can be adapted to other programming languages with descriptive errors, such as Rust and Elm.

Finally, the implementation of JCoffee, benchmarks, and outputs are available at [https://github.com/piyush69/JCoffee](https://github.com/piyush69/JCoffee).

**ACKNOWLEDGMENT**

This work is partly supported by Infosys Center for Artificial Intelligence at IIIT-Delhi, Department of Science and Technology (DST) (India), Science and Engineering Research Board (SERB), the Confederation of Indian Industry (CII), and Nucleus Software Exports Ltd.

**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, pp. 313–328.

[2] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, “Towards a big data curated benchmark of inter-project code clones,” in 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 2014, pp. 476–480.

[3] L. T. Melo, R. G. Ribeiro, M. R. de Araújo, and F. M. Q. Pereira, “Inference of static semantics for incomplete c programs,” Proceedings of the ACM on Programming Languages, vol. 2, no. POPL, pp. 1–28, 2017.

[4] H. Zhong and X. Wang, “Boosting complete-code tool for partial program,” in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017, pp. 671–681.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 9, no. 3, pp. 319–349, 1987.

[6] S. Thummalapanta and T. Xie, “Parseweb: a programmer assistant for reusing open source code on the web,” in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, pp. 204–213.

[7] A. Mishne, S. Shoham, and E. Yahav, “Typestate-based semantic code search over partial programs,” in Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 2012, pp. 997–1016.

[8] B. Dagenais and M. P. Robillard, “Recommended adaptive changes for framework evolution,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 20, no. 4, pp. 1–35, 2011.

[9] J. Ossher, S. Bajracharya, and C. Lopes, “Automated dependency resolution for open source software,” in 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, 2010, pp. 130–140.

[10] “Eclipse jdt core component,” [https://www.eclipse.org/jdt/core/index.php](https://www.eclipse.org/jdt/core/index.php) accessed: 2020-05-13.

[11] “Eclipse scrapbook pages feature,” [https://www.eclipsezone.com/eclipse/forums/rf61137.html](https://www.eclipsezone.com/eclipse/forums/rf61137.html) accessed: 2008-03-03.