Reconstruction of Incomplete Polymorphic Programs

Yong Qi Foo, Michael D. Adams and Siau-Cheng Khoo

11 July 2024

ABSTRACT. The ability to reconstruct the dependencies of incomplete programs to form complete and well-typed ones provides several benefits, including allowing static analysis tools to analyse incomplete programs where their surrounding dependencies are unavailable, and supporting stub generation and testing tools to work on snippets. However, earlier efforts to do so are unable to work with incomplete programs containing parametrically polymorphic types.

In this paper, we present a technique that receives an incomplete Java program that may contain parametrically polymorphic types, and reconstructs its surrounding dependencies to form a complete and well-typed program. We then present empirical results from our prototype implementation of this algorithm, JavaCIP, which outperforms prior works in this area.

Keywords: incomplete programs, program synthesis, type inference

1 INTRODUCTION

Programs encountered in practical contexts can be syntactically correct yet contain references to undeclared or unknown classes, methods or fields. Such instances are termed *partial* or *incomplete programs* in the literature [Dagenais, Hendrenis, 2008; Gupta et al., 2020; Melo et al., 2017]. Incomplete programs manifest in various scenarios, including during the planning, design, and prototyping phases of the software-development lifecycle, as well as in patches for code review, version control systems [Dagenais, Robillard, 2008], web repositories [Thummalapenta, Xie, 2007], bug reports [Bettenburg et al., 2008], forum threads and documentation. Despite their prevalence, incomplete programs pose challenges in areas such as program analysis, which plays a crucial role in tasks such as bug detection [Ayewah et al., 2007], language manipulation and optimization [Vallée-Rai et al., 2010] and feature location [Zhao et al., 2006]. These analyses frequently use an executable or intermediate representation (IR) of the program. However, compilers cannot generate IRs of incomplete programs because they refer to undeclared classes, methods and fields.

There are numerous works on this topic [Chugh et al., 2009; Godefroid, 2014; Knapen et al., 1999; Perelman et al., 2012; Rodrigues et al., 2019; Guimarães et al., 2019; Xue et al., 2022; Dong et al., 2022], and one technique for addressing this is the reconstruction of missing dependencies in incomplete programs [Dagenais, Hendrenis, 2008; Gupta et al., 2020; Melo et al., 2017]. This handles several practical challenges:

1. Several areas of analysis, such as functional code clone detection [Hua et al., 2021; Mehrotra et al., 2022], code search [Sun et al., 2022] and bug detection [Zhang et al., 2023] rely on program representations like control flow graphs, program dependence graphs, and instruction sequences, often derived from compiled representations. Since compilers only operate on complete programs, these analyses struggle when applied to partial sections of projects.

2. Other analyses, such as expertise-profile mining, involve extracting insights from large source-code repositories and analyzing diverse data sources like bug reports [Anvik, Murphy, 2007], project-change histories [Mockus, Herbsleb, 2002] and API-usage patterns [Mani et al., 2016]. Such data often involves projects with dependencies, which can be impractical to track down when analyzing a large number of projects [Williams, Hollingsworth, 2005]. Reconstructing

Addresses: **YONG QI FOO**, Email: yongqi@nus.edu.sg, Website: https://yongqi.foo/; **MICHAEL D. ADAMS**, Email: adamsmd@nus.edu.sg, Website: https://michaeldadams.org; **SIAU-CHENG KHOO**, Email: khoosc@nus.edu.sg, *National University of Singapore*.

missing dependencies in incomplete programs allows snippets from bug reports, code contributions, and projects to be transformed into complete programs, enabling standard techniques to extract relevant data.

3. Certain studies, such as studies on developer interactions [Wang et al., 2013], analyze source code posted on forum threads and message boards. Identifying the dependencies of these snippets is challenging, particularly when the dependencies have multiple versions or are not publicly available.

Previous research [Dagenais, Hendrenis, 2008; Gupta et al., 2020; Melo et al., 2017] has successfully reconstructed missing dependencies in incomplete programs. PPA [Dagenais, Hendrenis, 2008] and JCoffee [Gupta et al., 2020] have achieved this for Java programs, whereas PsycheC [Melo et al., 2017] has done so for C programs. However, these approaches are inadequate for handling *parametrically polymorphic* or *generic* types [Gosling et al., 2005]. This limitation is significant because many Java programs, particularly those utilizing the Java Collections Framework [Parnin et al., 2013], incorporate generic types. As a result, PPA and JCoffee cannot handle these programs. (PsycheC works only on C, which does not have generic types.)

In this paper, we present a technique for reconstructing the missing dependencies of incomplete polymorphic Java programs. Our algorithm receives an incomplete program that may refer to or depend on parametrically polymorphic types, and produces Java sources that make the combined program complete and well-typed. It does so without modifying the original incomplete program. The ideas presented in this paper have been implemented as a prototype called JavaCIP, and empirical evaluation shows that it outperforms prior works.

The remainder of this paper is organized as follows. Section 2 gives an overview of our algorithm via an example of how we might reconstruct by hand the dependencies of an incomplete program. Section 3 describes the technical details of our algorithm. In Section 4, we present our prototype implementation, JavaCIP, used for experimental evaluation in Section 5. Section 6 discusses limitations and avenues for future work. Section 7 reviews related work, and finally, Section 8 concludes the paper.

2 OVERVIEW

To gain an intuition for how our algorithm works, we attempt to replicate its effects by hand in an example. Observe that program $\mathcal{P}_1$ shown in Figure 1a is incomplete, by virtue of it referring to types `A`, `C` and `D` with no accompanying declaration. Our goal is to produce a program $\mathcal{Q}_1$ that declares these types such that $\mathcal{P}_1$ and $\mathcal{Q}_1$ combined form a complete and well-typed program.

2.1 Reconstructing Missing Declarations

We know from the outset that classes `A<T>`, `C` and `D` must exist. Thus, we add declarations of these classes to our program, giving us program $\mathcal{Q}_{1(a)}$ in Figure 1b. However, this is only the beginning. Firstly, as we see in line 6 of $\mathcal{P}_1$, the assignment statement `a = b` requires that the type of `b` (`B<D>`) is compatible in an assignment statement with the type of `a` (`A<? extends C>`). In other words, `B<D>` must be a *subtype* of `A<? extends C>`, written as `B<D> <: A<? extends C>`. From line 2 of $\mathcal{P}_1$, we know that `B<D>` is already a subtype of `A<D>`, hence, if `A<D> <: A<? extends C>`, then our original constraint holds. For `A<D> <: A<? extends C>` to be true, `D` must be a subtype of `C` (unlike languages like Scala [Odersky et al., 2006], Java does not have declaration-site variance). At this stage, no more judgements about the satisfiability of this condition can be made, because classes `C` and `D` are not declared in $\mathcal{P}_1$, and as yet, $\mathcal{Q}_{1(a)}$ does not indicate that `D <: C` is true. Fortunately, the fix for this is simple—if we allow class `D` to extend `C` in $\mathcal{Q}_{1(a)}$, then this constraint, and by implication,

```
1  import java.util.List;
2  class B<T> extends A<T> {
3    static void main() {
4      A<? extends C> a = new A<>();
5      B<D> b = new B<>();
6      a = b;
7      C c = new C();
8      D d = new D();
9      List<? super Integer> i = c.get();
10     List<? super String> s = d.get();
11   }
12 }
```

(a) Incomplete program $\mathcal{P}_1$

```
1 class A<T> { }
2 class C { }
3 class D { }
```

(b) Program $Q_{1(a)}$.

```
1 class A<T> { }
2 class C { }
3 class D extends C {
  }
```

(c) Program $Q_{1(b)}$.

```
1  class A<T> { }
2  class C {
3    τ1 get() {
4      return null;
5    }
6  }
7  class D extends C {
8    τ2 get() {
9      return null;
10   }
11 }
```

(d) Program $Q_{1(c)}$.

```
1  class A<T> { }
2  class C {
3    α1<τ11> get() {
4      return null;
5    }
6  }
7  class D extends C {
8    τ2 get() {
9      return null;
10   }
11 }
```

(e) Program $Q_{1(d)}$.

```
1  class A<T> { }
2  class C {
3    List<? super Integer> get() {
4      return null;
5    }
6  }
7  class D extends C {
8    List<? super String> get() {
9      return null;
10   }
11 }
```

(f) Program $Q_{1(e)}$.

```
1  class A<T> { }
2  class C {
3    List<? super Integer> get() {
4      return null;
5    }
6  }
7  class D extends C {
8    @Override
9    τ3 get() {
10     return null;
11   }
12 }
```

(g) Program $Q_{1(f)}$.

```
1  class A<T> { }
2  class C {
3    List<? super Integer> get() {
4      return null;
5    }
6  }
7  class D extends C {
8    @Override
9    List<Object> get() {
10     return null;
11   }
12 }
```

(h) Program Q_1.

Figure 1: A step-by-step example of reconstructing the dependencies of an incomplete program by hand.

our original constraint `B<D> <: A<? extends C>`, is satisfied. Therefore, we add this information into $Q_{1(a)}$, giving us $Q_{1(b)}$ in Figure 1c. Indeed, $Q_{1(b)}$ allows the original assignment statement of `a = b` in line 6 of $\mathcal{P}_1$ to be well-typed.

Lines 9 and 10 of $\mathcal{P}_1$ are more challenging to deal with, primarily because the types of `c.get()` and `d.get()` are completely unknown. A step towards making these lines well-typed is to create the relevant method declarations, and allow their return types to be some inference variables τ_1 and τ_2 respectively, so that we may begin to resolve constraints on them later. Adding these declarations to $Q_{1(b)}$ yields $Q_{1(c)}$ as shown in Figure 1d. Note that generating method bodies is out of the scope of this paper, and thus the method bodies we generate here return `null`.

2.2 Searching for Types

Let us temporarily focus on line 9 of $\mathcal{P}_1$ which now gives the constraint τ_1 <: `List<? super Integer>`. At this stage, it is still unclear as to what τ_1 might be—it could be a monomorphic class, a polymorphic class with one argument, two arguments and so on. This prompts us to begin a search over what τ_1 is. For now, we fix τ_1 as a polymorphic type with one unknown argument, giving us $\tau_1 = \alpha_1$<τ_{11}> for some single-argument polymorphic class α_1 and some type argument τ_{11}. If later we find this configuration to be unsatisfiable, we backtrack back to this point and try some other configuration, for example, by letting τ_1 to be some α_1 or α_1<τ_{11}, τ_{12}>, and so on. This revises our initial constraint, giving us α_1<τ_{11}> <: `List<? super Integer>` and program $\mathcal{Q}_{1(d)}$ as shown in Figure 1e.

We now have to decide what α_1 actually is. We know that α_1 must be a single-parameter polymorphic class that is already referred to by $\mathcal{P}_1$ and $\mathcal{Q}_{1(d)}$, or a completely new class whose declaration we shall add to $\mathcal{Q}_{1(d)}$. The constraint α_1<τ_{11}> <: `List<? super Integer>` we obtained earlier gives us an indication that α_1 might be `List`. Again as part of our search, we temporarily fix α_1 = `List` and continue onwards. As we continue to perform our search and expand the constraints further, eventually we arrive at α_1<τ_{11}> = `List<? super Integer>`; performing the same process of searching for a type for τ_2 gives us τ_2 = `List<? super String>` and program $\mathcal{Q}_{1(e)}$, as shown in Figure 1f. Indeed, $\mathcal{Q}_{1(e)}$ allows lines 9 and 10 in program $\mathcal{P}_1$ to be well-typed.

Readers familiar with the vast literature on type checking and type inference might have reservations towards the need for a *backtracking search*. Type inference algorithms for languages like ML or Haskell infer a *principal type* or *most general unifier* [Duggan, Bent, 1996]. However, the notion of *principality* is not applicable to our problem. For instance, if we find a type τ_1 to be a supertype of a type `B` and a subtype of a type `A`, τ_1 could indeed be either of these two types, and neither of these are more general than the other. Furthermore, classes in Java are open to extension, so if we were to initially discover that another unknown type τ_2 is a subtype of another type `C`, we cannot assert *a priori* that τ_2 *must* be equal to `C` for the program to be well-typed; it could very well be the case that later in the analysis we conclude that τ_2 must actually be equal to a newly-created type `D` that extends `C`.

2.3 Overriding Methods

Although we have dealt with all our initial constraints, programs $\mathcal{P}_1$ and $\mathcal{Q}_{1(e)}$ together is still not well-formed. Because we have required `D` to extend `C`, the method declaration of `get` in `D` must *override* the declaration of `get` in `C`, which in effect, means that the return type of `get` in `D` must be a subtype of the return type of `get` in `C`. What we must do is to generate a new return type for `get` in `D` that is a subtype of *both* `List<? super Integer>` *and* `List<? super String>`. To do so, we replace the current method declaration in `D` to one that returns a new inference variable τ_3 giving us program $\mathcal{Q}_{1(f)}$ in Figure 1g, and we constrain it to be a subtype of `List<? super Integer>` *and* `List<? super String>`. This brings us back to the process of searching for a type for τ_3 as we have done earlier for τ_1 and τ_2.

Eventually, after solving our new constraints on τ_3, we arrive at τ_3 = `List<Object>` and $\mathcal{Q}_1$ in Figure 1h. By this point, all constraints we could generate from both $\mathcal{P}_1$ and $\mathcal{Q}_1$ are satisfied, and as such, these two programs form a complete and well-typed program. We can now compile $\mathcal{P}_1$ and $\mathcal{Q}_1$ with Java's compiler to generate an IR of our overall program, enabling standard tools to perform analysis on $\mathcal{P}_1$.

2.4 Insights

In summary, we do the following to reconstruct the missing dependencies of an incomplete program $\mathcal{P}$:

1. Declare classes whose declarations are missing from $\mathcal{P}$ in $\mathcal{Q}$.
2. Obtain the constraints that must be satisfied for $\mathcal{P}$ to be complete and well-typed. We describe this in detail in Section 3.1.
3. Simplify constraints like `B<D>` <: `A<? extends C>` into one like `D` <: `C`, and modify the declarations in $\mathcal{Q}$ so that the constraint is solved. We describe this in Sections 3.2.1 and 3.2.2.
4. Search for a satisfiable type for the type of a method invocation, field or variable that is completely unknown. We describe this process in Section 3.2.3.
5. Ensure that any constraints (particularly on overriding methods) in $\mathcal{Q}$ are satisfied as described in Section 3.2.4.

3 RECONSTRUCTING MISSING DEPENDENCIES

To produce a complete and well-typed program out of an incomplete Java program, we proceed in two phases. First, we parse the incomplete program to obtain an abstract syntax tree (AST), and traverse the AST to generate constraints (Section 3.1). Then, we solve these constraints by reconstructing the missing dependencies as required (Section 3.2).

3.1 Constraint Generation

Our constraint generation process extends Java's constraint generation mechanism used during type checking [Gosling et al., 2005] by being able to generate constraints in the presence of missing declarations and unknown types. At a very high level, the constraint generation phase is described in Algorithm 1. The algorithm receives an incomplete program $\mathcal{P}$ and creates an empty stub $\mathcal{Q}$ and empty set of constraints C (line 2). After parsing $\mathcal{P}$ to obtain its AST (line 3), we search for each referenced but not declared type and declare it in $\mathcal{Q}$ (lines 4–6). Next, we generate types and constraints for nodes in the AST that cannot be typed due to missing dependencies (lines 7–11). Finally, we generate a typed AST of $\mathcal{P}$, and obtain more constraints for it and add them to C (lines 12–14).

We expand on lines 9 and 13 of Algorithm 1 in this subsection by first describing the constraints that we generate and solve (Section 3.1.1), how we generate constraints and types when the types of field accesses and method invocations cannot be determined due to missing dependencies (Section 3.1.2), and how we constrain types in relation to one another (Section 3.1.3).

3.1.1 *Constraints.* There are four kinds of constraints that we generate at this phase, the first two being fairly standard:

1. Equality constraints $S \equiv T$. These constrain the type S to be equal to the type T.
2. Subtype constraints $S <: T$. These constrain the type S to be a *subtype* of the type T as defined by the Java Language Specification [Gosling et al., 2005].
3. Field membership constraints in the form of $\mathit{hasFld}(S, \texttt{x} : T)$. These require that the type S has a field named `x` with type T.
4. Method invocation constraints $\mathit{hasInv}(S, R\ \texttt{m}(A_1, \ldots, A_n), \{P_1, \ldots, P_m\})$. These require that from an object of type S, we are able to invoke a method `m` with arguments A_1 to A_n in a context where P_1 to P_m are bound type parameters, and as a result, returns the type R.

Algorithm 1 Constraint Generation

Input: An incomplete program $\mathcal{P}$
Output: A configuration $\langle \mathcal{P}, \mathcal{Q}, C \rangle$ where C are the constraints that must be satisfied for $\mathcal{P}$ and $\mathcal{Q}$ to form a complete and well-typed program

```
1:  function ConstGen(P)
2:      Q ← ϵ; C ← ⊤                                   ▷ Q and C start empty
3:      a ← parse(P)                                    ▷ a is the AST of P
4:      for each type T in a do
5:          if T not declared in P or Q then
6:              Declare class T in Q
7:      for each field access and method invocation node N in a do
8:          if type of N cannot be determined then
9:              Generate type T and constraints C for N   ▷ Described in Section 3.1.2
10:             Set type of N as T
11:             C ← C ∧ C
        ▷ At this point, all nodes in a can be typed
12:     for each node N in a do
13:         Generate constraints C for N                  ▷ Described in Section 3.1.3
14:         C ← C ∧ C
15:     return ⟨P, Q, C⟩
```

Let us expand on the intuition behind field membership and method invocation constraints. These constraints get generated when field accesses/method invocations cannot be resolved due to ambiguity or missing dependencies. The $\mathit{hasFld}(S, \texttt{x} : T)$ constraint thus means that, at some point later during the reconstruction process when the type of S has been decided, the class declaration for S will either (1) already have declared a field `x` of some other type T', in which case we constrain T to be compatible with the declared type T' (they may not be equal due to polymorphism), or (2) not have a declaration of field `x`, in which case we will declare it.

The $\mathit{hasInv}(S, R\ \texttt{m(}A_1, \ldots, A_n\texttt{)}, \{P_1, \ldots, P_m\})$ might raise more questions, particularly, (1) why *hasFld* constraints assert the existence of a *declaration* for a field, while *hasInv* constraints only assert that a method can be *invoked*, and (2) what the set of bound type parameters $\{P_1, \ldots, P_m\}$ is for. Firstly, for reasons we describe in Section 3.1.2, even if the class declaration for S is available, we still might not know *which* declaration an invocation actually calls. Hence, we assert the ability to invoke a method first, then decide which method declaration it invokes later. Secondly, because methods can be polymorphic, invoking a method from the same type with the same arguments can result in a different return type if different *type arguments* are supplied. However, type arguments are often inferred, and thus when creating a new method declaration that supports an invocation, we must know what the accessible bound type parameters are so that we can reconstruct the inferred type arguments to the invocation.

3.1.2 *Dealing with Field Accesses and Method Invocations.* Field accesses like `E.x` and method invocations like `E.m(...)` are particularly problematic (the expression E is known as the *primary* in the Java Language Specification [Gosling et al., 2005]), because if the type of the primary does not have a declaration in the original program, or if it is unknown, then we do not know what the type of the field access or method invocation is. For example, in program $\mathcal{P}_2$ in Figure 2a, even though we know that `a1` has type `A<C>`, the types of expressions `a1.x`, `a1.x.y` and `a1.m(a2.x)` in lines 5 and 6 cannot be resolved.

Our goal at this stage is to be able to give a type for these field accesses and method invocations

```
1 class B<T, U> {
2   void main() {
3     A<C> a1 = new A<>();
4     A<D> a2 = new A<>();
5     Object o = a1.x.y;
6     String s = a1.m(a2.x);
7   }
8 }
```

(a) Incomplete program $\mathcal{P}_2$ with fields and methods.

```
1 class A<P1> {
2
3 }
4
5 class C { }
6
7 class D { }
```

(b) Reconstructed declarations of classes $\mathcal{Q}_2$.

Types
a1.x : τ_1[P1 ↦ C]
a1.x.y : δ_1
a2.x : τ_1[P1 ↦ D]
a1.m(a2.x) : τ_2
Constraints C_2
5: *hasFld*(A<P1>, x : τ_1)
5: *hasFld*(τ_1[P1 ↦ C], y : δ_1)
5: δ_1 <: Object
6: *hasInv*(A<C>, τ_2 m(τ_1[P1 ↦ D]))
6: τ_2 <: String
where $\tau_1 \in \mathcal{F}(\{\texttt{P1}\})$, $\tau_2 \in \mathcal{F}(\{\texttt{T}, \texttt{U}\})$

(c) Types and constraints derived from $\mathcal{P}_2$.

Figure 2: Incomplete program $\mathcal{P}_2$, and the result of ConstGen($\mathcal{P}_2$) as $\langle \mathcal{P}_2, \mathcal{Q}_2, C_2 \rangle$.

by adding some constraints to C. This enables these field accesses and method invocations to have some meaningful type for us to generate more constraints. This raises two questions: (1) What types should we assign to the types of missing field accesses and method invocations? (2) What constraints do we generate and under what circumstances?

Inference Variables. To answer the first question, we represent unknown types as *inference variables*, which correspond to *pre-types* or *type variables* in several other works [Melo et al., 2017; Dong et al., 2022]. These inference variables are temporary, and will be replaced with valid Java types at some point in the algorithm's execution, and they do not appear in the generated source code.

As we have described in Section 2, in the presence of both subtype and parametric polymorphism, we need to perform a search over the possible types a declaration or expression could have. Thus, some of the inference variables should carry with them information about their possible assignments, i.e., their search domains. To this end, we define three kinds of inference variables:

1. A τ-type represents a type whose search domain is known. For example, if we had ascribed an expression with the type $\tau \in \{S, T\}$, it means that we know the expression has type S or T.
2. An α-type represents some class type with known *arity* (the number of type parameters/arguments of a polymorphic type). For example, the type α<τ> is a class type with one type argument (has arity 1).
3. A δ-type represents a variable where nothing is known about it. These are replaced with other types as constraints on it are solved. (δ-types are essentially *type variables* which are frequently used in type checking and type inference [Duggan, Bent, 1996].)

τ-types are generated whenever the search domain is known, in particular, when the bound type parameters surrounding the context in which the type is present is known. For example, any field declaration in a class with type parameters P_1 to P_n must be equal to P_1 to P_n, or a class type that can only contain these parameters. Because this is so frequently the case, we define functions $\mathcal{F}$ and $\mathcal{F}_?$ that receives a set of type parameters $\mathbb{S}$ and produces a set of types and fresh inference variables defined as:

$$\mathcal{F}(\mathbb{S}) = \mathbb{S} \cup \{\alpha, \alpha\texttt{<}\tau_1\texttt{>}, \alpha\texttt{<}\tau_1, \tau_2\texttt{>}, \dots\}$$
$$\mathcal{F}_?(\mathbb{S}) = \mathcal{F}(\mathbb{S}) \cup \{?\} \cup \{\texttt{? extends } S \mid S \in \mathcal{F}(\mathbb{S})\} \cup \{\texttt{? super } S \mid S \in \mathcal{F}(\mathbb{S})\}$$

where for all i, $\tau_i \in \mathcal{F}_?(\mathbb{S})$, τ_i and α fresh.

Table 1: Generating constraints and types for field access.

Node	Circumstance	Generated Constraint	Type of Node
E.x	Type of E is $S\langle A_1, \ldots, A_n\rangle$, declared as class $S\langle P_1, \ldots, P_n\rangle$ in $\mathcal{Q}$	$\mathit{hasFld}(S\langle P_1, \ldots, P_n\rangle, \mathtt{x} : \tau)$ where $\tau \in \mathcal{F}(\{P_1, \ldots, P_n\})$, τ fresh	$\tau[P_1 \mapsto A_1, \ldots, P_n \mapsto A_n]$
E.x	Type of E is an inference variable T	$\mathit{hasFld}(T, \mathtt{x} : \delta)$ where δ is fresh	δ

Example 3.1. $\mathcal{F}(\{S, T\}) = \{S, T, \alpha, \alpha\langle\tau_1\rangle, \alpha\langle?\ \mathbf{extends}\ \tau_1\rangle, \alpha\langle?\ \mathbf{super}\ \tau_1\rangle, \alpha\langle?\rangle, \alpha\langle\tau_1, \tau_2\rangle, \ldots\}$ where for all i, $\tau_i \in \mathcal{F}(\{S, T\})$, τ_i and α fresh.

Now that we have described the possible types we could ascribe to unknown field accesses and method invocations, we proceed to describe the exact types and constraints we generate for each.

Field Accesses. There are two possible scenarios that limit our ability to determine the type of a field access E.x for some primary expression E and field name x:

1. The type of E is a concrete (not unknown) type whose declaration is not in $\mathcal{P}$, and therefore, is declared in $\mathcal{Q}$. Since we know the class declaration which the declaration of field x belongs to, we can immediately determine what the search domain of the type of x is—if E has type $S\langle A_1, \ldots, A_n\rangle$ where S is declared as a class $S\langle P_1, \ldots, P_n\rangle$ in $\mathcal{Q}$, then it must be the case that the type of x as declared in S is a fresh $\tau \in \mathcal{F}(\{P_1, \ldots, P_n\})$. Therefore, we generate the constraint $\mathit{hasFld}(S\langle P_1, \ldots, P_n\rangle, \mathtt{x} : \tau)$. Then, for *any* field access E'.x where the type of E' is $S\langle S_1, \ldots, S_n\rangle$ for any S_1 to S_n, the type of the field access is $\tau[P_1 \mapsto S_1, \ldots, P_n \mapsto S_n]$ (read τ substituting P_1 with S_1, and so on).

2. The type of E is unknown and represented by an inference variable. In this case, we are unaware of what it can be at all, and thus we have to assign it with a δ-type. Hence, if we let the type of E be T, we generate a fresh δ-type δ, the constraint $\mathit{hasFld}(T, \mathtt{x} : \delta)$, and conclude that the type of E.x is δ.

Both of these cases are summarized in Table 1.

Example 3.2. Observe in Figure 2a that the field accesses a1.x, a1.x.y and a2.x cannot be resolved, and the generated declarations for classes A, C and D are in Figure 2b. To resolve a1.x, since the type of a1 is known (A<C>), and its class declaration is in $\mathcal{Q}$, the first case described in Table 1 applies: we generate a fresh $\tau_1 \in \mathcal{F}(\{\mathtt{P_1}\})$ and the constraint $\mathit{hasFld}(\mathtt{A}\langle\mathtt{P_1}\rangle, \mathtt{x} : \tau_1)$. This results in us being able to resolve both a1.x and a2.x, which have types $\tau_1[\mathtt{P_1} \mapsto \mathtt{C}]$ and $\tau_1[\mathtt{P_1} \mapsto \mathtt{D}]$ respectively. To resolve a1.x.y, since the type of a1.x is an inference variable, the second case described in Table 1 applies—we generate a fresh δ_1 and the constraint $\mathit{hasFld}(\tau_1[\mathtt{P_1} \mapsto \mathtt{C}], \mathtt{y} : \delta_1)$, resulting in the type of a1.x.y being δ_1. The resulting types and constraints generated from these can be found in Figure 2c.

Method Invocations. Unlike field accesses, methods must be treated differently because they can be overloaded, overridden, and be polymorphic. For example, the call to m in line 4 of $\mathcal{P}_3$ in Figure 3 is ambiguous, because if we assume that the type of b.a is compatible in an invocation context with String, then the method declaration m defined in line 6 will be invoked, and the method invocation evaluates to B. However, b.a could be of some other type which is compatible with the method parameter of some other overloaded method m defined in class C, which A inherits. Due to these additional considerations, instead of possibly inferring the types of missing method declarations like we did for missing attributes, we will only create method *invocations*, then assign corresponding method *declarations* that support these invocations later.

As such, given a method invocation that is ambiguous or cannot be resolved, we assert that the supplied arguments are compatible with the parameters of one of the already declared methods, or that there is an invocation on a missing/unknown [super]type that is invoked. To do so, suppose the method invocation expression in question has the form $E\texttt{.m(}E_1, \ldots, E_n\texttt{)}$, E has type S, E_1 to E_n have types S_1 to S_n respectively, and the expression $E\texttt{.m(}E_1, \ldots, E_n\texttt{)}$ is found in a context containing declarations of bound type parameters P_1 to P_m. Then, if S has a declaration in $\mathcal{P}$:

- Let $\mathbb{M}$ be the set of all methods declared in $\mathcal{P}$ that S has access to, that has identifier `m` and arity n.
- Let $\mathbb{N}$ be the set of all superclasses of S that are not declared in $\mathcal{P}$ (if S and all its superclasses are declared in $\mathcal{P}$, then $\mathbb{N} = \emptyset$).

Otherwise, let $\mathbb{M} = \emptyset$ and $\mathbb{N} = \{S\}$.

```
1 class A extends C {
2   void main() {
3     B b = new B();
4     System.out.println(m(b.a));
5   }
6   B m(String s) { return new B();
      }
7 }
```

Figure 3: Incomplete program $\mathcal{P}_3$ with ambiguous method declaration called.

As a result, the type of the method invocation is a fresh $\tau_r \in \mathcal{F}(\{P_1, \ldots, P_n\})$ and is constrained by

$$\bigvee_{M \in \mathbb{M}} \left(\left(\bigwedge_{i=1}^{n} S_i <: \mathit{param}(M, i) \right) \wedge \tau_r \equiv \mathit{ret}(M) \right) \vee \bigvee_{T \in \mathbb{N}} \mathit{hasInv}(T, \tau_r\ \texttt{m}(S_1, \ldots, S_n), \{P_1, \ldots, P_m\})$$

$$\text{where } \mathit{param}(M, i) = \text{type of } i^{\text{th}} \text{ parameter of } M$$
$$\mathit{ret}(M) = \text{return type of } M$$

Example 3.3. Continuing from Example 3.2, in line 6 of incomplete program $\mathcal{P}_2$ in Figure 2a, the method invocation `a1.m(a2.x)` cannot be determined. Since `a1` has type `A<C>`, `a2.x` has type $\tau_1[\texttt{P}_1 \mapsto \texttt{C}]$, the expression `a1.m(a2.x)` is in a typing context where `T` and `U` are bound type parameters, $\mathbb{M} = \emptyset$, $\mathbb{N} = \{\texttt{A<C>}\}$, the type of `a1.m(a2.x)` is determined to be a fresh $\tau_2 \in \mathcal{F}(\{\texttt{T}, \texttt{U}\})$ and is constrained by $\mathit{hasInv}(\texttt{A<C>}, \tau_2\ \texttt{m}(\tau_1[\texttt{P}_1 \mapsto \texttt{C}]), \{\texttt{T}, \texttt{U}\})$. Both of these constraints are added to C_2 in Figure 2c.

Example 3.4. In line 4 of incomplete program $\mathcal{P}_3$ in Figure 3, the type of the method invocation `m(b.a)` cannot be determined. Suppose `b.a` has type τ_1. Since the type of the primary is `A`, we let $\mathbb{M} = \{\texttt{B m(String)}\}$, $\mathbb{N} = \{\texttt{C}\}$. As a result, the type of `m(b.a)` is a fresh $\tau_2 \in \mathcal{F}(\emptyset)$ and is constrained by $((\tau_1 <: \texttt{String}) \wedge (\tau_2 \equiv \texttt{B})) \vee \mathit{hasInv}(\texttt{C}, \tau_2\ \texttt{m}(\tau_1), \emptyset)$.

3.1.3 *Constraints On Relations Between Types.* Now that the types of all AST nodes can be determined, our algorithm can populate C with the constraints based on the typed AST of $\mathcal{P}$. At this stage, only two remaining types of AST nodes need to be considered:

1. For an assignment statement E_1 `=` E_2, if the type of E_1 and E_2 are S and T respectively, then we generate the constraint $S <: T$.
2. For a return statement `return` E, if the type of E is S and the statement is found in a method declaration that returns T, then we generate the constraint $S <: T$.

Other constraints also need to be generated for other kinds of AST nodes—like binary operations—for the completion of Java programs in general. These have been described by earlier works [Dagenais,

Algorithm 2 Constraint Solving

Input: A set of configurations, each in the form $\langle \mathcal{P}, \mathcal{Q}, C \rangle$ where $\mathcal{P}$ is the incomplete program, $\mathcal{Q}$ is the generated dependencies for $\mathcal{Q}$, and C are the constraints that must be satisfied for $\mathcal{P}$ and $\mathcal{Q}$ to form a complete and well-typed program
Output: Either `fail` or a configuration $\langle \mathcal{P}, \mathcal{Q}^*, \top \rangle$ where $\mathcal{P}$ and $\mathcal{Q}^*$ form a complete and well-typed program

```
1:  function >>=(x, f)                                          ▷ left-associative, infix
2:      return match x with
3:          case Right[y]: f(y)
4:          case Left[z]: Left[z]

5:  function ConstSolve(configurations)
6:      if configurations = ∅ then return fail
7:      (c, remaining) ← (a, configurations\{a}) for some a ∈ configurations
8:      res ← Reduce(c)                                         ▷ Described in Section 3.2.1
9:              >>= Resolve                                     ▷ Described in Section 3.2.2
10:             >>= Search                                      ▷ Described in Section 3.2.3
11:             >>= DeclareMethods                              ▷ Described in Section 3.2.4
12:     return match res with
13:         case Right[x]: x                                    ▷ x = ⟨P, Q*, ⊤⟩
14:         case Left[y]: ConstSolve(y ∪ remaining)
```

Hendrenis, 2008] and the Java Language Specification [Gosling et al., 2005]. We omit descriptions of these constraints since they are not key to understanding the core contributions of our algorithm. However, they are all generated by our implementation of the algorithm which we describe in Section 4.

Example 3.5. Continuing from Example 3.3, in incomplete program $\mathcal{P}_2$ found in Figure 2a, omitting lines 3 and 4, we have two assignment statements `o = a1.x.y` and `p = a1.m(a2.x)` in lines 5 and 6 respectively. For the former, since the type of `o` is `Object` and the type of `a1.x.y` is δ_1, the constraint we generate for this statement is δ_1 <: `Object`. For the latter, following the same rule, the constraint we generate is τ_2 <: `String`.

By this point, the starting configuration $\langle \mathcal{P}, \mathcal{Q}, C \rangle$ would have been populated successfully. To summarize, $\mathcal{P}$ is the original incomplete program, $\mathcal{Q}$ is built by adding missing class declarations, and C is populated with the constraints generated from resolving unknown field accesses and method invocations, and the constraints on relations between types.

3.2 Constraint Solving

Once the starting configuration $\langle \mathcal{P}, \mathcal{Q}, C \rangle$ has been obtained, we solve and satisfy all the constraints in C, so that $\mathcal{P}$ and $\mathcal{Q}$ form a complete and well typed program. The underlying strategy we employ is to make the necessary modifications to $\mathcal{Q}$ until all constraints in C are satisfied.

At a high level, the constraint solving phase is described in Algorithm 2. In line 6, as a base case we return the failure configuration `fail` if there are no configurations to solve, indicating that $\mathcal{P}$ cannot be completed. In line 7, as part of our search, we pull one configuration c out of *configurations*. Then, we pass c through four phases:

1. Reduce: reduce constraints in C so that they are easier to solve.
2. Resolve: amend $\mathcal{Q}$ so that irreducible constraints in C can be further reduced.

3. Search: search for a type to replace inference variables with.

4. DeclareMethods: declare methods based on invocation constraints.

Each of these phases return either a `Right[`x`]` where x is a configuration, or a `Left[`y`]` where y is a set of configurations. `Rights` signify that the phase has been completed and the configuration can move on to the next phase, whereas `Lefts` signify that the phase is not complete, and has produced a set of configurations which need to go through ConstSolve again. These phases are composed by the left-associative infix `>>=` operator, which is the `Right`-biased monadic bind operator, defined in lines 1–4. As a result, the initial configuration c either has completed all four phases successfully and can be returned (line 13), or has not completed one of the phases, and must go through the ConstSolve function again (line 14).

In the remainder of this section, we describe the four phases of constraint solving.

3.2.1 *Reducing Constraints.* Reducing constraints is routine in type checking and type inference in Java [Gosling et al., 2005] and is similar to term reduction in unification [Martelli, Montanari, 1982]. The idea of constraint reduction is that if a constraint **B** implies another constraint **A**, then solving **B** would solve **A** by implication. We have seen this in Section 2.1, where the constraint `D <: C` implies `A<D> <: A<? extends C>`, and because the class `B<T>` extends `A<T>`, `A<D> <: A<? extends C>` implies `B<D> <: A<? extends C>`. Therefore, by solving `D <: C`, we would have also solved `B<D> <: A<? extends C>`. Our goal is to reduce a constraint like `B<D> <: A<? extends C>` into one like `D <: C` that is easier to solve.

We present the rules of constraint reduction in Figure 4, where $\mathbf{A} \Rightarrow \mathbf{B}$ signifies that **A** reduces to **B**. Many of these rules are standard and taken from the Java Language Specification [Gosling et al., 2005]. In the rest of this subsection, we describe the new rules we have added that deal with constraints on types whose declarations are missing.

Subtype Constraints. There are two main rules that are of key concern to us. The first is the DecldType rule, which is how we reduce subtype constraints between instances of different classes. Loosely, if $S <: T$ where $S \neq T$, and if S extends U, since <: is transitive, if $U <: T$ then $S <: T$. However, this rule is specific to the case where S is declared in $\mathcal{P}$ and therefore cannot be amended. The other rule is MsType1, which describes the case of $S <: T$ where S is not declared in $\mathcal{P}$. The rule states that if in *ignoring type arguments* we can show that $S <: T$, then the constraint reduces in the same way the DecldType does. This distinction is important because it is possible to amend a class S declared in $\mathcal{Q}$ to extend any other class.

Field Membership and Method Invocation Constraints. Field membership and method invocation constraints are only reduced if the type of the primary expression of a field access or method invocation was previously an inference variable, then later replaced with a concrete type. There are two cases for when field membership constraints can be reduced: (DecldFld) the new type of the primary expression has already declared said attribute, in which case we constrain the declared type to be equal to the type of the field access as inferred, and (MsFld) the new type of the primary does not declare the attribute, in which case we reduce the constraint to one that asserts that the class declaration of the new type of the primary does, similar to what we have done in Section 3.1.2. *hasInv* constraints can be reduced when the new type of the primary expression is declared in $\mathcal{P}$, in which case as per the DecldInv rule, it reduces to the same constraint as the one generated in Section 3.1.2.

$$\frac{}{(\texttt{Object} <: S) \Rightarrow (S \equiv \texttt{Object})}\ \textsc{Top} \qquad \frac{}{(S\texttt{<}A_{11}, \ldots, A_{1n}\texttt{>} <: S\texttt{<}A_{21}, \ldots, A_{2n}\texttt{>}) \Rightarrow (\bigwedge_{i\in[1,n]}(A_{1i} \leq A_{2i}))}\ \textsc{SubArgs}$$

$$\frac{}{(S \leq T) \Rightarrow (\lfloor T \rfloor <: \lfloor S \rfloor \wedge \lceil S \rceil <: \lceil T \rceil)}\ \textsc{Containment} \qquad \frac{}{(\bot <: S) \Rightarrow \text{True}}\ \textsc{Bottom} \qquad \frac{S \neq \bot}{(S <: \bot) \Rightarrow \text{False}}\ \textsc{Bottom-Fail}$$

$$\text{where } \lceil S \rceil = \begin{cases} T & \text{if } S = \texttt{? extends } T \text{ or } S \text{ is a type } T \\ \texttt{Object} & \text{otherwise} \end{cases} \qquad \lfloor S \rfloor = \begin{cases} T & \text{if } S = \texttt{? super } T \text{ or } S \text{ is a type } T \\ \bot & \text{otherwise} \end{cases}$$

$$\frac{\text{class } S \text{ declared in } \mathcal{P} \quad T \text{ is concrete} \quad S \neq T \quad S\texttt{<}P_1, \ldots, P_n\texttt{>} \text{ extends } U\texttt{<}Q_1, \ldots, Q_q\texttt{>}}{(S\texttt{<}A_1, \ldots, A_n\texttt{>} <: T\texttt{<}B_1, \ldots, B_m\texttt{>}) \Rightarrow (U\texttt{<}Q_1, \ldots, Q_q\texttt{>}[P_1 \mapsto A_1, \ldots, P_n \mapsto A_n] <: T\texttt{<}B_1, \ldots, B_m\texttt{>})}\ \textsc{DecldType}$$

$$\frac{}{(S\texttt{<}A_{11}, \ldots, A_{1n}\texttt{>} \equiv S\texttt{<}A_{21}, \ldots, A_{2n}\texttt{>}) \Rightarrow (\bigwedge_{i\in[1,n]}(A_{1i} \equiv A_{2i}))}\ \textsc{EquivArgs} \qquad \frac{S \text{ and } T \text{ are concrete} \quad S \neq T}{(S\texttt{<}\ldots\texttt{>} \equiv T\texttt{<}\ldots\texttt{>}) \Rightarrow \text{False}}\ \textsc{Equiv-Fail}$$

$$\frac{S \text{ and } T \text{ are concrete} \quad S \neq T \quad S\texttt{<?}, \ldots, \texttt{?>} <: T\texttt{<?}, \ldots, \texttt{?>}}{T\texttt{<}\ldots\texttt{>} <: S\texttt{<}\ldots\texttt{>} \Rightarrow \text{False}}\ \textsc{SubFail}$$

$$\frac{\text{class } S \text{ declared in } \mathcal{Q} \quad T \text{ is concrete} \quad S\texttt{<}P_1, \ldots, P_n\texttt{>} \text{ extends } U\texttt{<}Q_1, \ldots, Q_q\texttt{>} \quad S\texttt{<?}, \ldots, \texttt{?>} <: T\texttt{<?}, \ldots, \texttt{?>}}{(S\texttt{<}A_1, \ldots, A_n\texttt{>} <: T\texttt{<}B_1, \ldots, B_m\texttt{>}) \Rightarrow (U\texttt{<}Q_1, \ldots, Q_q\texttt{>}[P_1 \mapsto A_1, \ldots, P_n \mapsto A_n] <: T\texttt{<}B_1, \ldots, B_m\texttt{>})}\ \textsc{MsType1}$$

$$\frac{\text{class } S\texttt{<}P_1, \ldots, P_n\texttt{>} \text{ declared in } \mathcal{P} \text{ or } \mathcal{Q} \qquad \text{class } S\texttt{<}P_1, \ldots, P_n\texttt{>} \text{ declares field } \texttt{x} : T}{\mathit{hasFld}(S\texttt{<}A_1, \ldots, A_n\texttt{>}, \texttt{x} : U) \Rightarrow (T[P_1 \mapsto A_1, \ldots, P_n \mapsto A_n] \equiv U)}\ \textsc{DecldFld}$$

$$\frac{\text{class } S\texttt{<}P_1, \ldots, P_n\texttt{>} \text{ declared in } \mathcal{Q} \qquad \text{class } S\texttt{<}P_1, \ldots, P_n\texttt{>} \text{ does not declare field } \texttt{x}}{\mathit{hasFld}(S\texttt{<}A_1, \ldots, A_n\texttt{>}, \texttt{x} : T) \Rightarrow (\mathit{hasFld}(S\texttt{<}P_1, \ldots, P_n\texttt{>}, \texttt{x} : \tau) \wedge \tau[P_1 \mapsto A_1, \ldots, P_n \mapsto A_n] \equiv T) \text{ where } \tau \in \mathcal{F}(\{P_1, \ldots, P_n\}), \tau \text{ fresh}}\ \textsc{MsFld}$$

$$\frac{\text{class } S \text{ declared in } \mathcal{P}}{\mathit{hasInv}(S, R\ \texttt{m}(S_1, \ldots, S_n), \mathbb{S}) \Rightarrow \mathbf{A}}\ \textsc{DecldInv}$$

$$\text{where } \mathbf{A} = \bigvee_{M\in\mathbb{M}} \left(\left(\bigwedge_{i=1}^{n} S_i <: \mathit{param}(M, i) \right) \wedge R \equiv \mathit{ret}(M) \right) \vee \bigvee_{T\in\mathbb{N}} (\mathit{hasInv}(T, R\ \texttt{m}(S_1, \ldots, S_n), \mathbb{S}))$$

Figure 4: Rules for constraint reduction.

The Implementation of Reduce. Given the rules of constraint reduction, we now define what Reduce returns in Figure 5. Essentially, Reduce reduces the constraints in C, as described by the rules ConRed, ConT and ConFail. As we have seen from earlier, disjunctions may be present in C, in which case, as per the DisjSel rule, Reduce returns a set of configurations where each configuration replaces the disjunctions with one of the disjuncts. If none of the rules apply, the configuration has no more reducible constraints, so $\textsc{Reduce}(\langle \mathcal{P}, \mathcal{Q}, C \rangle) = \texttt{Right[}\langle \mathcal{P}, \mathcal{Q}, C \rangle\texttt{]}$ and $\langle \mathcal{P}, \mathcal{Q}, C \rangle$ can proceed to the next stage.

Example 3.6. We show how the constraint `B<D> <: A<? extends C>` obtained from program $\mathcal{P}_1$ in Figure 1a reduces to `D <: C` as we have done in Section 2.1. Firstly, we see that the DecldType rule applies, reducing `B<D> <: A<? extends C>` to `A<D> <: A<? extends C>`.

$$\frac{\text{class B declared in } \mathcal{P}_1 \quad \texttt{A} \text{ is concrete} \quad \texttt{B} \neq \texttt{A} \quad \texttt{B<T>} \text{ extends } \texttt{A<T>}}{(\texttt{B<D> <: A<? extends C>}) \Rightarrow (\texttt{A<D> <: A<? extends C>})}\ \textsc{DecldType}$$

Next, based on the SubArgs rule, the constraint `A<D> <: A<? extends C>` reduces to a containment constraint on the arguments since both types are instances of the same class.

$$\frac{\mathbf{A} \Rightarrow \mathbf{B}}{\textsc{Reduce}(\langle \mathcal{P}, \mathcal{Q}, \mathbf{A} \wedge C \rangle) = \texttt{Left[}\ \{\langle \mathcal{P}, \mathcal{Q}, \mathbf{B} \wedge C \rangle\}\ \texttt{]}}\ \textsc{ConRed} \qquad \frac{\mathbf{A} \Rightarrow \text{True}}{\textsc{Reduce}(\langle \mathcal{P}, \mathcal{Q}, \mathbf{A} \wedge C \rangle) = \texttt{Left[}\{\langle \mathcal{P}, \mathcal{Q}, C \rangle\}\texttt{]}}\ \textsc{ConT}$$

$$\frac{\mathbf{A} \Rightarrow \text{False}}{\textsc{Reduce}(\langle \mathcal{P}, \mathcal{Q}, \mathbf{A} \wedge C \rangle) = \texttt{Left[}\emptyset\texttt{]}}\ \textsc{ConFail}$$

$$\frac{\mathbf{A} = \mathbf{A}_1 \vee \mathbf{A}_2 \vee \cdots \vee \mathbf{A}_n}{\textsc{Reduce}(\langle \mathcal{P}, \mathcal{Q}, \mathbf{A} \wedge C \rangle) = \texttt{Left[}\{\langle \mathcal{P}, \mathcal{Q}, \mathbf{A}_i \wedge C \rangle \mid 1 \leq i \leq n\}\texttt{]}}\ \textsc{DisjSel}$$

Figure 5: Returned result of Reduce based on result of constraint reduction.

$$\frac{}{(\texttt{A<D>} <: \texttt{A<? extends C>}) \Rightarrow (\texttt{D} \leq \texttt{?}\ \texttt{extends}\ \texttt{C})}\ \textsc{SubArgs}$$

Lastly, the containment constraint reduces to subtype constraints as per the Containment rule.

$$\frac{}{(\texttt{D} \leq \texttt{?}\ \texttt{extends}\ \texttt{C}) \Rightarrow ((\bot <: \texttt{D}) \wedge (\texttt{D} <: \texttt{C}))}\ \textsc{Containment}$$

The Bottom rule states that $\bot$ <: D reduces to True. Thus, what remains after constraint reduction is the constraint D <: C.

3.2.2 *Resolving Constraints.* At this stage, the configuration does not have any reducible constraints. Constraints are irreducible if there is not enough information in $\mathcal{P}$ to show that they are satisfiable. For example, what remains of Example 3.6 is the constraint D <: C, which cannot be reduced further because none of the rules in Figure 5 apply. Essentially, we can neither conclude that D *is already a subtype of* C nor that D *can never be a subtype of* C. The goal of this phase is to amend $\mathcal{Q}$ so that some of the remaining constraints can be reduced further.

The implementation of Resolve is shown in Figure 6. At this stage, there are two amendments to $\mathcal{Q}$ that we make to reduce constraints further. The first, as described by the MsType2 rule, states roughly that if S <: T but there is currently no way to reduce it further, then we force S to extend T. However, since a class can only extend one other class in Java, it must mean that if S already extends a class U, then we either assert that (1) U <: T, or (2) S now extends T and T <: U. The second case is described by the DeclFld rule, which states that if a class S must have a field x of type T, then we just add such a declaration to the class declaration of S. By making these changes to $\mathcal{Q}$, previously irreducible constraints can now be reduced. Finally, just like before, if none of the rules apply, it means that none of the remaining constraints can be resolved, so $\textsc{Resolve}(\langle \mathcal{P}, \mathcal{Q}, C \rangle) = \texttt{Right[}\langle \mathcal{P}, \mathcal{Q}, C \rangle\texttt{]}$ and $\langle \mathcal{P}, \mathcal{Q}, C \rangle$ can proceed to the next stage.

Example 3.7. From Example 3.6, we were left the constraint D <: C which was irreducible. In this scenario, the MsType2 rule applies. As D currently extends Object (by default, all classes in Java extend Object), Resolve branches the current configuration into two configurations—one where the constraint D <: C is substituted with Object <: C, and the other where the class D now extends C and the constraint C <: Object is added.

$$\frac{\text{class D declared in } \mathcal{Q}_1 \qquad \texttt{D} \text{ extends } \texttt{Object} \qquad \texttt{C} \text{ is concrete}}{\textsc{Resolve}(\langle \mathcal{P}_1, \mathcal{Q}_1, \mathbf{A} \wedge C_1 \rangle) = \texttt{Left[}\{\langle \mathcal{P}_1, \mathcal{Q}_1, C_1' \rangle, \langle \mathcal{P}_1, \mathcal{Q}_1'', C_1'' \rangle\}\texttt{]}}\ \textsc{MsType2}$$

$$\begin{aligned} \text{where } \mathbf{A} &= \texttt{D} <: \texttt{C} \\ C_1' &= \texttt{Object} <: \texttt{C} \wedge C \\ \mathcal{Q}_1'' &= \mathcal{Q}_1 \text{ except that class } \texttt{D} \text{ extends } \texttt{C} \\ C_1'' &= (\texttt{C} <: \texttt{Object}) \wedge \mathbf{A} \wedge C_1 \end{aligned}$$

Both configurations $\langle \mathcal{P}_1, \mathcal{Q}_1, C_1' \rangle$ and $\langle \mathcal{P}_1, \mathcal{Q}_1'', C_1'' \rangle$ now have reducible constraints. The first configuration $\langle \mathcal{P}_1, \mathcal{Q}_1, C_1' \rangle$ will be rejected because Object <: C reduces to False, while the constraints C <: Object and D <: C in the configuration $\langle \mathcal{P}_1, \mathcal{Q}_1'', C_1'' \rangle$ both reduce to True.

$$\frac{\text{class } S\langle P_1,\ldots,P_n\rangle \text{ declared in } \mathcal{Q} \qquad S\langle P_1,\ldots,P_n\rangle \text{ extends } U\langle Q_1,\ldots,Q_q\rangle \qquad T \text{ is concrete}}{\textsc{Resolve}(\langle \mathcal{P},\mathcal{Q},\mathbf{A}\wedge C\rangle) = \texttt{Left[}\{\langle \mathcal{P},\mathcal{Q},C_1\rangle,\langle \mathcal{P},\mathcal{Q}_2,C_2\rangle\}\texttt{]}}\ \textsc{MsType2}$$

where $\mathbf{A} = S\langle A_1,\ldots,A_n\rangle <: T\langle B_1,\ldots,B_m\rangle$
$C_1 = (U\langle Q_1,\ldots,Q_q\rangle[P_1 \mapsto A_1,\ldots,P_n \mapsto A_n] <: T\langle B_1,\ldots,B_m\rangle) \wedge C$
$\mathcal{Q}_2 = \mathcal{Q}$ except that class $S\langle P_1,\ldots,P_n\rangle$ extends $T\langle \tau_1,\ldots,\tau_m\rangle$
$C_2 = (T\langle \tau_1,\ldots,\tau_m\rangle <: U\langle Q_1,\ldots,Q_q\rangle) \wedge \mathbf{A} \wedge C$
$\forall i.\ \tau_i \in \mathcal{F}(\{P_1,\ldots,P_n\}), \tau_i$ fresh

$$\frac{\text{class } S\langle P_1,\ldots,P_n\rangle \text{ declared in } \mathcal{Q} \qquad \text{class } S\langle P_1,\ldots,P_n\rangle \text{ does not declare field } \texttt{x}}{\textsc{Resolve}(\langle \mathcal{P},\mathcal{Q},\mathbf{A}\wedge C\rangle) = \texttt{Left[}\{\langle \mathcal{P},\mathcal{Q}',C\rangle\}\texttt{]}}\ \textsc{DeclFld}$$

where $\mathbf{A} = \mathit{hasFld}(S\langle P_1,\ldots,P_n\rangle, \texttt{x} : T)$
$\mathcal{Q}'$ = $\mathcal{Q}$ where field declaration $\texttt{x} : T$ is added to class $S\langle P_1,\ldots,P_n\rangle$

Figure 6: Returned result of Resolve based on constraints in C.

$$\frac{S \text{ is a } \tau\text{-type that occurs in } \langle \mathcal{P},\mathcal{Q},C\rangle \qquad S \in \mathbb{S}}{\textsc{Search}(\langle \mathcal{P},\mathcal{Q},C\rangle) = \texttt{Left[}\{\langle \mathcal{P},\mathcal{Q},C\rangle[S \mapsto T] \mid T \in \mathbb{S}\}\texttt{]}}\ \tau\textsc{-Repl}$$

$$\frac{\alpha\langle S_1,\ldots,S_n\rangle \text{ occurs in } \langle \mathcal{P},\mathcal{Q},C\rangle}{\textsc{Search}(\langle \mathcal{P},\mathcal{Q},C\rangle) = \texttt{Left[}\{\langle \mathcal{P},\mathcal{Q},C\rangle[\alpha \mapsto T] \mid T \in \mathbb{S}\} \cup \{\langle \mathcal{P},\mathcal{Q}',C\rangle[\alpha \mapsto \texttt{UNKNOWN}_j]\}\texttt{]}}\ \alpha\textsc{-Repl}$$

where $\mathbb{S}$ = set of all classes in $\mathcal{P}$ and $\mathcal{Q}$ with arity n
$\mathcal{Q}' = \mathcal{Q}$ ++ `class` $\texttt{UNKNOWN}_j$ `{ }`, j fresh

$$\frac{S \text{ is a } \delta\text{-type}}{\textsc{Search}(\langle \mathcal{P},\mathcal{Q},(S \equiv T)\wedge C\rangle) = \texttt{Left[}\{\langle \mathcal{P},\mathcal{Q},C\rangle[S \mapsto T]\}\texttt{]}}\ \delta\textsc{-Repl}$$

Figure 7: Searching for replacements of inference variables.

3.2.3 *Searching for Types.* At this stage, constraints on concrete types have mostly been reduced and resolved (except for *hasInv* constraints which we deal with later). Now, the configuration is left with constraints on inference variables and we must begin to search for their assignments. Fortunately, τ-types and α-types already have a known search domain—τ-types have an explicitly stated search domain, and α-types are just regular class types occurring in the program. Therefore, Search replaces these types with one of the types in their respective domains in the configuration so that further constraint reduction and resolution can be made.

The implementation of Search is shown in Figure 7. The τ-Repl rule states that Search replaces a τ-type with one of the types in its domain. The α-Repl rule states that α is replaced with one of the existing class types in $\mathcal{P}$ and $\mathcal{Q}$ with the same arity as α, or a newly declared class type. As per usual, if none of the rules apply, then $\textsc{Search}(\langle \mathcal{P},\mathcal{Q},C\rangle) = \texttt{Right[}\langle \mathcal{P},\mathcal{Q},C\rangle\texttt{]}$.

Iteratively Searching for Polymorphic Types. The function $\mathcal{F}$ defined in Section 3.1.2 was constructed so that we are able to perform an iterative search over what a polymorphic inference variable could be. For example, suppose in some program τ_1 was replaced with $\alpha_1\langle\tau_{11}\rangle$ by τ-Repl, and then replaced with $\texttt{A}\langle\tau_{11}\rangle$ by α-Repl—this means that we have decided that τ_1 is some instance of A. This enables further constraint reduction and resolution, and eventually we arrive at having to search for a replacement for τ_{11}, and so on. This gives us a step-by-step framework for dealing with unknown types in an environment that permits polymorphic types.

Promotion of δ-types. The δ-Repl rule in Figure 7 states that if a δ-type is constrained to be equivalent with another type T, then it is replaced with T. Most notably, we do not perform a search over possible assignments of δ-types. The reason for this is that δ-types are *promoted* to τ-types as the search progresses. Recall that δ-types are generated when the type of the target of a field access is an inference variable, for example, the type of `a1.x.y` in line 5 of Figure 2a was decided to be δ_1, as stated in Figure 2c. Suppose in one path of our search, τ_1 was replaced with $\texttt{P}_1$, in which case, the constraint $\mathit{hasFld}(\tau_1[\texttt{P}_1 \mapsto \texttt{C}], \texttt{y} : \delta_1)$ is now replaced with $\mathit{hasFld}(\texttt{C}, \texttt{y} : \delta_1)$. As per MsFld in Figure 5, this constraint later reduces to $\mathit{hasFld}(\texttt{C}, \texttt{y} : \tau_3) \wedge \delta_1 \equiv \tau_3$ where $\tau_3 \in \mathcal{F}(\emptyset)$, and finally, δ-Repl replaces δ_1 with τ_3. Essentially, as the search progresses, δ-types are promoted to τ-types, revealing their search domains.

A Principled Search. The rules shown in Figure 7 perform a naive search over all the possible types that a τ-type can inhabit. Typically, a τ-type is either a type parameter or some class type represented by α-types. Recall that the set of α-types in the search domain of τ-types is infinitely large, for example, if $\tau \in \mathcal{F}(\emptyset)$ then $\tau \in \{\alpha, \alpha\texttt{<}\tau_1\texttt{>}, \alpha\texttt{<}\tau_1, \tau_2\texttt{>}, \dots\}$. Having to search through all of these types is a waste if we can determine that τ must actually be, for example, a `String`. However, being able to make this determination is nontrivial and not always possible, because even if we had decided that some τ is to be a class type, its arity and type arguments are still completely unknown. Thus in this stage, we exploit rules of well-formed type hierarchies, which only requires analysis of the *erasures* of class types (the class types without their arguments), to replace the infinitely large search domain of a τ-type with one that is finite.

There are three scenarios where we can decide immediately which class type(s) that a τ-type must be an instance of, without knowledge of its arity or type arguments:

1. Cyclic subtyping. Cyclic inheritance is prohibited in a well-formed Java program. Therefore, if a τ-type S a subtype of a concrete type T and T is likewise a subtype of S, then we cannot replace S with any other class type other than T.

2. Final superclasses. A class that is declared as `final` cannot be extended by any other class. Therefore, if a τ-type S is a subtype of a concrete type T where T is declared as `final`, then we cannot replace S with any class type other than T.

3. Fully-declared subtypes. A fully-declared class is a class whose declaration is in $\mathcal{P}$, and extends a fully-declared class. Because we do not mutate $\mathcal{P}$, if a τ-type S is found to be a supertype of a fully-declared class T, it must be the case that S cannot be a class type other than one of the supertypes of T.

From the above, we are able to prune the search space of some τ-types occurring in $\langle \mathcal{P}, \mathcal{Q}, C \rangle$. The rules for doing so are presented in Figure 8. The EExt, ESub, ESup and ETrns rules describe a transitive subtype relation where we ignore all type arguments and any substitutions to τ-types (note that $\tau[\dots]$ means that any substitutions on τ are unimportant for our analysis). Then, the following three rules describe the new search domain of a τ-type if any of the rules apply; the PrCycl rule prunes the search domain in the case of cyclic inheritance, the PrFinal rule prunes the search domain in the case of final superclasses, and the PrFD rule prunes the search domain in the case of fully-declared subtypes ($|T|$ is the arity of T as declared in $\mathcal{P}$). These rules let us prune the search domain of τ-types before applying the τ-Repl rule in Figure 7.

Example 3.8. Observe that one of the constraints generated from Example 3.5 as shown in Figure 2c is τ_2 <: `String` where $\tau_2 \in \mathcal{F}(\{\texttt{T}, \texttt{U}\})$. By ESup, $\tau_2 \hookrightarrow$ `String`. Since `String` is declared as `final`, by PrFinal, the new search domain of τ_2 is $\{\texttt{T}, \texttt{U}, \texttt{String}\}$.

$$\frac{\text{class } S\text{<}S_1,\ldots,S_m\text{>} \text{ extends class } T\text{<}T_1,\ldots,T_n\text{>} \text{ in } \mathcal{P} \text{ or } \mathcal{Q}}{S \hookrightarrow T}\ \textsc{EExt}$$

$$\frac{C = S\text{<}A_1\ldots,A_s\text{>} <: \tau[\ldots] \wedge \ldots}{S \hookrightarrow \tau}\ \textsc{ESub} \qquad \frac{C = \tau[\ldots] <: S\text{<}A_1\ldots,A_s\text{>} \wedge \ldots}{\tau \hookrightarrow S}\ \textsc{ESup} \qquad \frac{S \hookrightarrow T \qquad T \hookrightarrow U}{S \hookrightarrow U}\ \textsc{ETrns}$$

$$\frac{\text{class } S\text{<}Q_1,\ldots,Q_m\text{>} \text{ declared in } \mathcal{P} \text{ or } \mathcal{Q} \qquad S \hookrightarrow \tau \qquad \tau \hookrightarrow S \qquad \tau \in \{P_1,\ldots,P_n,\alpha,\alpha\text{<}\tau_1\text{>},\ldots\}}{\tau \in \{P_1,\ldots,P_n,S\text{<}\tau_1,\ldots,\tau_m\text{>}\}}\ \textsc{PrCycl}$$

$$\frac{\text{class } S\text{<}Q_1,\ldots,Q_m\text{>} \text{ declared in } \mathcal{P} \text{ as final} \qquad \tau \hookrightarrow S \qquad \tau \in \{P_1,\ldots,P_n,\alpha,\alpha\text{<}\tau_1\text{>},\ldots\}}{\tau \in \{P_1,\ldots,P_n,S\text{<}\tau_1,\ldots,\tau_m\text{>}\}}\ \textsc{PrFinal}$$

$$\frac{\text{class } S\text{<}Q_1,\ldots,Q_m\text{>} \text{ fully declared in } \mathcal{P} \qquad S \hookrightarrow \tau \qquad \tau \in \{P_1,\ldots,P_n,\alpha,\alpha\text{<}\tau_1\text{>},\ldots\}}{\tau \in \{P_1,\ldots,P_n,S\text{<}\tau_1,\ldots,\tau_m\text{>}\} \cup \{T\text{<}\tau_1,\ldots,\tau_{|T|}\text{>} \mid S \hookrightarrow T, T \text{ is concrete}\}}\ \textsc{PrFD}$$

Figure 8: Pruning the search domain of τ-types.

3.2.4 *Declaring Methods.* Finally, we get to the stage of declaring methods. At this stage, all inference variables have been replaced with concrete types and the only remaining constraints are *hasInv* constraints. The goals of this stage are (1) to resolve the *hasInv* constraints by either matching it with a declaration that the type of the primary expression has access to, or to create a brand new method declaration that supports this invocation, and (2) to ensure that all new method declarations that are override-equivalent to a supertype's method declaration actually overrides it. These goals are achieved by the DeclMtd and Overrides rules presented in Figure 9 respectively. Because the methods that a class inherits depends on the declaration of its superclasses, these rules are applied in the order based on a topological sort over the type hierarchy of the classes in the program.

If none of the rules apply, it means that $C = \top$ and $\mathcal{P}$ and $\mathcal{Q}$ form a complete and well-typed program, and our objective has been achieved.

Example 3.9. We continue with Example 3.5 by performing constraint solving on the configuration $\langle \mathcal{P}_2, \mathcal{Q}_2, C_2 \rangle$ obtained from $\mathcal{P}_2$ as shown in Figure 2. Each step of ConstSolve is shown in Figure 10, and the resulting generated dependencies $\mathcal{Q}_2^*$ are shown in Figure 10a. The resulting $\mathcal{Q}_2^*$, together with $\mathcal{P}_2$, form a complete and well-typed program.

4 IMPLEMENTATION OF JAVACIP

To evaluate the efficacy of our algorithm, we implemented it as a program called the Java Compiler for Incomplete Programs (JavaCIP), which is available publicly on (repository link redacted for double-blind reviewing). JavaCIP is developed in Scala [Odersky et al., 2006] and makes use of the JavaParser [jav, ????] parser to build the AST of incomplete programs. As input, JavaCIP receives an incomplete Java program, and as output, produces Java source code containing the class declarations that complete it. The algorithm implemented by JavaCIP is described in Algorithm 3. Although our description of the algorithm in Section 3 deals only with classes, fields and methods, JavaCIP is able to deal with programs containing other commonly-used Java features and program constructs such as interfaces, static members, constructors, primitive types, arrays, loops, conditional statements, and so on. This only required small extensions to the techniques that we have presented. However, JavaCIP does not support programs containing the Java features listed in Section 4.2.

4.1 Additional Features

We also introduced several additional features to JavaCIP that were not discussed earlier.

$$\frac{\text{class } S\texttt{<}P_1, \ldots, P_s\texttt{>} \text{ declared in } \mathcal{Q}}{\textsc{DeclareMethods}(\langle \mathcal{P}, \mathcal{Q}, \mathbf{A} \wedge C \rangle) = \texttt{Left[}\{\langle \mathcal{P}, \mathcal{Q}, \mathbf{B} \wedge C \rangle\} \cup \{\langle \mathcal{P}, \mathcal{Q}_i, \mathbf{C}_i \wedge C \rangle \mid i \geq 0\}\texttt{]}} \textsc{DeclMtd}$$

$$\text{where } \mathbf{A} = \mathit{hasInv}(S\texttt{<}A_1, \ldots, A_s\texttt{>}, R\ \texttt{m(}S_1, \ldots, S_n\texttt{)}, \mathbb{S})$$

$$\mathbf{B} = \bigvee_{M \in \mathbb{M}} \left(\left(\bigwedge_{i=1}^{n} S_i <: \mathit{param}(M, i) \right) \wedge R \equiv \mathit{ret}(M) \right)$$

$$\forall i. \mathcal{Q}_i = \mathcal{Q} \text{ where } S\texttt{<}P_1, \ldots, P_s\texttt{>} \text{ declares method } \texttt{<}T_1, \ldots, T_i\texttt{>}\ \tau_{1r}\ \texttt{m(}\tau_{11}, \ldots, \tau_{1n}\texttt{)}$$

$$\mathbf{C}_i = \tau_{1r} <: R \wedge \bigwedge_{1 \leq j < n} S_j <: \tau_{1j}[P_1 \mapsto A_1, \ldots, P_s \mapsto A_s, T_1 \mapsto \tau_{21}, \ldots, T_i \mapsto \tau_{2i}]$$

$$\forall j.\ \tau_{1j} \in \mathcal{F}(\{P_1, \ldots, P_s, T_1, \ldots, T_i\});\quad \tau_{2j} \in \mathcal{F}(\mathbb{S});\quad \tau_{1j}, \tau_{2j} \text{ fresh}$$

$$\frac{\begin{array}{c}\text{class } S\texttt{<}P_1, \ldots, P_s\texttt{>} \text{ declares method } \texttt{<}T_1, \ldots, T_t\texttt{>}\ R_1\ \texttt{m(}A_1, \ldots, A_n\texttt{)} \\ \text{class } S\texttt{<}P_1, \ldots, P_s\texttt{>} \text{ inherits method } \texttt{<}T_1, \ldots, T_t\texttt{>}\ R_2\ \texttt{m(}A_1, \ldots, A_n\texttt{)} \\ R_1 \text{ is not a subtype of } R_2\end{array}}{\textsc{DeclareMethods}(\langle \mathcal{P}, \mathcal{Q}, C \rangle) = \texttt{Left[}\{\langle \mathcal{P}, \mathcal{Q}', \mathbf{A} \wedge C \rangle\}\texttt{]}} \textsc{Overrides}$$

$$\text{where } \mathbf{A} = \tau <: R_1 \wedge \tau <: R_2$$

$$\forall i. \mathcal{Q}_i = \mathcal{Q} \text{ where } R_1 \text{ is replaced with } \tau$$

$$\tau \in \mathcal{F}(\{P_1, \ldots, P_s, T_1, \ldots, T_t\})$$

Figure 9: Declaring methods.

Algorithm 3 JavaCIP

Input: An incomplete Java program $\mathcal{P}$
Output: A set of source files that complete $\mathcal{P}$

```
1: R ← ConstSolve({ConstGen(P)})
2: if R = ⟨P, Q*, ⊤⟩ then                    ▷ Ensure constraint generation/solving did not fail
3:     for each Class/Interface declaration S in Q* do
4:         write S as Java file
```

4.1.1 *Unresolvable Primaries for Fields and Methods.* It is possible for some incomplete programs to be impossible to complete because the primary expression of some fields/methods cannot be resolved. An example of this is an incomplete program defining a class `A` that does not extend any other class, and yet refers to an undeclared variable `x`. Normally, we would reject this program since no completions to the incomplete program can be made to resolve the primary of `x`. However, for experimentation purposes we would still like to make some attempt to complete it. Hence, we prepend the primary type `JavaCIPUnknownScope` to these unresolvable fields and methods. Note that this is the only case where the incomplete program is amended.

4.1.2 *Searching Through Configurations.* The ConstSolveImpl algorithm shown in Algorithm 2 chooses any configuration within the provided set of configurations and performs the four stages of constraint solving on it. The default behaviour is to search through the configurations depth-first, i.e., to select the last configuration that was added to this set. This is so that we can explore one path in the search space and quickly reject unsatisfiable configurations. However, in some scenarios the search space expands too deeply, which may not be ideal. Thus, we added an option to search the configurations sorted by maximum arity and nesting depth of types occurring in the program, which may, in some scenarios, allow us to complete the incomplete program more quickly.

Definitions after Constraint Generation (from Figure 2)
Q_2 = `class A<P1> { } class C { } class D { }`
C_2 = $hasFld($`A<P1>`$, x : \tau_1) \wedge hasFld(\tau_1[P_1 \mapsto C], y : \delta_1) \wedge \delta_1 <:$ `Object` $\wedge$
$hasInv($`A<C>`$, \tau_2$ `m(`$\tau_1[P_1 \mapsto D]$`)`$) \wedge \tau_2 <:$ `String`

1. **Resolve** (declare field `x` in class `A`)
Q_2 = `class A<P1> {` $\underline{\tau_1}$ `x;` `} class C { } class D { }`
C_2 = ~~$hasFld($`A<P1>`$, x : \tau_1) \wedge$~~ $hasFld(\tau_1[P_1 \mapsto C], y : \delta_1) \wedge \delta_1 <:$ `Object` $\wedge$
$hasInv($`A<C>`$, \tau_2$ `m(`$\tau_1[P_1 \mapsto D]$`)`$) \wedge \tau_2 <:$ `String`

2. **Search** (replace τ_1 with `P1`)
Q_2 = `class A<P1> {` $\underline{\texttt{P}_1}$ `x; } class C { } class D { }`
C_2 = $hasFld($`C`$, y : \delta_1) \wedge \delta_1 <:$ `Object` $\wedge$ $hasInv($`A<C>`$, \tau_2$ `m(`$\underline{\texttt{D}}$`)`$) \wedge \tau_2 <:$ `String`

3. **Reduce** ($hasFld($`C`$, y : \delta_1)$ reduces to $hasFld($`C`$, y : \tau_3) \wedge \delta_1 \equiv \tau_3$)
C_2 = ~~$hasFld($`C, y :`$ \delta_1)$~~ $\underline{hasFld(\texttt{C}, \texttt{y} : \tau_3) \wedge \delta_1 \equiv \tau_3} \wedge \delta_1 <:$ `Object` $\wedge$
$hasInv($`A<C>`$, \tau_2$ `m(D))` $\wedge \tau_2 <:$ `String` where $\tau_3 \in \mathcal{F}(\emptyset)$

4. **Resolve** (declare field `y` in class `C`)
Q_2 = `... class C {` $\underline{\tau_3}$ `y;` `} ...`
C_2 = ~~$hasFld($`C, y :`$ \tau_3)$ $\wedge$~~ $\delta_1 \equiv \tau_3 \wedge \delta_1 <:$ `Object` $\wedge$
$hasInv($`A<C>`$, \tau_2$ `m(D))` $\wedge \tau_2 <:$ `String`

5. **Search** (replace δ_1 with τ_3, then α_3, then `Object`)
Q_2 = `... class C {` $\underline{\texttt{Object}}$ `y; } ...`
C_2 = ~~$\delta_1 \equiv \tau_3 \wedge$~~ $\underline{\texttt{Object}} <:$ `Object` $\wedge$
$hasInv($`A<C>`$, \tau_2$ `m(D))` $\wedge \tau_2 <:$ `String`

6. **Reduce** (`Object <: Object` reduces to True)
C_2 = ~~`Object <: Object` $\wedge$~~ $hasInv($`A<C>`$, \tau_2$ `m(D))` $\wedge$
$\tau_2 <:$ `String`

7. **Search** (replace τ_2 with `String`)
C_2 = $hasInv($`A<C>`$, \underline{\texttt{String}}$ `m(D))` $\wedge \underline{\texttt{String}} <:$ `String`

8. **Reduce** (`String <: String` reduces to True)
C_2 = $hasInv($`A<C>`$,$ `String m(D))` ~~$\wedge$ `String <: String`~~

9. **DeclareMethods** (declare method `m` in class `A`)
Q_2 = `class A<P1> {` $\underline{\tau_5}$ `m(`τ_4 `a) { ... } } ... } ...`
C_2 = ~~$hasInv($`A<C>, String m(D))`~~
$\underline{\texttt{D} <: \tau_4[P_1 \mapsto C] \wedge \tau_5[P_1 \mapsto C] <:}$ `String`
where $\tau_4, \tau_5 \in \mathcal{F}(\{$`P1`$\})$

10. **Search** (replace τ_4 with α_4 then `D`, and τ_5 with `String`)
Q_2 = `class A<P1> {` $\underline{\texttt{String}}$ `m(`$\underline{\texttt{D}}$ `a) { ... } ... } ...`
C_2 = `D <:` $\underline{\texttt{D}} \wedge \underline{\texttt{String}} <:$ `String`

11. **Reduce** (both constraints reduce to True)
C_2 = ~~`D <: D` $\wedge$ `String <: String`~~ $\top$

```
1  class A<P1> {
2      P1 x;
3      String m(D a) {
4          return null;
5      }
6  }
7
8  class C {
9      Object y;
10 }
11
12 class D { }
```

(a) Q_2^* generated by ConstSolve.

Figure 10: Example of constraint solving.

4.1.3 *Artificially Limiting the Search Space.* The search space of a τ-type can be infinitely large. Also, in the DeclareMethods stage the method we declare can contain an arbitrary number of type parameters. Although we have seen in Section 3.2.3 that we can conclusively limit the search domain of types in some cases, it is likely very difficult—or impossible—to do so in general. Hence, we implemented options for the user to specify the maximum arity and nesting depth of types occurring in the configurations that will be searched so that the search space becomes finite. Although this affects the completeness of the search, our experiments described in Section 5 show that providing a hard limit on these parameters is not a limiting factor on the effectiveness of JavaCIP.

4.2 Unsupported Features

JavaCIP is not able to deal with Java programs using the following Java features either because (1) supporting them requires substantial extensions to the underlying algorithm we presented or (2) supporting them requires additional implementation effort for JavaCIP without providing insight to the effectiveness of our algorithm.

- Exceptions: While at the type level JavaCIP is able to determine that an object that is being thrown/caught must be an instance of `Throwable` or `Exception`, because Java may give

compiler errors for uncaught/unthrown exceptions, JavaCIP assumes that all exceptions thrown are unchecked exceptions, i.e., subtypes of `RuntimeException`s. However, unchecked exceptions are also unsupported because JavaCIP disregards the order in which exceptions are caught, and this affects whether some exceptions can be subtypes of others. Support for exceptions require a substantial extension to the underlying algorithm which we leave for future work.

- Type parameter bounds: JavaCIP does receive programs that have type parameter bounds, but does not generate them in the reconstructed surrounding dependencies because type parameter bounds affect method overriding and overloading. This complicates the DeclareMethods phase of our algorithm, and we leave this for future work.
- Lambda expressions and method references: These are tricky to deal with because a single lambda expression can be of different types, and method references are ambiguous because methods can be overloaded.
- Annotations, static/inner/anonymous classes, enumerations, the `var` annotation: Support for these require additional implementation effort for JavaCIP without providing insight on the efficacy of the underlying algorithm.
- Features introduced after JDK 11: JDK 11 is a Long Term Support (LTS) of Java. At the time of this writing, two more LTS versions of Java have been released since JDK 11 (JDK 17 and 21). In these versions, several features and other forms of syntax sugar have been introduced, and support for these also require additional implementation effort for JavaCIP without providing insight on the efficacy of our underlying algorithm.

5 EXPERIMENTAL EVALUATION

For our experimental evaluation, our goal is to answer the following research questions:

RQ1: How effectively does JavaCIP reconstruct missing dependencies of incomplete programs?

RQ2: What is the runtime efficiency of JavaCIP, and how does program size affect runtime?

RQ3: Does JavaCIP produce wrong completions?

In answering these questions we also benchmark our results against earlier works that meet similar research objectives. The two most relevant tools that complete incomplete Java programs are PPA [Dagenais, Hendrenis, 2008] and JCoffee [Gupta et al., 2020]. Between the two, we chose to compare JavaCIP with JCoffee because (1) PPA has been deprecated, and (2) just like JavaCIP, JCoffee generates compilable Java source code, while PPA generates Java bytecode directly. Note that although JCoffee has similar objectives to JavaCIP, it has trouble handling parametrically polymorphic types and may modify the input programs [Gupta et al., 2020] by inserting typecasts and import statements.

5.1 Experimental Setup

5.1.1 *Datasets.* We provision two datasets for our experiments. The first dataset is derived from the same dataset used in the evaluation of JCoffee, and the second dataset contains real-world code examples. All datasets and experimental results shown are available online at (link redacted for double-blind reviewing).

Table 2: JavaCIP on Spring Boot Web (SBW), BigCloneBench (BCB), BCB Subset containing polymorphic programs (BCBSub) and Custom datasets.

(a) Number of programs that were completed, cannot be completed and caused timeouts.

Result \ Dataset	SBW		BCB		BCBSub		Custom	
Completed	108	(91.5%)	4848	(99.1%)	426	(97.7%)	15	(50.0%)
Cannot Complete	1	(0.8%)	8	(0.2%)	6	(1.4%)	15	(50.0%)
Timeout (1min)	9	(7.6%)	33	(0.7%)	4	(0.9%)	0	(0.0%)

(b) Classification of outcomes.

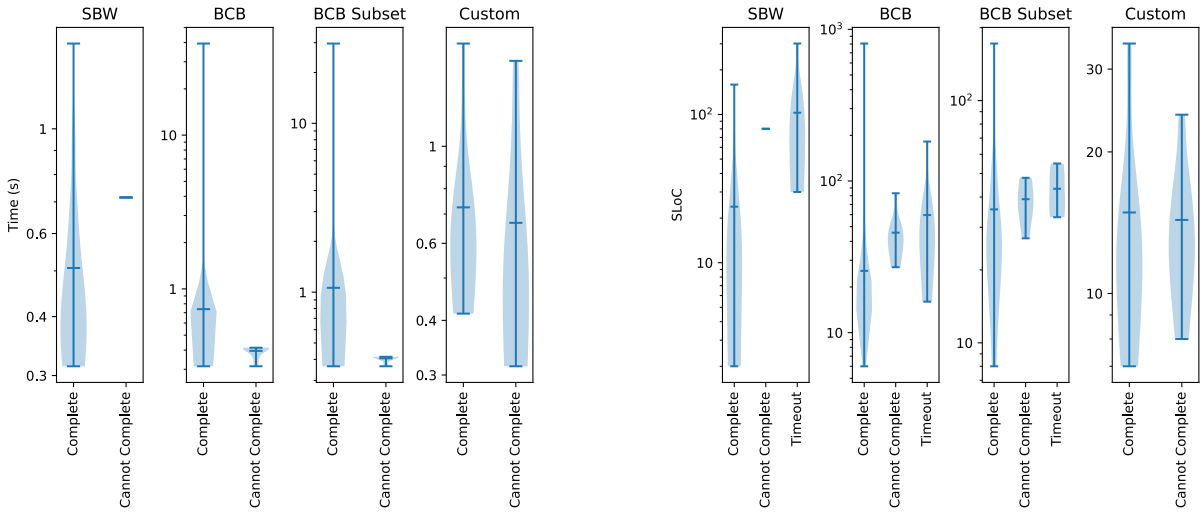
Result \ Dataset	SBW		BCB		Custom		Total	
True Positive	108	(91.5%)	4848	(99.1%)	15	(50.0%)	4971	(98.7%)
False Positive	0	(0%)	0	(0%)	0	(0%)	0	(0%)
True Negative	0	(0%)	8	(0.2%)	15	(50%)	23	(0.4%)
False Negative	10	(8.5%)	33	(0.7%)	0	(0%)	43	(0.9%)

BigCloneBench. The first dataset we use is the same dataset of 9133 Java programs used in the evaluation of JCoffee [Gupta et al., 2020], which is derived from BigCloneBench [Svajlenko et al., 2014]. We removed 4244 programs that contained uses of features unsupported by JavaCIP, leaving us with 4889 programs. Among the 4889 programs, 436 referred to or required the generation of polymorphic types. They have an average SLoC of 25.8, where the smallest and largest programs have 6 SLoC and 806 SLoC respectively.

Spring Boot Web. Because the original BigCloneBench dataset contains artificially created programs, we obtain a second dataset containing source code of real-world applications. We chose to use the source code for the Spring Boot Web framework (`https://spring.io/projects/spring-boot`), which is a widely-used open-source framework for creating web servers in Java. It originally contains 168 Java files, and 43 of these were removed because they contain extensive uses of JavaCIP's unsupported features. Among the remaining 125 files, 54 of these programs contained minimal uses of the unsupported features, for example, a single occurrence of a lambda expression. These uses of unsupported features were stripped from the programs. Of these 125 files, 7 were already complete, and were removed from the dataset. This leaves 118 files in our dataset with an average Source Lines of Code (SLoC) of 30.4, where the smallest and largest files have 2 SLoC and 301 SLoC respectively. Of these 118 programs, 51 of them referred to or relied on the generation of polymorphic types. Unlike the BigCloneBench dataset, this dataset has significantly fewer programs because of the manual preparation required.

5.1.2 *Approach.* For both datasets, each file is treated as a standalone incomplete program and has no access to any other Java files or external dependency JARs. Each file is then passed through JavaCIP for dependency reconstruction, and the generated dependencies, together with the original file, are compiled with the Java compiler (`javac`) [Gosling et al., 2005] for verification of correctness. The experiments are run on an Arch Linux system equipped with an Intel i7-14700 CPU. The time limit for completing each program is 1 minute.

5.1.3 *Experimental Results.* The results of our experiments are shown in Table 2 and Figures 11 and 12. Table 2a shows the number of programs that JavaCIP completed, cannot complete and timed-out for each dataset. Table 2b classifies the outcomes from 2a. Violin plots for time taken for JavaCIP to



(a) Time taken for JavaCIP to terminate.

(b) SLoC of programs.

Figure 11: Time taken for JavaCIP to terminate and SLoC of programs in Spring Boot Web (SBW), BigCloneBench (BCB), BigCloneBench Subset containing polymorphic programs and Custom datasets. All axes are log-scale.

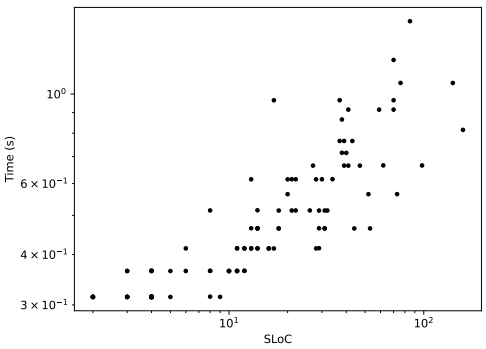
terminate and the SLoC of programs of each dataset are shown in Figures 11a and 11b respectively. Finally, the relationship between JavaCIP run time and program SLoC for the Spring Boot Web and BigCloneBench datasets are plotted in Figures 12a and 12b respectively.

5.2 RQ1: Effectiveness

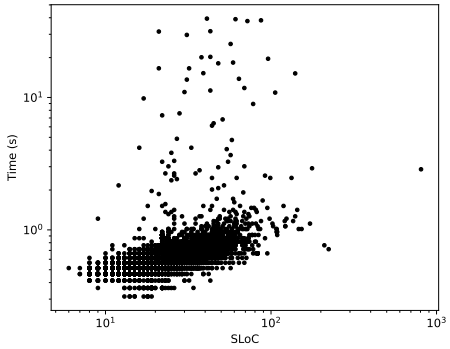
There are four possible outcomes when JavaCIP receives an incomplete program:

True positive: JavaCIP produces completions which are **compilable**
False positive: JavaCIP produces completions that are **not compilable**
True negative: JavaCIP **cannot complete** the incomplete program because it is **impossible**
False negative: JavaCIP **cannot complete** the incomplete program although it is **possible**

The data shows that 108 (91.5%) and 4848 (99.1%) programs from the Spring Boot Web and BigCloneBench datasets respectively were completed by JavaCIP, and in both cases, 100% of the programs produced by JavaCIP were able to be compiled by `javac`. Only 1 program from the Spring Boot Web dataset cannot be completed because JavaCIP did not generate subtype constraints from access modifiers—if a class S accesses a protected field/method from another class T then $S <: T$. Instead, the current behaviour of JavaCIP is to reject programs that invoke protected methods that are defined in built-in Java classes. This edge case is therefore a false negative and we leave this for future implementation. On the other hand, all 8 programs that cannot be completed in the BigCloneBench dataset were impossible to complete because of references to methods that cannot exist, such as method invocations like `this.getLocation()` in a class that does not declare the method `getLocation` (we believe this happens in BigCloneBench because some programs were generated by extracting a single method from a large class and placed in a new class that only contains that method). These are therefore true negative results. Finally, 9 (7.6%) and 33 (0.7%) of programs from the Spring Boot Web and BigCloneBench datasets respectively cannot be completed within the 1-minute time limit, and therefore contribute to the false negative rate of JavaCIP. Therefore, we conclude that JavaCIP operates correctly in 91.5% and 99.3% of programs in the Spring Boot Web and BigCloneBench datasets respectively, and operates incorrectly in 8.5% and 0.7% of programs in the



(a) Run time against incomplete program SLoC for Spring Boot Web dataset.



(b) Run-time against incomplete program SLoC for BigCloneBench dataset.

Figure 12: JavaCIP run time against incomplete program SLoC. All axes are log-scale.

Spring Boot Web and BigCloneBench datasets respectively. We summarize these results in Table 2b.

Comparison to JCoffee. Gupta et al. [2020] reports that JCoffee completed 8220 (90%) of the 9133 programs in the original BigCloneBench dataset. Unfortunately, we were not able to reliably run JCoffee without it reporting "some error occurred".[1] This means we cannot measure its performance on the subsets used in our paper, but we note that Gupta et al. [2020] reports that JCoffee has difficulty dealing with polymorphic types. In contrast, JavaCIP worked correctly for 432 (99.1%) of the 436 programs in the BigCloneBench subset containing polymorphic types (Table 2a).

5.3 RQ2: Runtime Efficiency

Experimental data shows that for the Spring Boot Web dataset, JavaCIP successfully completed 108 programs in an average of 0.5s, with a minimum and maximum runtime of 0.31s and 1.52s respectively. This is for programs with an average SLoC of 23.9, with the smallest and largest files being of 2 SLoC and 159 SLoC respectively. For the BigCloneBench dataset, JavaCIP successfully completed 4848 programs in an average of 0.74s, with a minimum and maximum runtime of 0.31s and 39.40s respectively. Most notably, JavaCIP is even able to complete the largest program in this dataset with 806 SLoC (JavaCIP completed this program in 2.87s).

We plot the relationship between JavaCIP runtime and program SLoC in Figure 12. The plot shows that broadly, as the size of the program increases, the time required to reconstruct its missing dependencies increases as well.

Comparison with JCoffee. As before, we are unable to replicate the results reported for JCoffee, thus we must rely on their reported figures for comparison. The authors of JCoffee report that JCoffee completed the programs in the BigCloneBench dataset in 1.64s on average [Gupta et al., 2020]. We believe our average run-time result outperforms their reported figures only because (1) our experiments are performed on modern hardware and (2) JCoffee is implemented in Python. As such, comparisons on this basis are inconclusive. However, unlike JCoffee, JavaCIP timed-out for some programs. We believe this is evidence that the algorithm that we have presented suffers from *path explosion*.

[1] In our own investigation, the errors appear to stem from implementation issues. JCoffee has not been actively maintained since the time of its publication, and although benchmark programs were published, the exact programs which JCoffee failed to complete were not identified. We have reached out to the authors of JCoffee to see if they can

```
1 class Test {
2   void main() {
3     A a = new A();
4     B<? extends C> b1 = new B<>();
5     B<? extends D> b2 = new B<>();
6     C c = a.id(a.extract(b1));
7     D d = a.id(a.extract(b2));
8   }
9 }
10 class B<T> { }
11 class C { }
12 class D { }
```

(a) Input incomplete program $\mathcal{P}_4$ (a) that causes JavaCIP to suffer path explosion.

```
1 class Test {
2   void main() {
3     A a = new A();
4     B<? extends C> b1 = new B<>();
5     B<? extends D> b2 = new B<>();
6     C c = a.id((C) a.extract(b1));
7     D d = a.id((D) a.extract(b2));
8   }
9 }
10 class B<T> { }
11 class C { }
12 class D { }
```

(b) Input incomplete program $\mathcal{P}_4$ (b) which adds type casting to $\mathcal{P}_4$ (a) in lines 6 and 7.

Figure 13: Example incomplete program that suffers from path explosion.

5.3.1 *Path Explosion.* Several analyses like Symbolic Execution [King, 1976] suffer from *path explosion* [Cadar, Sen, 2013], which JavaCIP also suffers from. Among the programs that cause timeouts in our experiments, a recurring programming pattern we observe is in the form of expressions like `g(f())`, where `f` and `g` are not declared. This forces the resolution of the method parameter/return types to the last stage DeclareMethods, which requires a lot of analysis. For example, JavaCIP suffers a timeout without being able to complete $\mathcal{P}_4$ (a) shown in Figure 13a. Just by specifying the argument types to the method invocations via typecasting as shown in Figure 13b, JavaCIP is able to complete the program in less than half a second. We view this as a limitation of our approach and we look forward to future work mitigating the path explosion that our algorithm faces.

5.4 RQ3: Wrong Completions

The results in Table 2b show that JavaCIP has a 0% false positive rate. This is expected, because the techniques we have presented produce programs that satisfy the constraints generated during type checking by Java's compiler. However, there are two potential types of false positives: (1) incorrect completions for programs that are *possible* to complete and (2) *any* completion of programs that are *impossible* to complete. While the dataset contains 5029 programs (99.8%) that are possible to complete (type 1), and JavaCIP never produced incorrect completions for those, the dataset contains only 8 programs (0.2%) that are impossible to complete (type 2).

An example of a program that is impossible to complete is $\mathcal{P}_5$ shown in Figure 14. $\mathcal{P}_5$ requires that the type of `a.x` is a subtype of `String`, and yet a supertype of `Object`. It is impossible to ascribe a type to the declaration of `x` in `A` that meets these constraints, therefore $\mathcal{P}_5$ is impossible to complete and JavaCIP should not generate a completion for it.

```
1 class B {
2   static void main() {
3     A a = new A();
4     a.x = new Object();
5     String s = a.x;
6   }
7 }
```

Figure 14: Incomplete program $\mathcal{P}_5$.

Running JavaCIP against a realistic and representative dataset of incomplete programs that are impossible to complete would increase our confidence in the correctness of JavaCIP. However, obtaining such a dataset is difficult, and we are not aware of anything similar the literature. Nevertheless, to give a *rough indication* of how JavaCIP would behave against such a dataset, we created a custom dataset containing 30 handwritten incomplete programs that refer to or require the generation of polymorphic types. Among the 30 programs, 15 of these can be completed, while the remaining 15 are impossible to complete. As

assist us in fixing the issues we faced.

shown in Table 2a, JavaCIP produces correct results for all 30 programs, reconstructing missing dependencies of the first 15 programs, and terminating without generating completions for the remaining 15. No timeouts occurred on these 30 programs.

6 LIMITATIONS AND FUTURE WORK

Completion Ambiguity. An incomplete program may have more than one correct completion. For example, if an incomplete program only contains `System.out.println(1 + size())`, a completion that declares `size()` to return `int` and another that declares `size()` to return `String` both allow the program to become complete and well-typed. In this example, neither of these completions are "more correct" than the other, and our algorithm produces the first completion it encounters. Future work that aims to generate more "intuitive" code may consider other factors when deciding which completion is preferred. For example, the method name `size` hints that a numeric type is desired, and extensions to our algorithm may incorporate this knowledge in deciding that `size` should return `int`.

Undecidability of Java Subtyping. Java's parametrically polymorphic type system is Turing complete, and subtyping in Java (which is central to type checking in Java) is *undecidable* [Grigore, 2017]. Therefore, because our algorithm extends type checking and type inference, incomplete programs that cause type checking to never terminate will also cause our algorithm to never terminate. However, these programs do not generally pose a problem to us, because even if we were able to reconstruct their surrounding dependencies, the Java compiler will still be unable to compile them. On the other hand, we must be careful to not *generate* surrounding dependencies that cause type checking not to terminate. It has been shown that subtyping in the absence of contravariant type constructors is *decidable* [Kennedy, Pierce, 2007]. Consequently, we prohibit our algorithm from generating classes that inherit types containing upper-bounded wildcard arguments (for example, `class C<T> extends N<N<? super C<C<T>>>>`). We believe this restriction poses little threat to the completeness of our algorithm—as our experiments reveal, the need to generate class inheritance with upper-bounded wildcard arguments is rare.

Infinite Search Domain. As stated earlier, when an incomplete program requires the generation of a new parametrically polymorphic type S, we know of no mechanism to determine the maximum arity or nesting depth that S can have. Therefore, our algorithm requires user assistance to determine the maximum arity and nesting depth of all new types to ensure termination. As stated in Section 4.1.3, this affects the completeness of our algorithm, but in practice, assigning a reasonably large number should suffice for majority of programs. For example, we assigned a maximum arity and depth of 3 for our experiments.

Unsupported Features. As discussed in Section 4.2, JavaCIP has not implemented support for several Java features. Furthermore, our experimentation shows that many programs use features that are not supported by JavaCIP. Some of these features like annotations and anonymous classes can be implemented in JavaCIP without significant extensions to the underlying algorithm, while the others, like lambda expressions, require significant work. We leave the support for those features for future work.

Path Explosion. As discussed in Section 5.3.1, path explosion poses a threat to the effectiveness of our algorithm. This may be mitigated with some heuristics and more sophisticated analyses that can prune the search space further than we did in Section 3.2.3, which we leave for future work.

7 RELATED WORK

The works that are the most similar in research objectives are JCoffee by Gupta et al. [Gupta et al., 2020], PPA by Dagenais and Hendrenis [Dagenais, Hendrenis, 2008], and PsycheC by Melo et al. [Melo et al., 2017]. JCoffee leverages the verbosity of the Java Compiler's feedback and reconstructs the missing dependencies of partial programs by fixing compiler errors. The PPA algorithm, instead of reconstructing missing dependencies, receives a partial Java program and uses constraint generation and solving to directly produce Jimple intermediate code to support static analysis. PsycheC on the other hand works on incomplete C sources via a syntax-directed process of constraint generation, then uses a two-phase unification [Martelli, Montanari, 1982] approach to solve typing constraints. All of these three algorithms do not support incomplete programs with parametrically polymorphic types.

Adjacent to our work is *type inference*, for which typically some form of unification [Robinson, 1965; Martelli, Montanari, 1982] is employed, for example, Dolan and Mycroft's biunification [Dolan, Mycroft, 2017] to perform type inference in MLSub, which admits subtyping in its type system. Type inference in general is similar to our research objectives, and we base the overall structure of our algorithm based on well-known type inference algorithms.

AI-powered program synthesis is not a new concept [Manna, Waldinger, 1971], with several works [Odena et al., 2021; Jain et al., 2022] leveraging Pre-Trained LLMs like GPT-3 [Brown et al., 2020] to perform program synthesis based on a mixture of `assert` statements and/or natural language descriptions of the intended operation of the synthesized program. However, LLMs like ChatGPT [OpenAI, ????] are currently not suited for our purposes. As of writing this paper, LLMs do not understand program semantics and treat programs as text [Odena et al., 2021]—our problem requires knowledge of types in the program and semantics of type inference/checking to ensure that the synthesized program is complete and well-typed. In addition, LLMs suffer from *hallucination* [Ji et al., 2023] and can produce ill-typed programs, or cite the existence of type errors despite the program being well-typed.

Other works dealing with incomplete programs include GRAPA [Zhong, Wang, 2017] that locates Java archive files to resolve unknown constructs and build System Dependency Graphs [Ferrante et al., 1987] for analysis of partial programs, ParseWeb [Thummalapenta, Xie, 2007] and PRIME [Mishne et al., 2012] which obtain API calls from partial code snippets and recommends method calls with matching signatures from the web and a variety of other mined sources, and SnR [Dong et al., 2022] which determines matching import statements where they are missing.

8 CONCLUSION

We described a novel algorithm that is able to reconstruct the missing dependencies of incomplete polymorphic programs. The core of the algorithm extends type checking and type inference techniques, where the constraint generation phase identifies the search domain of types, and the constraint solving phase systematically searches for new types. Our implementation outperforms earlier works by being able to deal with polymorphic types, and by achieving an overall success rate of 99.1% across all code samples.

We propose that the core ideas we have presented can be extended to support programs containing other Java features not discussed, and may even be extended to achieve the same objective for incomplete programs written in other commonly-used programming languages with similar type systems, such as C++, Rust and others.

REFERENCES

JavaParser. ????

Anvik John, Murphy Gail C. Determining Implementation Expertise from Bug Reports // Proceedings of the Fourth International Workshop on Mining Software Repositories. USA: IEEE Computer Society, 2007. 2. (MSR '07).

Ayewah Nathaniel, Pugh William, Morgenthaler J. David, Penix John, Zhou YuQian. Using FindBugs on Production Software // Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion. 2007. 805–806.

Bettenburg Nicolas, Premraj Rahul, Zimmermann Thomas, Kim Sunghun. Extracting Structural Information from Bug Reports // Proceedings of the 2008 International Working Conference on Mining Software Repositories. 2008. 27–30.

Brown Tom, Mann Benjamin, Ryder Nick, Subbiah Melanie, Kaplan Jared D, Dhariwal Prafulla, Neelakantan Arvind, Shyam Pranav, Sastry Girish, Askell Amanda, Agarwal Sandhini, Herbert-Voss Ariel, Krueger Gretchen, Henighan Tom, Child Rewon, Ramesh Aditya, Ziegler Daniel, Wu Jeffrey, Winter Clemens, Hesse Chris, Chen Mark, Sigler Eric, Litwin Mateusz, Gray Scott, Chess Benjamin, Clark Jack, Berner Christopher, McCandlish Sam, Radford Alec, Sutskever Ilya, Amodei Dario. Language Models are Few-Shot Learners // Advances in Neural Information Processing Systems. 33. 2020. 1877–1901.

Cadar Cristian, Sen Koushik. Symbolic Execution for Software Testing: Three Decades Later // Commun. ACM. feb 2013. 56, 2. 82–90.

Chugh Ravi, Meister Jeffrey A., Jhala Ranjit, Lerner Sorin. Staged Information Flow for Javascript // Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009. 50–62.

Dagenais Barthelemy, Hendrenis Laurie. Enabling static analysis for partial java programs // Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. 10 2008. 313–328.

Dagenais Barthélémy, Robillard Martin. Recommending adaptive changes for framework evolution // 2008 ACM/IEEE 30th International Conference on Software Engineering. 2008. 481–490.

Dolan Stephen, Mycroft Alan. Polymorphism, Subtyping, and Type Inference in MLsub // Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. 2017. 60–72.

Dong Yiwen, Gu Tianxiao, Tian Yongqiang, Sun Chengnian. SnR: Constraint-Based Type Inference for Incomplete Java Code Snippets // Proceedings of the 44th International Conference on Software Engineering. 2022. 1982–1993.

Duggan Dominic, Bent Frederick. Explaining type inference // Science of Computer Programming. 1996. 27, 1. 37–83.

Ferrante Jeanne, Ottenstein Karl J., Warren Joe D. The Program Dependence Graph and Its Use in Optimization // ACM Transactions on Programming Languages and Systems. 7 1987. 9, 3. 319–349.

Godefroid Patrice. Micro Execution // Proceedings of the 36th International Conference on Software Engineering. 2014. 539–549.

Gosling James, Joy Bill, Steele Guy, Bracha Gilad. Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)). 2005.

Grigore Radu. Java Generics Are Turing Complete // Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. New York, NY, USA: Association for Computing Machinery, 2017. 73–85. (POPL '17).

Guimarães Breno C F, Magalhães José Wesley de S, Silva Anderson Faustino da, Pereira Fernando M Q. Synthesis of Benchmarks for the C Programming Language by Mining Software Repositories // Proceedings of the XXIII Brazilian Symposium on Programming Languages. 2019. 62–69.

Gupta Piyush, Mehrotra Nikita, Purandare Rahul. JCoffee: Using Compiler Feedback to Make Partial Code Snippets Compilable // 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2020. 810–813.

Hua Wei, Sui Yulei, Wan Yao, Liu Guangzhong, Xu Guandong. FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks // IEEE Transactions on Reliability. 2021. 70, 1. 304–318.

Jain Naman, Vaidyanath Skanda, Iyer Arun, Natarajan Nagarajan, Parthasarathy Suresh, Rajamani Sriram, Sharma Rahul. Jigsaw: Large Language Models Meet Program Synthesis // Proceedings of the 44th International Conference on Software Engineering. New York, NY, USA: Association for Computing Machinery, 2022. 1219–1231. (ICSE '22).

Ji Ziwei, Lee Nayeon, Frieske Rita, Yu Tiezheng, Su Dan, Xu Yan, Ishii Etsuko, Bang Ye Jin, Madotto Andrea, Fung Pascale. Survey of Hallucination in Natural Language Generation // ACM Comput. Surv. mar 2023. 55, 12.

Kennedy Andrew, Pierce Benjamin C. On Decidability of Nominal Subtyping with Variance // International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD). January 2007.

King James C. Symbolic Execution and Program Testing // Commun. ACM. jul 1976. 19, 7. 385–394.

Knapen G., Lague B., Dagenais M., Merlo E. Parsing C++ despite missing declarations // Proceedings Seventh International Workshop on Program Comprehension. 1999. 114–125.

Mani Senthil, Padhye Rohan, Sinha Vibha Singhal. Mining API Expertise Profiles with Partial Program Analysis // Proceedings of the 9th India Software Engineering Conference. New York, NY, USA: Association for Computing Machinery, 2016. 109–118. (ISEC '16).

Manna Zohar, Waldinger Richard J. Toward Automatic Program Synthesis // Commun. ACM. mar 1971. 14, 3. 151–165.

Martelli Alberto, Montanari Ugo. An Efficient Unification Algorithm // ACM Transactions on Programming Languages and Systems (TOPLAS). 4 1982. 4, 2. 258–282.

Mehrotra Nikita, Agarwal Navdha, Gupta Piyush, Anand Saket, Lo David, Purandare Rahul. Modeling Functional Similarity in Source Code With Graph-Based Siamese Networks // IEEE Transactions on Software Engineering. 2022. 48, 10. 3771–3789.

Melo Leandro T. C., Ribeiro Rodrigo G., Araújo Marcus R. de, Pereira Fernando Magno Quintão. Inference of Static Semantics for Incomplete C Programs // Proceedings of the ACM on Programming Languages. 12 2017. 2, POPL.

Mishne Alon, Shoham Sharon, Yahav Eran. Typestate-Based Semantic Code Search over Partial Programs // Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. 2012. 997–1016.

Mockus Audris, Herbsleb James D. Expertise browser: a quantitative approach to identifying expertise // Proceedings of the 24th International Conference on Software Engineering. New York, NY, USA: Association for Computing Machinery, 2002. 503–512. (ICSE '02).

Odena Augustus, Sutton Charles, Dohan David Martin, Jiang Ellen, Michalewski Henryk, Austin Jacob, Bosma Maarten Paul, Nye Maxwell, Terry Michael, Le Quoc V. Program Synthesis with Large Language Models // n/a. n/a, 2021. n/a. n/a.

Odersky Martin, Altherr Philippe, Cremet Vincent, Dubochet Iulian Dragos Gilles, Emir Burak, McDirmid Sean, Micheloud Stéphane, Mihaylov Nikolay, Schinz Michel, Stenman Erik, Spoon Lex, Zenger Matthias. An Overview of the Scala Programming Language. 2006.

OpenAI . Introducing ChatGPT. ????

Parnin Chris, Bird Christian, Murphy-Hill Emerson. Adoption and Use of Java Generics // Empirical Software Engineering. 12 2013. 18, 6. 1047–1089.

Perelman Daniel, Gulwani Sumit, Ball Thomas, Grossman Dan. Type-Directed Completion of Partial Expressions // Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. 2012. 275–286.

Robinson J. A. A Machine-Oriented Logic Based on the Resolution Principle // J. ACM. 1 1965. 12, 1. 23–41.

Rodrigues Marcus, Guimarães Breno, Pereira Fernando Magno Quintão. Generation of In-Bounds Inputs for Arrays in Memory-Unsafe Languages // 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2019. 136–148.

Sun Weisong, Fang Chunrong, Chen Yuchen, Tao Guanhong, Han Tingxu, Zhang Quanjun. Code Search based on Context-aware Code Translation // 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). 2022. 388–400.

Svajlenko Jeffrey, Islam Judith F., Keivanloo Iman, Roy Chanchal K., Mia Mohammad Mamun. Towards a Big Data Curated Benchmark of Inter-project Code Clones // 2014 IEEE International Conference on Software Maintenance and Evolution. 2014. 476–480.

Thummalapenta Suresh, Xie Tao. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web // Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. 2007. 204–213.

Vallée-Rai Raja, Co Phong, Gagnon Etienne, Hendren Laurie, Lam Patrick, Sundaresan Vijay. Soot: A Java Bytecode Optimization Framework // CASCON First Decade High Impact Papers. 2010. 214–224.

Wang Shaowei, Lo David, Jiang Lingxiao. An empirical study on developer interactions in StackOverflow // Proceedings of the 28th Annual ACM Symposium on Applied Computing. New York, NY, USA: Association for Computing Machinery, 2013. 1019–1024. (SAC '13).

Williams Chadd C., Hollingsworth Jeffrey K. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques // IEEE Trans. Softw. Eng. jun 2005. 31, 6. 466–480.

Xue Zhipeng, Zhang Yuanliang, Xu Rulin. Clone-Based Code Method Usage Pattern Mining // Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. 2022. 543–547.

Zhang Jian, Wang Xu, Zhang Hongyu, Sun Hailong, Liu Xudong, Hu Chunming, Liu Yang. Detecting Condition-Related Bugs with Control Flow Graph Neural Network // Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. New York, NY, USA: Association for Computing Machinery, 2023. 1370–1382. (ISSTA 2023).

Zhao Wei, Zhang Lu, Liu Yin, Sun Jiasu, Yang Fuqing. SNIAFL: Towards a static noninteractive approace to feature location // ACM Transactions on Software Engineering and Methodology. 4 2006. 15, 2. 195–226.

Zhong Hao, Wang Xiaoyin. Boosting Complete-Code Tool for Partial Program // Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. 2017. 671–681.