Type-Safe Auto-Completion of Incomplete Polymorphic Programs

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

20 October 2023

ABSTRACT. Incomplete programs are ubiquitous in both web repositories and evolving software projects. The ability to perform auto-completion on these incomplete programs provides several benefits, including allowing static analysis tools to analyse incomplete programs and boosting developer productivity. However, earlier efforts to do so (1) are unable to work with incomplete programs containing parametrically polymorphic types and (2) may not respect the type safety of the incomplete program.

In this paper we present an algorithm 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. As this algorithm does so in a type-safe manner, it can also find non-trivial type errors in incomplete programs. Finally, we present results from a prototype implementation of this algorithm, JavaCIP.

Keywords: incomplete programs, program synthesis, type inference

1 INTRODUCTION

This paper focuses on *incomplete Java programs*, which we define as syntactically correct and nonempty Java compilation units (consisting of package/import/class/interface declarations) containing references to undeclared constructs. This definition is more loose than that of *partial programs* [DH08, GMP20, MRdAP17] in that we do not assume that it is a subset of an otherwise complete and well-typed program, but syntactical correctness can still be assumed because it can be easily verified by a parser.

Incomplete programs present themselves in many situations—during the planning, designing and prototyping phases of the software development life cycle, patches for code review, software versioning systems [DR08], in web repositories [TX07], bug reports [BPZK08], forum threads, documentation etc. However, incomplete programs prove difficult to work with. For example, program analyzers often work on an executable or intermediate representation (IR) of source code. These can usually be built by a compiler, but compilers cannot compile incomplete programs because they contain references to undeclared constructs. In addition, type checkers cannot verify the type safety of incomplete programs because the types of some expressions cannot be determined.

The ubiquity of incomplete programs, coupled with the difficulty of working with them, resulted in a wide variety of works produced by members the programming languages community to deal with incomplete programs [DH08, GMP20, MRdAP17, CMJL09, God14, KLDM99, PGBG12, RGP19, GaMadSP19, XZX22, DGTS22]. Among these works, incomplete program auto-completion alleviates many of the main difficulties faced:

1. Program analysis is often relied upon during bug detection [APM+07], language manipulation and optimization [VRCG+10] and feature location [ZZL+06]. With incomplete program auto-completion, researchers now have access to a wider corpus of programs that can be collectively analysed, giving insight into software project evolution and program development behavior for software engineers;

2. Program auto-completion and stub generation boosts developer productivity;

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, Website: https://www.comp.nus.edu.sg/~khoosc/, *National University of Singapore.*

3. Type-safe auto-completion additionally serves the benefit of type-checking in an evolving software project;

We highlight three key challenges that remain unsolved in the realm of auto-completion of incomplete programs:

1. earlier works do not support programs with *parametrically polymorphic* (or *generic*) types [GJSB05], which many programs contain (such as in Java's Collections Framework [PBMH13]);

2. earlier works often operate on *partial programs* that obey the *compilable program assumption* [DH08, GMP20]—that a partial program can always be made into a complete, well-typed and compilable program—but this assumption is violated during the evolution of a project when code is syntactically correct but contains type errors that cannot be caught by a type-checker due to its incompleteness;

3. several earlier works do not make guarantees on preserving type safety of the original incomplete programs [DH08, GMP20].

In this paper, we propose a novel algorithm that solves these three challenges to perform auto-completion of incomplete polymorphic programs in a type-safe manner so that the resulting program is complete and well-typed. Our algorithm reads an incomplete program P that may contain parametrically polymorphic types and produces a program R satisfying the condition that P and R together is complete and well-typed. We call the set of all programs satisfying that condition $\mathcal{R}$.

Similar to many other related works on type checking and type inference [DH08, MRdAP17, Rob65, MM82, DM17], our algorithm employs two phases, *constraint generation* and *constraint solving*. It works by obtaining *constraints* about the types that occur in P that must be true based on Java's type system. Then, for each of these constraints, it either verifies that it is satisfied, or, if it relies on missing types, declarations of the missing types are produced so that it holds. If any constraint does not hold or cannot be made to hold, then $\mathcal{R} = \emptyset$, and therefore type errors must be present in the program.

1.1 Motivating Examples

```
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 P_1

```
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 }
```

(b) Program R_1 generated by JavaCIP such that P_1 and R_1 form a complete and well-typed program.

Figure 1: An example of JavaCIP successfully completing an incomplete program.

Example 1.1. Suppose a developer working on a project has written the incomplete program P_1 shown in Figure 1a. This program is incomplete because it refers to types `A`, `C`, `D`, but no declaration of these types is found. Our goal is to produce a program R_1 that declares these types such that P_1

and R_1 combined form a complete and well-typed program. To do so, we can observe some properties of these types and their members:

- The assignment statement `a = b` in line 6 asserts that the type of `b` must be compatible in an assignment context with the type of `a`. We know from lines 4 and 5 that `a` has type `A<? extends C>` and `b` has type `B<D>`, so we assert that `B<D>` is a *subtype* of `A<? extends C>`. We write this as `B<D> <: A<? extends C>` and let this be constraint (C1).
- The expression `c.get()` in line 9 tells us that the declaration of `C` contains a method `get` that takes no arguments and returns an object of an unknown type. We let this unknown type be τ_1. In the same line, the assignment statement `List<? super Integer> i = c.get();` tells us τ_1 `<: List<? super Integer>`, and we let this constraint be (C2).
- Similarly, in line 10, we know that `D` must have a method `get` that takes no arguments and returns some type τ_2 and τ_2 `<: List<? super String>`, and we let this constraint be (C3).

To resolve the existence of `A`, `C` and `D` and the method `get()`, we can immediately generate the class declarations `class A<T> { }`, `class C {` τ_1 `get() { }}` and `class D {` τ_2 `get() { }}`. Next, we ensure that the three constraints (C1), (C2) and (C3) hold. We do so by working with Java's typing rules. To solve (C1), we observe that `B<T>` extends `A<T>`, which tells us that `B<D>` extends `A<D>` (we write this as `B<D>` $\rightsquigarrow$ `A<D>`). Therefore, based on Java's typing rules, the constraint (C1A) `A<D> <: A<? extends C>` implies (C1). Similarly, the constraint (C1B) `D <: C` implies (C1A). We express this notion of implication between constraints as *reductions* on constraint (C1) like so:

$$\frac{\Delta \vdash \texttt{B<D>} \rightsquigarrow \texttt{A<D>}}{\Delta \vdash \texttt{B<D>} <: \texttt{A<? extends C>} \Rightarrow \texttt{A<D>} <: \texttt{A<? extends C>}}\ \textsc{Red-Upcast}$$

$$\frac{}{\vdash \texttt{A<D>} <: \texttt{A<? extends C>} \Rightarrow \texttt{D} <: \texttt{C}}\ \textsc{Red-Args}$$

However, constraint (C1B) is irreducible since the constraint `D` $\rightsquigarrow$ `C` is required to prove it. Thus, to solve (C1B) we allow class `D` to extend `C`. This modification to our new class declaration solves (C1B), and since (C1B) implies (C1A) and (C1A) implies (C1), (C1) has also been solved.

Next, we solve constraint (C2). As τ_1 does not have any other constraints on it, by letting τ_1 be equal to `List<? super Integer>`, our constraint holds. Therefore, our newly generated class declaration for `C` contains `List<? super Integer> get() {...}`. In a similar fashion, we can resolve constraint (C3) τ_2 `<: List<? super String>` by letting τ_2 = `List<? super String>`, so our newly generated class declaration for `D` contains the method declaration `List<? super String> get() {...}`.

We have resolved all constraints on our incomplete program, but our job is incomplete. From solving (C1) we inferred that `D` extends `C`, thus all members of `C` are inherited by `D`. It means that `D` has access to both methods `List<? super Integer> get() {...}` and `List<? super String> get() {...}`. This method cannot be overloaded, because based on Java's type system, the erasure of the input signatures of both methods are the same (in this case, both methods do not receive any parameters). Therefore, the declaration of `get` in `D` must *override* `List<? super Integer> get() {...}` in `C`, and in doing so, its return type must be a subtype of both `List<? super Integer>` and `List<? super String>`. To solve this, we replace the method declaration of `get` in `D` to one that returns a new type τ_3, constrained by (C4) τ_3 `<: List<? super Integer>` and (C5) τ_3 `<: List<? super String>`. Solving (C4) and (C5) gives one possible type τ_3 = `List<Object>`, so we let that be the return type of `get` in `D`. Now all constraints have truly been solved, and we can build program R_1 shown in Figure 1b which, together with P_1, is complete and well-typed.

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

Figure 2: Incomplete program P_2

Example 1.2. Next, show a scenario where non-trivial type errors in incomplete programs can be discovered. Suppose a developer writes the incomplete program P_2 in Figure 2 instead, which is the same as P_1 except for lines 8 and 9. A compiler or type checker would only report that the declarations of `A`, `C` and `D` are missing. Instead, our algorithm obtains the constraints $\tau_1 <: \texttt{int}$ and $\tau_2 <: \texttt{String}$, and similarly infers that the declaration of class `C` contains `int get() {...}` and the declaration of `D` contains `String get() {...}`. Again, because `D` <: `C`, `D` has access to both methods `int get() {...}` and `String get() {...}`, so the declaration of `get` in `D` must override `int get() {...}` in `C`, and in doing so, its return type must be a subtype of both `int` and `String`. This time, no such type can exist because the `Integer` and `String` classes are declared as `final` (meaning no class can extend them) and are not subtypes of each other. As no overriding declaration of `get` can exist in `D`, this tells us that `D` cannot be a subtype of `C`, therefore `A<D>` cannot be a subtype of `A<? extends C>`, therefore `B<D>` cannot be a subtype of `A<? extends C>`, therefore the assignment `a = b` is erroneous, which finally means that this incomplete program has a type error.

1.2 Limitations

The nature of this problem and results from earlier works impose limits on our algorithm that are important to highlight. Firstly:

1. Type inference algorithms for languages like ML or Haskell infer a *principal type* or a *most general unifier* [DB96], preventing type checkers from reporting false-positive type errors. However, the notion of *principality* is ill-defined in our problem, and the generation of principal types (if they could be defined) is not necessary to complete an incomplete program.

2. Suppose a developer has written an incomplete program P, and *intends* to let the surrounding dependencies be a *particular* $R^* \in \mathcal{R}$. In general, P alone may not give enough information to determine what R^* is. For example, if P only contains `println(1 + get())` then in the absence of other constraints, definitions of `get()` that return `1`, `2`, or even a string `'a'` can all complete P in a type-safe manner.

Therefore, *if our algorithm produces some R and $\mathcal{R}$ contains more than one program, we consider R to be correct as long as $R \in \mathcal{R}$.* On a similar note, if the input program P is already complete and well-typed, our algorithm produces no output.

Secondly, Java's parametrically polymorphic type system is Turing complete, and its subtyping (which is central to type checking in Java) is *undecidable* [Gri17]. As it has been shown that subtyping in the absence of contravariant type constructors is *decidable* [KP07], *our algorithm only allows and generates classes where inheritance makes no use of contravariance.*

Thirdly, when an incomplete program P requires the generation of a new parametrically polymorphic type S, we know of no mechanism to determine the maximum *arity* (number of type parameters/arguments) or *height* (*height*(`S<T>`) = 1 + *height*(T)) that S can have (*arity* and *height* are more precisely defined in Section 2). Therefore, *our algorithm requires user assistance to determine the maximum arity and height of all new types, to ensure termination.* For example, if the user of our algorithm specifies the maximum arity and height of new types to be 1 and 2 respectively, then our algorithm does not consider any program in $\mathcal{R}$ that contains types like `Map<K, V>` or `A<B<C<D>>>`;

2 PRELIMINARIES

In this section, we briefly define notation and relations needed to describe our algorithm. We omit descriptions of important features like interfaces, primitive types and constructors to describe the core ideas of our algorithm. These features can be handled via relatively conservative extensions to our algorithm, which our implementation JavaCIP described in Section 5 does.

A variety of works have modelled and studied Java's type system in detail [IPW01, TEH05, CDE08, SCDCD10, KP07, Gri17, CD09, RV05], from which we shall extensively borrow notation. Precise typing rules can be found in these works and the Java Language Specification [GJSB05], which we will not deviate from, and only re-state here. Readers familiar with subtyping, capture conversion and classes in Java may skip ahead to Section 3 on the description of our algorithm.

2.1 Types

We let uppercase emphatic letters like S and T be meta-variables over the types, and uppercase blackboard letters like $\mathbb{S}$ and $\mathbb{T}$ be meta-variables over sets of types. In some instances we need to decompose a parametrically polymorphic type into its raw type and its type parameters/arguments, for this we write $S\texttt{<}P_1, \ldots, P_n\texttt{>}$. For example, if we let $S\texttt{<}P_1, P_2\texttt{>}$ be the type `Pair<Integer,String>`, then $S = \texttt{Pair}$, $P_1 = \texttt{Integer}$ and $P_2 = \texttt{String}$. Following [KP07] and [Gri17], the *arity* of a type is its number of parameters/arguments ($\mathit{arity}(S\texttt{<}A_1, \ldots, A_n\texttt{>}) = n$); and the *height* of a type is 1 for a type of arity 0, and 1 more than the maximum height of its arguments in every other case:

$$\mathit{height}(S) = \begin{cases} 1 & \text{if } S \text{ is not parametrically polymorphic} \\ 1 + \max\{\mathit{height}(A_i) \mid 1 \leq i \leq n\} & \text{if } S = T\texttt{<}A_1, \ldots, A_n\texttt{>} \end{cases}$$

With this we can inductively define the set of all valid Java types with height n, given as $\mathbb{T}_n$, where without loss of generality, we assume that all class types have arity no more than 1: (1) if S is a type parameter or a class type with arity 0, $S \in \mathbb{T}_1$, (2) if S is a class type with arity 1 then $S\texttt{<?>} \in \mathbb{T}_2$, (3) if S is a class type with arity 1 and $T \in \mathbb{T}_{n-1}$ then $S\texttt{<}T\texttt{>}, S\texttt{<? extends }T\texttt{>}, S\texttt{<? super }T\texttt{>} \in \mathbb{T}_n$.

A *substitution* is a function $\phi = \{P_1 \mapsto A_1, \ldots, P_n \mapsto A_n\}$ to substitute all occurrences of type parameters P_1 to P_n with type arguments A_1 to A_n respectively of a type, and we write $S\phi$ to describe the resulting type obtained from perfoming substitution ϕ on S. Correspondingly, given any type $T = S\texttt{<}A_1, \ldots, A_n\texttt{>}$ we can recover this substitution by obtaining the *expansion* of T, giving us $(S\texttt{<}P_1, \ldots, P_n\texttt{>}, \{P_1 \mapsto A_1, \ldots, P_n \mapsto A_n\})$ where P_1 to P_n are the type parameters declared in the class declaration of S. The *erasure* of a type $S\texttt{<}A_1, A_2\texttt{>}$, denoted $\epsilon(S\texttt{<}A_1, A_2\texttt{>})$ is its raw type S, and the erasure of a type parameter is `java.lang.Object` (this is because type parameters have no bounds in our model).

2.2 Wildcards

All types in Java are *declaration-site invariant*, thus Java introduced wildcard type arguments [GJSB05] to support *use-site variance* [THE+04]. Wildcard arguments take three forms: upper-bounded wildcards like $S\texttt{<? }\textbf{\texttt{extends}}\texttt{ }T\texttt{>}$, lower-bounded wildcards like $S\texttt{<? }\textbf{\texttt{super}}\texttt{ }T\texttt{>}$, and unbounded wildcards like $S\texttt{<?>}$. To make Java's wildcards more useful, Java supports *capture conversion* on types, which performs an *open* operation on type arguments to capture wildcard arguments as *globally-fresh* potentially bounded captured type variables:

$$\mathit{open}(S) = S \qquad \mathit{open}(\texttt{? }\textbf{\texttt{extends}}\texttt{ }S) = \exists x \lhd S.x \qquad \mathit{open}(\texttt{? }\textbf{\texttt{super}}\texttt{ }S) = \exists x \rhd S.x$$

$$\mathit{open}(\texttt{?}) = \exists x.x \qquad \mathit{capture}(S\texttt{<}A_1, \ldots, A_n\texttt{>}) = S\texttt{<}\mathit{open}(A_1), \ldots, \mathit{open}(A_n)\texttt{>}$$

$$\frac{\Delta \vdash S \rightsquigarrow T}{\Delta \vdash S\phi \rightsquigarrow T\phi}\ \textsc{Inherit-Subs} \qquad \frac{\Delta \vdash S \dot{\rightsquigarrow} T}{\Delta \vdash S <: T}\ \textsc{Subtype-Inherit} \qquad \frac{}{\vdash S \leq ?}\ \textsc{Cont-NB}$$

$$\frac{\Delta \vdash S <: T}{\Delta \vdash S \leq ?\ \texttt{extends}\ T}\ \textsc{Cont-UB} \qquad \frac{\Delta \vdash T <: S}{\Delta \vdash S \leq ?\ \texttt{super}\ T}\ \textsc{Cont-LB} \qquad \frac{\Delta \vdash \lceil S \rceil <: \lfloor T \rfloor}{\Delta \vdash S <: T}\ \textsc{Subtype-Bds}$$

$$\frac{\forall i : \Delta \vdash A_{1i} \leq A_{2i}}{\Delta \vdash S\langle A_{11}, \ldots, A_{1n}\rangle <: S\langle A_{21}, \ldots, A_{2n}\rangle}\ \textsc{Subtype-Args} \qquad \frac{\Delta \vdash \mathit{capture}(S) <: T}{\Delta \vdash S =: T}\ \textsc{Compat}$$

Figure 3: Rules for satisfying inheritance, subtyping, containment and compatibility

It will also be helpful to obtain the bounds of a captured type variable. For this, we let $\lceil S \rceil$ produce the upper bound of S, and $\lfloor S \rfloor$ produce its lower bound. For all S, the upper bound of $\exists x \triangleright S.x$ and $\exists x.x$ is always `java.lang.Object`, and the lower bound of $\exists x \triangleleft S.x$ and $\exists x.x$ is always $\bot$, the bottom type, which is the subtype of all types and only the supertype of itself.

2.3 Class Declarations

A class declaration consists of (1) a *base type* (the class being declared) $S\langle P_1, \ldots, P_n\rangle$ (P_1 to P_n are *type parameters*), (2) an inheritance rule $S\langle P_1, \ldots, P_n\rangle \rightsquigarrow T\langle A_1, \ldots, A_m\rangle$[1], (3) a set of attribute declarations mapping variable names to types, (4) a set of method declarations mapping method names to a set of method signatures (which may themselves be parametrically polymorphic), and (5) a set of modifiers on the class, which for our definition, can only contain `final`. This class declaration allows us to look up the types of its superclass, attributes, and methods. Suppose the class declaration of S contains the declaration of an attribute `a` of type T, and a variable `s` has type $S\phi_1$. Then the type of `s.a` would be $T\phi_2$ where $\mathit{expansion}(\mathit{capture}(S\phi_1)) = (S, \phi_2)$ (capture conversion is applied on `s` before looking up the type of `a`).

2.4 Relations

The relations between types relevant to our problem are mostly (1) equivalence $S \equiv T$, (2) inheritance $S \dot{\rightsquigarrow} T$ and (3) subtyping $S <: T$. While equivalence is trivial to define, inheritance and subtyping can only be adequately understood based on judgements from *class tables*, which are sets of class declarations. These judgements are in the form of $\Delta \vdash S \dot{\rightsquigarrow} T$, which in other words means that the class table Δ shows that $S \dot{\rightsquigarrow} T$ is true. The inductive definitions of these relations are shown in Figure 3.

Firstly, a constraint $S \rightsquigarrow T$ can be shown by Δ if there exists the inheritance rule $S \rightsquigarrow T$ in one of the class declarations in Δ. Then, the Inherit-Subs rule asserts that if $S \rightsquigarrow T$ is true, then $S\phi \rightsquigarrow T\phi$ is also true, for any substitution ϕ. From this definition, we can then let $\dot{\rightsquigarrow}$ be the reflexive and transitive closure of $\rightsquigarrow$. Lastly, $\dot{\rightsquigarrow}$ implies $<:$ as per the Subtype-Inherit rule. We also define $<:$ to be reflexive and transitive.

However, due to wildcard type arguments we require special considerations for subtyping. For this, the reflexive and transitive *containment* relation $S \leq T$ asserts that if we treat T as an interval with a lower and upper bound, then S is contained in T. The three rules Cont-NB, Cont-UB and Cont-LB describe the conditions that satisfy containment. Additionally, we include Subtype-Bds for subtyping with captured type variables. These rules allow us to define subtyping between two types that have the same raw type but different type arguments, shown in Subtype-Args. Finally, the

[1] In Java, a class declaration that does not have an `extends` clause will implicitly extend `java.lang.Object`, which corresponds to the $\top$ type.

Compat rule describes the *compatibility* constraint—if $S =: T$ then S is compatible in an assignment or strict/loose invocation context with T.

3 CONSTRAINT GENERATION

As described in Section 1, our algorithm involves constraint generation and constraint solving. In this section, we describe the constraint generation phase of our algorithm.

We have seen from previous sections that it is insufficient to only gather constraints from an incomplete program to complete it; additional information like the class table and generated types is also needed to make judgements on the satisfiability of constraints, and to support termination of our algorithm. We store all the required information as a *configuration* which our algorithm manipulates to arrive at a well-typed and complete program. The elements of a configuration $\langle \Delta, \Phi, \Omega, \Psi, \Theta \rangle$ are:

- Δ and Φ are class tables; the former contains the class declarations found in the program (which therefore our algorithm never mutates), and the latter contains newly created declarations of the classes that occur in the incomplete program but are not declared in it;
- Ω is the smallest set of constraints on types which must be true for the program to be well-typed and complete, such that if any constraint in Ω cannot be satisfied, then the program will be ill-typed;
- Ψ is the set of all types generated by our algorithm—this is needed to ensure that the types we generate do not exceed the maximum arity and height limits specified by the user (see Subsection 1.2);
- Θ is the set of all method invocations—this is needed to ensure that all method invocations have an unambiguous method declaration which is invoked.

The relationship between Δ, Φ, Ω, Ψ and Θ is that the goal is to populate Φ with class declarations such that (1) all constraints in Ω are satisfied; (2) all generated types in Ψ (which all belong to some declaration in Δ or Φ) do not exceed the maximum arity and height requirements of the program; and (3) all method invocations in Θ are unambiguous. The result is that the algorithm terminates, either saying that given the maximum arity and height requirements, no additional class declarations can allow the program to compile, or that the declarations in $\Delta \cup \Phi$ form a complete and well-typed program. With this in view, the first order of business of our algorithm is to obtain its starting configuration.

Populating Δ is relatively simple: each class declaration in the Abstract Syntax Tree (AST) of the input incomplete program will be added to Δ. We call a type that has a class declaration in Δ a *declared type*. In addition, every class type occurring in the incomplete program must also have a class declaration, which may be missing from Δ because the program is incomplete. Such types are what we call *missing types*. For all missing types $S\langle A_1, \ldots, A_n \rangle$, if its class declaration does not already exist in Φ, we create a new class declaration for it and add the new declaration to Φ. This declaration is empty, only describing the base type $S\langle P_1, \ldots, P_n \rangle$ where P_1 to P_n are new unique type parameters. Implicitly, the only inheritance rule in this class declaration is $S\langle P_1, \ldots, P_n \rangle \rightsquigarrow$ `java.lang.Object`. An example of this can be seen in Figure 4; in Program P_3 shown in Figure 4a, the types `A` and `C` are missing, so we add class declarations for them into Φ as shown in Figure 4b.

3.1 Dealing with Unknown Types

The types of the members in the declaration of a missing type are completely unknown since the declarations of missing types are not available in the program; for these we term *unknown*

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

(a) Incomplete program P_3 with missing attributes and methods

```
1 class A<V> {
2   \tau_1 x;
3   \tau_2 m(\tau_1{V -> C}) {
4     // typing ctx has T, U
5     // and {V -> C}
6   }
7 }
8 class \tau_1{V -> C} {
9   \delta_3 y;
10 }
11 class C { }
```

(b) Φ generated from P_3

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

(c) Incomplete program P_4 with ambiguous method declaration called

Figure 4: Example incomplete programs with missing attributes and methods.

types. For example, in line 9 of program P_1 in Figure 1a, the return type of `c.get()` is completely unknown because `C` is missing. To represent unknown types we make use of *inference variables*, which correspond to *pre-types* or *type variables* in several other works [MRdAP17, DGTS22]. This allows the algorithm to soundly generate typing constraints for unknown types. 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.

The first key insight behind our algorithm is that *the domain of any inference variable can be determined by observing its scope*. This comes from the fact that if a typing context only contains declarations of type parameters P_1 to P_n, then for all types occurring in the same context, the only type parameters they can contain are P_1 to P_n. In incomplete program P_3 in Figure 4a, the class `A` must have declarations of method `m` and attribute `x`. As the expressions `a.m(...)` and `a.x` appear in a typing context where the only type parameters declared are `T` and `U`, the only type parameters that the types of `a.x` and `a.m(...)` can contain are thus `T` and `U`.

We can make an even stronger claim about the *declared* type of `x` in class `A<V>`; as attributes do not receive type parameters unlike methods, we know that the only type parameter that the declared type of `x` can contain is `V`. These two claims do not contradict—let the declared type of `x` in `A` be τ_1, since τ_1 only contains `V`, then the type of `a.x` must be in the form $\tau_1\{\mathtt{V} \mapsto S\}$ where S is a type only containing type parameters `T` and `U`, so $\tau_1\{\mathtt{V} \mapsto S\}$ only contains type parameters `T` and `U`. Although we currently cannot make this same claim for the declared type of `y` because the enclosing type is completely unknown, once the declared type of `x` has been decided, this will once again give enough information about the typing context of the declared type of `y`. For example, if we let the declared type of `x` be `V`, then the type of `a.x` will now be `C`. We can now claim that the class `C` contains `y`, and its declared type does not contain type parameters since `C` does not have type parameters either.

With this in mind, we must be able to assign sound unknown types that encodes this restriction, and a semantics for soundly generating any valid type that adheres to the restriction. For this, we define three kinds of inference variables or unknown types: (1) *disjunctive types*, (2) *abstract types* and (3) *delayed placeholders*.

3.1.1 *Disjunctive Types.* A *disjunctive type* $\tau_i = \bigvee \mathbb{S}$ is an unknown type which we know must be equal to one of the types in $\mathbb{S}$. We write a disjunctive type as $\tau_i = \bigvee \mathbb{S}$ for some i where i allows us to distinguish between different disjunctive types. Finally, we write τ to denote "some disjunctive type". For example, we write $\tau_2 = \bigvee \{\mathtt{T}, \mathtt{U}\}$ to say that τ_2 is either `T` or `U`.

3.1.2 *Abstract Types.* In this paper, abstract types do not refer to abstract classes or interfaces that cannot be instantiated. Instead, an *abstract type* $\alpha_i\texttt{<...>}^{C}_{\mathbb{S}}$ is an unknown type which repre-

sents 'some' potentially parametrically polymorphic class type containing type arguments that are (recursively) unknown types. It is constrained by the constraints in the *constraint store* C and its arguments can only contain the type parameters in $\mathbb{S}$. We write α<...> to denote "some abstract type", and write α_i as the erasure of $\alpha_i<...>^C_{\mathbb{S}}$ (i.e it is $\epsilon(\alpha_i<...>^C_{\mathbb{S}})$).

Abstract types are meant to be *concretized* to an instance of one of the class types appearing in the incomplete program, or one which is newly created by our algorithm. The *concretization* of an abstract type $\alpha_i<...>^C_{\mathbb{S}}$ into an instance of a class type U is the raw type of U, along with arguments which are disjunctive types, each of which could be one of the type parameters in $\mathbb{S}$, or (recursively) fresh abstract types of the empty constraint store and the same type parameter set $\mathbb{S}$:

$$
\begin{aligned}
\mathit{concretize}(\alpha_i<...>^C_{\mathbb{S}}, S<A_1,\ldots,A_n>) &= S<\tau_1,\ldots,\tau_n> \\
\text{where } \mathbb{S}_j &= \mathbb{S} \cup \{\alpha_j<...>^{\emptyset}_{\mathbb{S}}\} \\
\mathbb{S}^*_j &= \mathbb{S}_j \cup \{\texttt{? extends } S \mid S \in \mathbb{S}_j\} \cup \{\texttt{? super } S \mid S \in \mathbb{S}_j\} \cup \{\texttt{?}\} \\
\tau_j &= \bigvee \mathbb{S}^*_j \\
\tau_j, \mathbb{S}_j, \alpha_j<...>^{\emptyset}_{\mathbb{S}} &\text{ fresh for each } j \in [1,n]
\end{aligned}
$$

For example, concretizing an abstract type $\alpha<...>^C_{\{\texttt{T},\texttt{U}\}}$ to an instance of `List<E>` gives us $\mathit{concretize}(\alpha<...>^C_{\{\texttt{T},\texttt{U}\}}, \texttt{List<E>}) = \texttt{List<}\tau_1\texttt{>}$ where τ_1 is either `T`, `U`, $\alpha_1<...>^{\emptyset}_{\{\texttt{T},\texttt{U}\}}$, `? extends T`, `? extends U`, `? extends` $\alpha_1<...>^{\emptyset}_{\{\texttt{T},\texttt{U}\}}$, `? super T`, `? super U`, `? super` $\alpha_1<...>^{\emptyset}_{\{\texttt{T},\texttt{U}\}}$ or `?`, and τ_1 and $\alpha_1<...>^{\emptyset}_{\{\texttt{T},\texttt{U}\}}$ are fresh inference variables.

3.1.3 *Delayed Placeholders.* A delayed placeholder ${\delta_i}^C$ is an inference variable used as a placeholder type where nothing meaningful is known about it. This is most commonly ascribed to the types of attributes where the containing type is itself unknown, for example, the declared type of `y` in the declaration of the type of `a.x` in program P_3 in Figure 4a. We write a delayed placeholder as ${\delta_i}^C$ for some i and some constraint store C.

3.1.4 *Substitutions and Replacements of Unknown Types.* Nothing meaningful happens during a substitution on an unknown type. However, as described earlier, unknown types are meant to be replaced with other types. Therefore, we define a *replacement* to be a function $\chi = \{U_1 \mapsto A_1, \ldots, U_n \mapsto A_n\}$ to replace all occurrences of unknown types U_1 to U_n with types A_1 to A_n respectively. Importantly, if an unknown type U has substitutions and replacements, such as $U\phi_1\phi_2\ldots\phi_n\chi$, the replacements are done first. For example, suppose we have $\phi_1 = \{\texttt{T} \mapsto \texttt{List<U>}\}$, $\phi_2 = \{\texttt{U} \mapsto \texttt{Integer}\}$, $\chi = \{\tau \mapsto \texttt{T}\}$ and $\tau = \bigvee\{\texttt{T},\texttt{U}\}$, then $\tau\phi_1\phi_2\chi \equiv (\tau\chi)\phi_1\phi_2 \equiv \texttt{T}\phi_1\phi_2 \equiv \texttt{List<U>}\phi_2 \equiv \texttt{List<U}\phi_2\texttt{>} \equiv \texttt{List<Integer>}$.

There are three properties of valid replacements on types. (1) The domain of a replacement can only be unknown types, (2) for any $\tau = \bigvee \mathbb{S}$ a valid replacement χ will only replace τ with some $S \in \mathbb{S}$, and (3) for any types α<...> and S a valid replacement χ will only replace α<...> with $\mathit{concretize}(\alpha<...>, S)$.

3.1.5 *Unknown Types With Known Typing Contexts.* As described earlier, if in a typing context there are only the declarations of type parameters P_1 to P_n, then all types occurring in this same typing context must not contain any other type parameters other than P_1 to P_n. Given attributes and methods where their types are unknown, our goal is to assign inference variables such that a sequence of valid replacements allow them to form any valid Java type, subject to this restriction.

For this, we define $\mathbb{T}_n(\mathbb{S})$ where $\mathbb{S}$ is a set of type parameters, and $\mathbb{T}_n(\mathbb{S})$ is the set of all valid Java types S where if S contains a type parameter P, then $P \in \mathbb{S}$. We can inductively construct

$\mathbb{T}_n(\mathbb{S})$ with the following rules, assuming (without loss of generality) all class types have arity at most 1: (1) if $P \in \mathbb{S}$ then $P \in \mathbb{T}_1(\mathbb{S})$, (2) if S is a class type with arity 0 then $S \in \mathbb{T}_1(\mathbb{S})$, (3) if S is a class type with arity 1 then $S\texttt{<?>} \in \mathbb{T}_2(\mathbb{S})$, (4) if $A \in \mathbb{T}_{n-1}(\mathbb{S})$ and S is a class type with arity 1 then $S\texttt{<}A\texttt{>}, S\texttt{<? extends }A\texttt{>}, S\texttt{<? super }A\texttt{>} \in \mathbb{T}_n(\mathbb{S})$.

Suppose $\tau = \bigvee\left(\mathbb{S} \cup \{\alpha\texttt{<...>}^{C}_{\mathbb{S}}\}\right)$ where $\mathbb{S}$ are type parameters, and let $\chi_1 \ldots \chi_m$ be a sequence of valid replacements on τ, so that $\tau\chi_1 \ldots \chi_m = S$ where S does not contain unknown types. Theorem 3.1 shows S will *always be a valid Java type that only contains type parameters in* $\mathbb{S}$, while Theorem 3.2 shows that for *all* valid Java types S that only contain type parameters in $\mathbb{S}$, the sequence $\chi_1 \ldots \chi_m$ *always exists*. These theorems are proven in Appendix A.

Theorem 3.1. *For all* $\tau = \bigvee\left(\mathbb{S} \cup \{\alpha\texttt{<...>}^{C}_{\mathbb{S}}\}\right)$ *where* $\mathbb{S}$ *are type parameters, for any sequence of valid replacements* χ_1 *to* χ_m*, if* $\tau\chi_1 \ldots \chi_m$ *has height* n *and does not contain unknown types,* $\tau\chi_1 \ldots \chi_m \in \mathbb{T}_n(\mathbb{S})$.

Theorem 3.2. *For all* $\tau = \bigvee\left(\mathbb{S} \cup \{\alpha\texttt{<...>}^{C}_{\mathbb{S}}\}\right)$ *where* $\mathbb{S}$ *are type parameters, for all* $S \in \mathbb{T}_n(\mathbb{S})$*, there exists a sequence of valid replacements* χ_1 *to* χ_m *such that* $\tau\chi_1 \ldots \chi_m = S$.

This is great news because we have constructed an unknown type that allows us to soundly generate any Java type we want, restricted to only contain type parameters declared in any given typing context. Therefore, to assign unknown types to declarations and expressions that occur in a typing context declaring type parameters $\mathbb{S}$, the domain of all possible types it can take can be encoded by $\tau = \bigvee\left(\mathbb{S} \cup \{\alpha\texttt{<...>}^{C}_{\mathbb{S}}\}\right)$. We thus let $\tau_i(\mathbb{S})$ be shorthand for a globally fresh $\tau_i = \bigvee\left(\mathbb{S} \cup \{\alpha_i\texttt{<...>}^{\emptyset}_{\mathbb{S}}\}\right)$.

3.2 Types of Missing Attributes

Obtaining the type of an attribute expression `e.i` is straightforward if the type of `e` is S and S is *fully-declared*, i.e. the class declaration of S is in the incomplete program P (and in Δ), and its supertype is also fully-declared. If the declaration of `i` is not found then we may terminate with an error because no possible additions to the incomplete program will ever complete the program, and if it is then we may proceed as per usual. Otherwise, if the type of `e.i` cannot be found, it must mean that S is a missing or unknown type, or is a declared type with a missing ancestor and all of the declared ancestors do not declare `i`.

If S is missing or has a missing superclass and the declaration of attribute `i` is not found, we let $T\texttt{<}P_1, \ldots, P_n\texttt{>}$ be the missing superclass of S (or $T\texttt{<}P_1, \ldots, P_n\texttt{>} = S$ itself if it is missing), and add the attribute declaration $(\texttt{i} \mapsto \tau_i(\{P_1, \ldots, P_n\}))$ into the declaration of class $T\texttt{<}P_1, \ldots, P_n\texttt{>}$.

Finally, if S is an unknown type $U\phi_1 \ldots \phi_n$ and there isn't a declaration in Φ where $U\phi_1 \ldots \phi_n$ is its base type, we create a new class declaration for it in Φ. Then, if `i` is not yet an attribute of $U\phi_1 \ldots \phi_n$ in its class declaration, we add the attribute declaration $(\texttt{i} \mapsto {\delta_i}^{\emptyset})$ in its class declaration, where ${\delta_i}^{\emptyset}$ is fresh. All generated unknown types will be added to Ψ.

Figure 4 shows an example of how the types of missing attributes are resolved. From lines 4 and 5 in P_3 (Figure 4a), the type of `a` has an attribute `x`, and the type of `a.x` has attribute `y`. We know the type of `a` is `A<C>`, so we insert a new attribute declaration for `x` in the class declaration of `A<V>`. Since we know that the declared type of `x` can only contain the type parameter `V`, we let the declared type of `x` be $\tau_1 = \bigvee\{\texttt{V}, \alpha_1\texttt{<...>}^{\emptyset}_{\{\texttt{V}\}}\}$, and thus the type of `a.x` will be $\tau_1\{\texttt{V} \mapsto \texttt{C}\}$. Subsequently, since there is no class declaration for $\tau_1\{\texttt{V} \mapsto \texttt{C}\}$, we create one, so we can add the declaration of attribute `y` of type ${\delta_3}^{\emptyset}$ to it. All inference variables are globally fresh. The result of the creation of these attribute declarations is shown in the corresponding Φ (expressed as Java code) in Figure 4b.

3.3 Types of Missing Methods

Just like our treatment of missing attributes, if given a method invocation expression, a corresponding method declaration cannot be made to exist because the type of the target is fully-declared and the method declaration is not found, we can fail immediately citing an error. However, unlike attributes, methods must be treated differently because they can be overloaded, overridden, and can receive type parameters. For example, the call to `m` in line 4 of P_4 in Figure 4c is ambiguous, because if we assume that the type of `b.a` is compatible in an invocation context with `String`, then the method `m` defined in line 6 will be called, and the method call will evaluate to an `int`. However, `b.a` could be of some other type which is compatible in an invocation context with the formal parameter of some other overloaded method `m` defined in class `C`, which `A` inherits. Also, as methods themselves can receive type parameters, it is not immediately clear at this stage what type parameters can be contained in the formal parameter and return types of a new method declaration. Due to these additional considerations, instead of immediately adding method declarations to missing types, we will only add method *invocations* to the class declarations of missing/unknown types, then assign corresponding method *declarations* that support these invocations later.

We make the distinction between method invocations and declarations clear by way of an example. Suppose we have class `B<T>` containing the variable declarations `T x;` and `A a;`, and the method invocation expression `x = a.id(x)`. Furthermore, we have the class declaration `A` containing only the method declaration `<V> V id(V v) { return v; }`. In this case, the *invocation* maps the *argument* type (the type of `x` which is `T`) to the type of the method call expression `a.id(x)` (which is also `T`) giving us $\mathtt{T} \longrightarrow \mathtt{T}$, while the *declaration* maps the *formal parameter* type `V` to the return type `V` giving us $\mathtt{V} \longrightarrow \mathtt{V}$. We temporarily infer the types of method invocations first because inference of other types in the incomplete program may depend on them; reconstructing method declarations can be done later: given method invocation $A \longrightarrow O$ we can infer a method declaration $P \longrightarrow R$ where $A =: P\phi$ and $R\phi =: O$ for some ϕ determined by Java's type inference algorithm [GJSB05] under A.

Now suppose a method `m` is called with some arguments $S_1, S_2, \ldots, S_n$ in a typing context containing the type parameters $P_1, \ldots, P_x$. To deal with the ambiguity of the potential other method declarations being invoked, we first assign a fresh placeholder δ_r as the type of the method invocation expression, then create a disjunctive constraint to account for the different possible method declarations being invoked. Next, we partition the type of the target and its superclasses into two sets, $\mathbb{D}$, the declared types, and $\mathbb{M}$, the missing/unknown types (there can be at most one type in $\mathbb{M}$). Then, let $M_{\mathbb{D}}$ be the set of methods declared by the types in $\mathbb{D}$ that have identifier `m` and arity n. We assert that either

- there exists one method declaration in $M_{\mathbb{D}}$ such that the arguments $S_1, S_2, \ldots, S_n$ are compatible with its parameters, and its return type is equivalent to δ_r, or
- the type in $\mathbb{M}$ (if it exists) has a method invocation $\mathtt{m}(S_1, S_2, \ldots, S_n)$ that returns $\delta_r \equiv \tau_r(\{P_1, \ldots, P_x\})$; this will be represented as a $\mathit{hasMethodInv}(T, s, \mathbb{S})$ constraint, stating that target type T has method invocation s and the set of type parameters the method invocation has access to in its typing context is $\mathbb{S}$.

Formally, if we let $\mathit{param}(m, [S_1, \ldots, S_n], i)$ be the type of the i^{th} formal parameter of method declaration m under the inferred typing context based on arguments $[S_1, \ldots, S_n]$ (by Java's type inference algorithm), and $\mathit{ret}(m, [S_1, \ldots, S_n])$ be the corresponding return type, and if $\mathbb{S}$ is the set of type parameters the method invocation has access to, the generated constraint that will be added to Ω

would be:

$$\bigvee_{M \in M_{\mathbb{D}}} \left(\bigwedge_{i=1}^{n} (S_i =: param(M, [S_1, \ldots, S_n], i) \wedge \delta_r \equiv ret(M, [S_1, \ldots, S_n])) \right) \vee \bigvee_{M \in \mathbb{M}} (hasMethodInv(M, \mathtt{m} : [S_1, \ldots, S_n] \rightarrow \tau_r, \mathbb{S}) \wedge \delta_r \equiv \tau_r)$$

For example, in line 5 of incomplete program P_3 in Figure 4a there is a method invocation `a.m(a.x)`. The argument (as inferred earlier) has type $\tau_1\{\mathtt{V} \mapsto \mathtt{C}\}$, and the expression is in a typing context containing type parameters `T` and `U`. We first assign a placeholder δ_4 as the type of `a.m(a.x)`, then partition the type of the target (the type of `a` which is `A<C>`) into two sets, $\mathbb{D} = \{\texttt{java.lang.Object}\}$ and $\mathbb{M} = \{\texttt{A<C>}\}$. The set $M_{\mathbb{D}}$ of method declarations is empty because `java.lang.Object` does not have any methods named `m`. Therefore, the constraint generated is $hasMethodInv\left(\texttt{A<C>}, \mathtt{m} : \tau_1\{\mathtt{V} \mapsto \mathtt{C}\} \rightarrow \tau_2 = \bigvee\{\mathtt{T}, \mathtt{U}, \alpha_2\texttt{<...>}^{\emptyset}_{\{\mathtt{T},\mathtt{U}\}}\}, \{\mathtt{T}, \mathtt{U}\}\right) \wedge \delta_4 \equiv \tau_2$. For visualization purposes we immediately add this method invocation into the class declaration of `A<V>` in Φ and replace δ_4 with τ_2, as shown in Figure 4b.

3.4 Constraints On Relations Between Types

Now that the type of all expressions can be determined, our algorithm can populate Ω with the constraints on relations between types by analyzing the AST of the incomplete program. In our model, only class types exist, so the only constraint relevant to this step is the compatibility constraint $S =: T$. Other constraints that need to be generated for the completion of Java programs have been described by earlier works [DH08] and the Java Language Specification [GJSB05]; these are all generated by JavaCIP too. There are only two statements that could generate constraints on relations between types: assignments statements `x = y` generates $typeof(\mathtt{x}, \Gamma) := typeof(\mathtt{y}, \Gamma)$, and return statements `return z` in a method that returns R generates $R := typeof(\mathtt{z}, \Gamma)$, where Γ denotes the familiar typing environment. For example, the constraints generated by P_3 in Figure 4a are `A<C>` := `A<C>` from line 3, `java.lang.Object` := $\delta_3{}^{\emptyset}$ from line 4, and `java.lang.Object` := τ_2 from line 5.

By this point, our configuration $\langle \Delta, \Phi, \Omega, \Psi, \Theta \rangle$ would have been populated successfully, or failed with an error (due to unresolvable references to missing attributes or methods). To summarize, Δ was populated with class declarations in the incomplete program, Φ was built by adding class declarations of missing and unknown types, including new attribute declarations if they are required, Ω was populated with the constraints generated from resolving method invocations and the constraints on relations between types, Ψ contains all the inference variables generated, and Θ contains all the method invocations in the incomplete program.

4 CONSTRAINT SOLVING

Once the starting configuration has been obtained, as described in earlier sections, we need to satisfy all the constraints in Ω, so that our incomplete program is well-typed. The underlying strategy in doing so is to make necessary modifications to Φ until all constraints in Ω are satisfied. Our algorithm performs four stages in the constraint solving phase:

1. Resolve. We *reduce* constraints in Ω to solve them, either by verifying they are true, failing if they are false, or making them true by adding/amending the declarations in Φ. In this step, we relax the constraints on well-formed class tables that 1) $S \rightsquigarrow T\texttt{<}A_{11}, \ldots, A_{1n}\texttt{>}$ and $S \dot{\rightsquigarrow} T\texttt{<}A_{21}, \ldots, A_{2n}\texttt{>}$ implies $A_{1i} \equiv A_{2i}$ for $1 \leq i \leq n$, and 2) a class can only extend one other class;

2. Deconflict. We re-introduce the two constraints which were relaxed in the Resolve step, then resolve conflicts in Φ that arise from such, or fail;

3. Concretize. We concretize abstract types in Ψ into class types, or fail;

4. DeclareMethods. We assign each method invocation in Φ (obtained from Subsection 3.3) to a method declaration, or create one if it doesn't exist, and ensure that all method invocations in Θ are unambiguous, or fail;

Each of these stages receive a configuration as input, and may produce more than one configuration because there are choices to be made. For example, some of the constraints in Ω could be in the form of disjunctions, such as those produced from dealing with ambiguous method invocations; in such a case we produce one configuration for each constraint in the disjunction, and further continue with constraint solving for each of them. In addition, each of the stages may produce more constraints, which once again need to be resolved. As such, each of the four stages receive a configuration as input, and as output, returns either a `Left[`y`]` where y is a set of configurations (indicating that more constraint solving needs to be done) or `Right[`x`]` where x is a configuration that has completed the stage successfully.

This process is outlined by the ConstSolve and ConstSolveImpl algorithm, shown in Algorithm 1, which explores all possible configurations which may lead to a successful complete program. In Line 4, as a base case we return the failure configuration `fail` if there are no configurations to solve, indicating that the incomplete program cannot be completed. In Line 5 we choose any arbitrary configuration in the input set, and in line 6 we pass it through the four stages which we elaborate on later. »= is the `Right`-biased monadic bind operator, meaning that we only operate on the result of the previous stage if it is a `Right[`x`]` where x is a configuration; if the result of the previous stage is a `Left[`y`]` where y is a set of configurations, each stage acts as the identity function. From lines 7 through 9, if passing a configuration through the four stages results in a `Right`, then it means that the resulting configuration represents a complete and well-typed program and can be returned immediately. Otherwise, more constraint solving needs to be done on the resulting configuration(s) and the remaining configurations and therefore a recursive call to ConstSolveImpl is made.

Algorithm 1 Constraint Solving

Input: A configuration $\langle \Delta, \Phi, \Omega, \Psi, \Theta \rangle$
Output: Either a configuration $\langle \Delta^*, \Phi^*, \emptyset, \Psi^*, \Theta^* \rangle$ where the declarations in $\Delta^* \cup \Phi^*$ form a complete and well-typed program, or `fail`

```
1: function ConstSolve(configuration)
2:     return ConstSolveImpl({configuration})
3: function ConstSolveImpl(configurations)
4:     if configurations = ∅ then return fail
5:     (c, remaining) ← (a, configurations\{a}) for some a ∈ configurations
6:     res ← Resolve(c) »= Deconflict »= Concretize »= DeclareMethods
7:     return match res with
8:         case Right[x]: x                                  ▷ x is a configuration
9:         case Left[y]: ConstSolveImpl(y ∪ remaining)        ▷ y is a set of configurations
```

4.1 Resolving Constraints

In this section we describe the Resolve stage, which performs the initial pass of constraint solving. So far, the main constraints we have generated are *hasMethodInv* constraints, and *compatibility*

$$\frac{}{\vdash S\texttt{<}A_{11},\ldots,A_{1n}\texttt{>} <: S\texttt{<}A_{21},\ldots,A_{2n}\texttt{>} \Rightarrow \bigwedge_{i\in[1,n]}\{A_{1i} \leq A_{2i}\}}\ \textsc{Red-Args}$$

$$\frac{\Delta,\Phi \vdash S\texttt{<}A_1,\ldots,A_n\texttt{>} \dot{\rightsquigarrow} T\texttt{<}B_1,\ldots,B_m\texttt{>} \qquad S \neq T}{\Delta,\Phi \vdash S\texttt{<}A_1,\ldots,A_n\texttt{>} <: T\texttt{<}C_1,\ldots,C_m\texttt{>} \Rightarrow T\texttt{<}B_1,\ldots,B_m\texttt{>} <: T\texttt{<}C_1,\ldots,C_m\texttt{>}}\ \textsc{Red-Upcast}$$

$$\frac{\mathbf{A} \in \Omega \qquad \mathbf{A} = \mathbf{A}_1 \vee \cdots \vee \mathbf{A}_n, \text{where } n > 1}{\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle \xrightarrow{\texttt{Left}} \{\langle \Delta,\Phi,(\Omega\backslash\{\mathbf{A}\}) \cup \{\mathbf{A}_i\},\Psi,\Theta\rangle \mid i \in [1,n]\}}\ \textsc{Disj-Sel}$$

$$\frac{}{\langle \Delta,\Phi,\emptyset,\Psi,\Theta\rangle \xrightarrow{\texttt{Right}} \langle \Delta,\Phi,\emptyset,\Psi,\Theta\rangle}\ \textsc{Resolve-Complete}$$

Figure 5: Some constraint reduction and configuration transition rules

constraints. The former is easy to solve—we simply add the method invocation to the class declaration of the target type (similar to how `m` was added to `A<V>` in Figure 4b)—resolution of the latter comes from our second key insight: based on the set of inference rules for inductively satisfying various constraints in Figure 3, if a constraint $\mathbf{A}$ in Ω appears in the consequent of a rule, replacing it with the antecedent preserves satisfiability. In other words, if $\mathbf{B}$ implies $\mathbf{A}$ then we say $\mathbf{A}$ *reduces* to $\mathbf{B}$, so we can replace $\mathbf{A}$ with $\mathbf{B}$ in Ω. Reducing constraints is routine in type checking and type inference in Java [GJSB05] and is similar to term reduction in unification [MM82]; this gives us a sound framework for solving constraints step-by-step, and most importantly, *if any constraint $\mathbf{A}$ is irreducible because it is missing an antecedent $\mathbf{B}$, we amend Φ so that $\mathbf{B}$ holds, which allows us to reduce $\mathbf{A}$ further.*

We first define constraint reduction. Letting uppercase boldface letters like $\mathbf{A}$ and $\mathbf{B}$ be meta-variables over constraints, we write $\mathbf{A} \Rightarrow \mathbf{B}$ to say that $\mathbf{A}$ *reduces* to $\mathbf{B}$ or $\mathbf{B}$ *implies* $\mathbf{A}$. If by the rules in Figure 3 $\mathbf{A}$ is satisfied, $\mathbf{A} \Rightarrow$ True, and if $\mathbf{A}$ is falsified then $\mathbf{A} \Rightarrow$ False. The rules for constraint reduction can all be derived and proven correct from the rules in Figure 3. We list two of the most important rules in Figure 5, and show proof of the Red-Upcast rule as an example:

Theorem 4.1. *If* $S\texttt{<}A_1,\ldots,A_n\texttt{>} \dot{\rightsquigarrow} T\texttt{<}B_1,\ldots,B_m\texttt{>}$ *where* $S \neq T$*, then* $S\texttt{<}A_1,\ldots,A_n\texttt{>} <: T\texttt{<}C_1,\ldots,C_m\texttt{>} \Rightarrow T\texttt{<}B_1,\ldots,B_m\texttt{>} <: T\texttt{<}C_1,\ldots,C_m\texttt{>}$.

Proof. By Subtype-Inherit in Figure 3, $S\texttt{<}A_1,\ldots,A_n\texttt{>} <: T\texttt{<}B_1,\ldots,B_m\texttt{>}$. If $T\texttt{<}B_1,\ldots,B_m\texttt{>} <: T\texttt{<}C_1,\ldots,C_m\texttt{>}$ then we have $S\texttt{<}A_1,\ldots,A_n\texttt{>} <: T\texttt{<}C_1,\ldots,C_m\texttt{>}$ by transitivity of $<:$. □

Next, to describe the Resolve stage, we define *configuration transitions*. A *configuration transition* describes the input/output pair of a given stage. We write $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle \rightarrow x$ to say that the configuration $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle$ *transitions* to x, i.e. Resolve returns x on input $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle$. Resolve either returns a `Left` of a set of configurations (meaning more resolution needs to be done), or a `Right` of a single configuration (meaning that resolution has completed. To denote the former we write $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle \xrightarrow{\texttt{Left}} x$ to mean $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle \rightarrow \texttt{Left[}x\texttt{]}$, and for the latter write $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle \xrightarrow{\texttt{Right}} x$ to mean $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle \rightarrow \texttt{Right[}x\texttt{]}$. The transition rules for Resolve are relatively straightforward: if a constraint $\mathbf{A}$ in Ω reduces to $\mathbf{B}$, then $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle \xrightarrow{\texttt{Left}} \langle \Delta,\Phi,\Omega\backslash\{\mathbf{A}\} \cup \{\mathbf{B}\},\Psi,\Theta\rangle$; if a constraint $\mathbf{A}$ in Ω is satisfied then $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle \xrightarrow{\texttt{Left}} \langle \Delta,\Phi,\Omega\backslash\{\mathbf{A}\},\Psi,\Theta\rangle$, and if it is falsified then $\langle \Delta,\Phi,\Omega,\Psi,\Theta\rangle \xrightarrow{\texttt{Left}} \emptyset$. We also show two important transition rules in Figure 5—the Disj-Sel rule describes how given a disjunctive constraint in Ω, Resolve branches into one configuration for each constituent configuration in the disjunction, and the Resolve-Complete rule which states that the Resolve stage is complete when Ω is empty.

Chiefly concerning this stage are constraints that cannot be reduced because not enough information is present in Δ and Φ. There are three instances where this is the case, all concerning subtype constraints $S <: T$. Firstly, if either S or T is some abstract type $\alpha\texttt{<...>}^{C}_{\mathbb{S}}$ or placeholder δ^{C}, we add the constraints to their respective constraint stores C and remove them from Ω for later resolution at the Concretize stage. Next, if either S or T is a disjunctive type $\tau = \bigvee \mathbb{S}$, similar to Disj-Sel in Figure 5, for each $S \in \mathbb{S}$ we transition $\langle \Delta, \Phi, \Omega, \Psi, \Theta \rangle$ into one configuration where τ is replaced with S. The third case is when S is missing, T is not an inference variable, and there is no way to prove or disprove $S <: T$ due to missing inheritance rules, in which case we allow S to greedily extend some instance of T in its class declaration (therefore the class declaration of S may contain more than one inheritance rule). Formally, suppose $S = S_R\texttt{<}A_1, \ldots, A_n\texttt{>}$ and $T = T_R\texttt{<}B_1, \ldots, B_m\texttt{>}$ and the base type of the class declaration of S is $S_R\texttt{<}P_1, \ldots, P_n\texttt{>}$. Then, Resolve allows $S_R\texttt{<}P_1, \ldots, P_n\texttt{>}$ to greedily extend some instance of T_R by adding the inheritance rule $S_R\texttt{<}P_1, \ldots, P_n\texttt{>} \rightsquigarrow T_R\texttt{<}\tau_1(\{P_1, \ldots, P_n\}), \ldots, \tau_n(\{P_1, \ldots, P_n\})\texttt{>}$ to the class declaration of S. This allows the original constraint to be reduced to $T_R\texttt{<}\tau_1, \ldots, \tau_n\texttt{>}\{P_1 \mapsto A_1, \ldots, P_n \mapsto A_n\} <: T_R\texttt{<}B_1, \ldots, B_m\texttt{>}$ by the Red-Upcast rule in Figure 5.

4.2 Removing Conflicts in Class Tables

In Resolve, we relaxed the constraints on well-formed class tables that (1) $S \dot{\rightsquigarrow} T\texttt{<}A_{11}, \ldots, A_{1n}\texttt{>}$ and $S \dot{\rightsquigarrow} T\texttt{<}A_{21}, \ldots, A_{2n}\texttt{>}$ implies $A_{1i} \equiv A_{2i}$ for $1 \leq i \leq n$, and (2) a class can only extend one other class; these have been violated due greedy extension. In the Deconflict stage we re-introduce these constraints so that the class tables in Δ and Φ together are well-formed.

Deconflict checks if such conflicts exist, and if they do, it changes Φ and/or introduces new constraints to be solved by Resolve again. For the first constraint, if in the configuration $\langle \Delta, \Phi, \emptyset, \Psi, \Theta \rangle$ there exists types S, $T\texttt{<}A_{11}, \ldots, A_{1n}\texttt{>}$ and $T\texttt{<}A_{21}, \ldots, A_{2n}\texttt{>}$ that satisfy constraint (1) but it cannot be shown that $A_{1i} \equiv A_{2i}$ for $1 \leq i \leq n$, then $\langle \Delta, \Phi, \emptyset, \Psi, \Theta \rangle \xrightarrow{\texttt{Left}} \{\langle \Delta, \Phi, \bigwedge_{i \in [1,n]} \{A_{1i} \equiv A_{2i}\}, \Psi, \Theta \rangle\}$. For the second constraint, if there exists class S that has two (without loss of generality) inheritance rules $S \rightsquigarrow T$ and $S \rightsquigarrow U$, Deconflict transitions the configuration to two configurations, one where $S \rightsquigarrow U$ is removed and the constraint $T <: U$ is introduced into Ω, the other $S \rightsquigarrow T$ is removed and the constraint $U <: T$ is introduced. The introduction of the subtype constraint forces Resolve to preserve satisfiability of $\dot{\rightsquigarrow}$.

4.3 Concretizing Abstract Types

In Resolve, when encountering subtype constraints on abstract types $\alpha\texttt{<...>}\phi_1 \ldots \phi_n$, we removed them from Ω and added the constraint to their constraint stores. The reason for this comes from the third key insight of our algorithm: intuitively, each abstract type represents 'some class type'. This could mean that it is an instance of one of the many other classes in the program, or a brand new class which must be created, making the domain of possible types $\alpha\texttt{<...>}\phi_1 \ldots \phi_n$ can be, infinite (putting aside the maximum arity and height constraints). For example, if we suppose that a particular $\alpha_i\texttt{<...>}_{\{\texttt{T}\}}\{\texttt{T} \mapsto \texttt{A}\}$ could only be an instance of `class A` or `class Box<T>`, then $\alpha_i\texttt{<...>}_{\{\texttt{T}\}}\{\texttt{T} \mapsto \texttt{A}\}$ could be `A`, `Box<A>`, `Box<Box<A>>`, and so on. However, if we are able to conclusively rule out the possibility that $\alpha_i\texttt{<...>}$ is an instance of `Box`, then we immediately know $\alpha_i\texttt{<...>} \equiv \texttt{A}$. This process should not require knowledge of the type arguments of $\alpha_i\texttt{<...>}$, that is because the arguments (more fundamentally, the arity) of $\alpha_i\texttt{<...>}$ is not known. Thus, *our algorithm eliminates possible concretizations of abstract types by exploiting rules of well-formed type hierarchies*, which only requires analysis of the *erasures* of types (the types without their arguments).

We first obtain the combined constraint store C^*, the union of the constraint stores of all abstract types. Then define a new relation between types $\rightarrowtail$ given by: if $\Delta, \Phi \vdash S \rightsquigarrow T$ then $\epsilon(S) \rightarrowtail \epsilon(T)$,

and if $C^* \vdash \alpha_i\texttt{<...>} \texttt{<:}\ S$ or $C^* \vdash T \texttt{<:}\ \alpha_j\texttt{<...>}$ then $\epsilon(\alpha_i\texttt{<...>}) \rightarrowtail \epsilon(S)$ and $\epsilon(T) \rightarrowtail \epsilon(\alpha_j\texttt{<...>})$ respectively. This gives us an *erasure graph* as a directed graph $\mathcal{G}$ consisting of the erasure of all (concrete) class types and abstract types in Δ, Φ and Ψ as vertices, and there is an edge from S to T if and only if $S \rightarrowtail T$. We can then restrict the possible concretizations of some α based on the following properties that $\mathcal{G}$ must have.

4.3.1 *Cycles.* Cyclic inheritance is prohibited in a well-formed Java program [GJSB05]. Considering this, we define $\mathbb{E}(S)$ to be the set of all types appearing in a cycle with S in $\mathcal{G}$. We can further partition this set to $\mathbb{E}_c(S)$ the concrete (non-abstract) class types and $\mathbb{E}_a(S)$ the abstract types. Then, Theorem 4.2 shows one case where we can determine the exact type a particular abstract type must be concretized into (Theorem 4.2 is proven in Appendix B).

Theorem 4.2. *For any α_i, if $\mathbb{E}_c(\alpha_i) = \{S\}$, then all types in $\mathbb{E}_a(\alpha_i)$ must all be concretized to instances of S.*

We can simultaneously show what a particular α_i must *not* be instances of. We write $\mathbf{Path}_{\mathcal{G}}(S, T)$ as the sentence 'there exists a path from S to T in $\mathcal{G}$', and define the *lower field* of α_i as $\mathbb{L}(\alpha_i)$ to be $\{S \in \mathcal{G} \mid \mathbf{Path}_{\mathcal{G}}(S, \alpha_i) \wedge \neg\exists T \in \mathcal{G} : \mathbf{Path}_{\mathcal{G}}(S, T) \wedge \mathbf{Path}_{\mathcal{G}}(T, \alpha_i)\}$ where S and T are non-abstract.

Theorem 4.3. *For non-abstract S, if $\exists T \in \mathbb{L}(\alpha_i) : \boldsymbol{Path}_{\mathcal{G}}(S, T)$ where $S \neq T$, then α_i must not be concretized into an instance of S.*

Proof. By definition, $\mathbf{Path}_{\mathcal{G}}(T, \alpha_i)$. If we concretize α_i into an instance of S we will also have $\mathbf{Path}_{\mathcal{G}}(T, S)$. Since $\mathbf{Path}_{\mathcal{G}}(S, T)$, this concretization causes cyclic inheritance. □

We can dually define the upper field and prove the dual of Theorem 4.3.

4.3.2 *Final Types.* A final class cannot be extended by any other class. Therefore, if there exists α_i where $\exists S \in \mathcal{G} : \mathbf{Path}_{\mathcal{G}}(\alpha_i, S)$ and S is `final`, then α_i must be concretized to an instance of S.

4.3.3 *Fully-Declared Subtypes.* When given an incomplete program P, our algorithm does not mutate P so that the types and semantics of P is preserved. We can exploit this fact because all declared types are immutable, therefore we cannot add any inheritance relations to fully-declared types (if a class is declared but has a missing supertype, we may always add an inheritance rule to that missing supertype). Therefore, if an abstract type is found to be a supertype of a fully-declared class, then it must be concretized to an instance of one of the superclasses of the fully-declared class.

Formally, we define $\mathbb{L}_{\text{fd}}(\alpha_i)$ be the subset of $\mathbb{L}(\alpha_i)$ containing only fully-declared types, and let $\mathbb{P}(S)$ be the set $\{T \in \mathcal{G} \mid S = T \vee \mathbf{Path}_{\mathcal{G}}(S, T), T \text{ non-abstract}\}$.

Theorem 4.4. *If $|\mathbb{L}_{fd}(\alpha_i)| \geq 1$, α_i must be concretized to an instance of one of the types in $\bigcap_{S \in \mathbb{L}_{fd}(\alpha_i)} \mathbb{P}(S)$.*

Proof. Suppose not. Suppose α_i can be concretized to an instance of non-abstract type T, where there exists some $S \in \mathbb{L}_{\text{fd}}(\alpha_i)$ such that $T \notin \mathbb{P}(S)$. Then by definition, there is no path from S to T in $\mathcal{G}$. Because S is fully-declared, we cannot add a new inheritance rule to S or any of its fully-declared supertypes to create such a path, so there can never be a path from S to T. However, if we concretize α_i to T, by definition of $\mathbb{L}_{\text{fd}}(\alpha_i)$ there must also be a path from S to T, which is a contradiction. □

4.3.4 *Other Scenarios.* Given the all-encompassing definition of an abstract type to mean 'some class type', it means that it could be an instance of any one of the existing class types in the program, or a new missing type which we must create. Fix one $\alpha_i\texttt{<...>}^{C}_{\mathbb{S}}$, and let $\mathbb{C}_1$ contain all existing class types in the program (this could be class types created from earlier iterations of Concretize), and let

the user-specified maximum arity of any type be n. We create a set $\mathbb{C}_2$ of newly created class types, `UNKNOWN_i0`, `UNKNOWN_i1<P_1>`, up to `UNKNOWN_in<P_1,...,P_n>` for fresh `i`, and let $\mathbb{C}^- = \mathbb{C}_1 \cup \mathbb{C}_2$. Finally, let $\mathbb{C}^*$ be the subset of $\mathbb{C}^-$ that are valid candidates for α_i to concretize to by observations on $\mathcal{G}$ as described earlier. Concretize then transitions the original input configuration into a set of configurations, containing one configuration for each type $S \in \mathbb{C}^*$, where α_i is replaced with $concretize(\alpha_i, S)$. The constraint store C is also lifted back into Ω, and this set of configurations goes through Resolve again.

4.4 Creating Method Declarations

When a configuration gets to the DeclareMethods stage, Ω is empty (all constraints have been solved), the class tables in Δ and Φ together are well-formed, and all types in Ψ are not unknown. From Subsection 3.3 we have only inferred method invocations, whereas method declarations are required. Since the method declarations a class has access to also depends on those in its superclass, we perform a topological sort (by inheritance) on the classes in Φ and begin declaring methods from the highest class.

Suppose DeclareMethods is currently declaring methods for class S, the user-specified maximum arity of types is n, and without loss of generality assume methods only have one parameter. If S has a method invocation $\mathtt{m} : A \longrightarrow O$, we remove this invocation from S, then assert one of the following must be true: (1) there is a method `m` already declared or inherited by S with signature $P \longrightarrow R$ such that $A =: P\phi \wedge O\phi =: R$ where ϕ is inferred by Java's type inference algorithm [GJSB05], or (2) a new method declaration $\mathtt{m} : \tau_p \longrightarrow \tau_r$ with 0 to n type parameters and signature is created and $A =: \tau_p(\mathbb{S})\phi \wedge \tau_r(\mathbb{S})\phi =: O$ where $\mathbb{S}$ contains all type parameters in this typing context (including the newly created ones). Each of these possibilities has its own configuration, and the DeclareMethods stage transitions the input configuration into this set. Finally, if S does not have any method invocations in its class declaration left, the relevant invocations in Θ where the target type are all instances of S are then checked for unambiguity, and that overriding method declarations are compatible, i.e. if there is an inherited superclass method $\mathtt{m} : P \longrightarrow R_1$ and a declared method $\mathtt{m} : P \longrightarrow R_2$ then we assert $R_2 =: R_1$. If all these have been completed, DeclareMethods moves on to the next class declaration in the topological sort.

4.5 An Example of Constraint Solving

We continue with our example presented in Figure 4a and Figure 4b. Recall that the constraints collected from incomplete program P_3 are `A<C>` := `A<C>`, `java.lang.Object` := ${\delta_3}^{\emptyset}$ and `java.lang.Object` := τ_2. As shown by rules like Disj-Sel in Figure 5, some stages of constraint solving will produce a set of multiple configurations; for brevity we only show one of the resulting configurations that lead to a successful completion of P_3.

We first start with Resolve. In the presence of disjunctive types $\tau_1 = \bigvee\{\mathtt{V}, \alpha_1\texttt{<...>}^{\emptyset}_{\{\mathtt{V}\}}\}$ and $\tau_2 = \bigvee\{\mathtt{T}, \mathtt{U}, \alpha_2\texttt{<...>}^{\emptyset}_{\{\mathtt{T},\mathtt{U}\}}\}$, we first replace them with one of their constituent choices in the configuration. Suppose we replaced τ_1 with `V` and τ_2 with $\alpha_2\texttt{<...>}^{\emptyset}_{\{\mathtt{T},\mathtt{U}\}}$. Because $\tau_1\{\mathtt{V} \mapsto \mathtt{C}\}$ has been replaced with `C`, we can see that the typing context of ${\delta_3}^{\emptyset}$ is known, so we replace it with a fresh $\tau_3(\emptyset)$, which is basically equivalent to $\alpha_3\texttt{<...>}^{\emptyset}_{\emptyset}$. We then replace δ_3 with $\alpha_3\texttt{<...>}^{\emptyset}_{\emptyset}$, giving us Φ_1 and Ω_1 in Figure 6a and Figure 6d respectively.

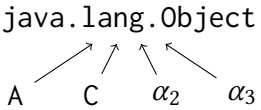
Now we can begin constraint reduction. The first constraint `A<C>` := `A<C>` reduces to True via `A<C>` := `A<C>` $\Rightarrow$ $capture$(`A<C>`) <: `A<C>` $\Rightarrow$ `A<C>` <: `A<C>` $\Rightarrow$ `C` $\leq$ `C` $\Rightarrow$ `C` <: `C` $\Rightarrow$ True. The next two constraints each reduce to subtype constraints (we elide the technicalities of capture conversion on abstract types, so we take the path where it does not matter), which are then stored in their respective constraint stores. We thus get a new configuration $\langle \Delta, \Phi_1, \emptyset, \Psi_1, \Theta_1 \rangle$.

```
1  class A<V> {
2    V x;
3    \alpha_2 m(C) {
4      // typing ctx has T, U
5      // and {V -> C}
6    }
7  }
8  class C {
9    \alpha_3 y;
10 }
```

(a) Φ_1

```
1  class A<V> {
2    V x;
3    Object m(C) {
4      // typing ctx has T, U
5      // and {V -> C}
6    }
7  }
8  class C {
9    Object y;
10 }
```

(b) Φ_2



(c) $\mathcal{G}$

$$\Omega_1 = \{\texttt{A<C>} := \texttt{A<C>}, \texttt{java.lang.Object} := \alpha_3\texttt{<...>}^{\emptyset}_{\emptyset}, \texttt{java.lang.Object} := \alpha_2\texttt{<...>}^{\emptyset}_{\{\texttt{T,U}\}}\}$$

$$\Omega_2 = \{\texttt{java.lang.Object} := \texttt{java.lang.Object}, \texttt{java.lang.Object} := \texttt{java.lang.Object}\}$$

(d) Ω_1 and Ω_2

Figure 6: States of configurations during constraint solving for incomplete program P_3

At this point Resolve is complete since there are no more constraints in the configuration, and since the type hierarchy is well-formed, Deconflict is also complete. Now we need to concretize the abstract types. By Δ, Φ and the effective constraint stores we obtain the erasure graph $\mathcal{G}$ shown in Figure 6c. Clearly, we can replace α_2 and α_3 with their concretizations to `java.lang.Object`. We then lift the constraints in the constraint stores of α_2 and α_3 back into Ω_2, giving us $\langle \Delta, \Phi_2, \Omega_2, \Psi_2, \Theta_2 \rangle$ where Φ_2 and Ω_2 are shown in Figure 6b and Figure 6d respectively. Constraint reduction would show these constraints to be satisfied, giving us $\langle \Delta, \Phi_2, \emptyset, \Psi_2, \Theta_2 \rangle$.

Now we get to the DeclareMethods stage; the only method invocation we need to deal with is $\texttt{m} : \texttt{C} \to \texttt{java.lang.Object}$. We create a new method declaration receiving no type parameters, giving it the signature $\texttt{m} : \tau_p(\{\texttt{V}\}) \to \tau_r(\{\texttt{V}\})$ where $\texttt{C} =: \tau_p$ and $\tau_r =: \texttt{java.lang.Object}$. Bringing these constraints through to Resolve, Deconflict and Concretize can show that $\tau_p = \texttt{C}$ and $\tau_r = \texttt{java.lang.Object}$, giving us $\Phi_3 = \Phi_2$. Therefore, the declarations in Φ_2 will complete the original incomplete program P_3 in Figure 4a. Running P_3 and Φ_2 through the Java compiler will indeed show this to be the case.

5 EXPERIMENTAL EVALUATION

To evaluate the efficacy of our algorithm, we developed a proof-of-concept of our algorithm as a program called the Java Compiler for Incomplete Programs (JavaCIP). JavaCIP is developed in Scala [OAC+06] and makes use of the JavaParser (`javaparser.org`) parser to build the AST of incomplete programs. As input, JavaCIP receives an incomplete Java program as a single source file, and as output, produces Java source code representing the type declarations in Φ, that completes the incomplete program. The algorithm implemented by JavaCIP is described in Algorithm 2. While the description of our algorithm omits some features of Java, JavaCIP admits most Java programs, with interfaces, static members, constructors, primitive types and array types all taken into consideration.

However, JavaCIP has not implemented the following features:

- Exceptions—while at the type level JavaCIP is able to determine that an object that is being thrown/caught must be an instance of `java.lang.Throwable` or `java.lang.Exception`, because Java may give compiler errors for uncaught/unthrown exceptions, JavaCIP assumes that all exceptions thrown are `java.lang.RuntimeExceptions`. In addition, JavaCIP disregards

Algorithm 2 JavaCIP

```
1: ast ← JavaParser(sourceFile)
2: C ← ConstGen(ast, maxBreadth)
3: if C = ⟨Δ, Φ, Ω, Ψ, Θ⟩ then
4:     R ← ConstSolve(C)
5:     if R = ⟨Δ*, Φ*, ∅, Ψ*, Θ*⟩ then
6:         for each class declaration S in Φ* do
7:             write S as Java file
```

the order of which exceptions are caught;

- Type parameter bounds—JavaCIP admits programs that have type parameter bounds but does not generate them in the surrounding dependencies, this is because the generation of type parameter bounds can complicate method declarations, since the rules for overriding and overloading depend on the erasures of types;
- Lambda expressions and method references—these are tricky to deal with because a single lambda expression in Java can be of different types, not to mention that methods can be overloaded in Java;
- Annotations;
- Static/inner/anonymous classes—while these do not threaten the validity of our approach, dealing with such classes poses a syntactic problem without providing additional insight to the problem we are tackling, so we leave this for future implementation;
- Enumerations; missing enumerations are frequently indistinguishable from regular class-level attributes
- The `var` type annotation is not supported;
- Any other feature introduced after JDK 11;

Finally, we introduced two additional features that was not discussed earlier. Firstly, it is common for some incomplete programs to be impossible to complete because the scope of variables/methods are unresolvable. To tackle this, we prepend the scope `JavaCIPUnknownScope` to these unresolvable attributes and methods so that some attempt can be made to complete them. The second feature that is included in JavaCIP is a *heuristic search*. The ConstSolveImpl algorithm chooses any configuration within the provided set of configurations and performs the four stages on it. The default behaviour would be to select the last configuration that was added into this set (i.e. the set of configurations is implemented as a stack)—this is so that we can quickly 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 a 'compiler option' to search the configurations sorted by maximum arity and height, which may, in some scenarios, allow us to complete the incomplete program more quickly.

5.1 Experimental Setup

We wish to benchmark JavaCIP against earlier works that meet similar research objectives. The two most relevant tools that complete incomplete Java programs are PPA [DH08] and JCoffee [GMP20]. Between the two, we chose to compare JavaCIP with JCoffee because (1) as of writing

Description	# Programs	Avg. Time	Avg. LoC	Polymorphic?
Completed	429 (98.4%)	2.57s	35.6	Yes
Cannot complete	6 (1.4%)	1.02s	39.2	
Timeout	1 (0.2%)	Timeout	33	
Completed	4466 (99.8%)	1.5s	24.8	No
Cannot complete	2 (0.0%)	0.69s	65	
Timeout	8 (0.2%)	Timeout	61.9	

Table 1: JavaCIP on BigCloneBench Subset

this paper, PPA has been deprecated (2) JCoffee also generates compilable Java code, while PPA generates Java bytecode directly. Note that although JCoffee is similar in objectives to JavaCIP, it has trouble handling parametrically polymorphic types and allows modifications to the input programs [GMP20].

For our experiments we provisioned two datasets. To obtain the first dataset we derived a set of 4912 programs from the BigCloneBench dataset [SIK+14] used by JCoffee, which originally contained 9133 Java programs—we derived this set by removing programs that cause JavaCIP to exhibit undefined behaviour due to the unimplemented Java constructs described above. Fewer than 1% of the programs in this set were complete; each of the incomplete programs require the generation of approximately 30 unknown types on average. The second dataset is a custom-curated dataset containing 30 small incomplete Java programs with complex relationships between polymorphic types, 15 of which can be completed while the remaining 15 have type errors. All outputs from JavaCIP are passed into the Java compiler (`javac`) [GJSB05] for verification of correctness. The experiments are run on an Arch Linux system equipped with an Intel i7-6700 CPU. The time limit for completing each program is 1 minute. Table 1 shows the results from running JavaCIP from the BigCloneBench datasets. Appendix C shows other experimental data not crucial to show here, but is still discussed in this section nonetheless.

The goal of the experiments is to answer the following research questions:
RQ1: How effectively does JavaCIP complete incomplete Java programs?
RQ2: How does JavaCIP compare with earlier works?

5.2 RQ1

4895 (99.7%) programs from the BigCloneBench dataset were successfully completed by JavaCIP in total. Manual verification of the 8 programs that cannot be completed by JavaCIP shows that they can never be completed due to unresolvable references to attributes and methods like `this.a`. Specific to the partition containing polymorphic programs, JavaCIP managed to correctly handle 99.8% of them in 1.7s on average. In addition, running JavaCIP on the custom dataset shows that JavaCIP successfully completed all 15 incomplete polymorphic programs not containing type errors, and does not complete any of the 15 programs that did contain type errors. We see this as sufficient evidence that JavaCIP completes incomplete polymorphic Java programs in a type-safe manner, without false positives. We discuss the 9 programs from BigCloneBench exceeding the time limit of 1 minute in Subsection 5.4.

5.3 RQ2

JCoffee successfully completed 2425 (49.4%) of the incomplete programs from the BigCloneBench dataset in total. Of the 436 polymorphic programs from BigCloneBench, JCoffee only managed to complete 133 (30.5%). JCoffee managed to complete these programs in less than 2s on average, and

```
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 P_5 (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 P_5 (b) which adds type casting to P_5 (a) in lines 6 and 7

Figure 7: Example incomplete program that suffers from path explosion

did not suffer from timeouts. Of the 15 programs not containing type errors in the custom dataset, JCoffee only managed to complete 2 (13%). This shows that JavaCIP outperforms JCoffee in the ability to complete incomplete Java programs, especially polymorphic ones.

5.4 Discussion on Path Explosion

Several analyses like Symbolic Execution [Kin76] suffer *path explosion* [CS13], which unsurprisingly, JavaCIP also suffers from. Mainly, JavaCIP suffers from path explosion when the incomplete program is large, and especially when dealing with the types of unknown methods—this forces the resolution of the method parameter/return types to the last stage DeclareMethods, which requires a lot of analysis to get to. Evidence of the former can be seen from the average size of incomplete programs causing timeouts in Table 1. For the latter, an example P_5 (a) is given in Figure 7a, where JavaCIP runs out of memory before being able to complete it. To mitigate these issues, a user can refine the analysis by specifying the types of some unknown types via typecasting. An example of this is shown in P_5 (b) in Figure 7b where the return types of the two invocations of `extract` are specified by typecasting, and now P_5 (b) in Figure 7b can be completed in 1 second with less than 1,000 configuration transitions. However, we look forward to future work providing mitigations to the path explosion problem that applies to our research objectives.

6 RELATED WORK

The works that are the most similar in research objectives are JCoffee by Gupta et al. [GMP20], PPA by Dagenais and Hendrenis [DH08], and PsycheC by Melo et al. [MRdAP17]. 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 performing auto-completion, receives a partial Java program and uses constraint generation and solving to produce Jimple intermediate code to support static analysis. Both JCoffee and PPA do not guarantee type safety from the generated dependencies or intermediate code. PsycheC on the other hand works on incomplete C sources; it follows a syntax-directed process of constraint generation, then uses a two-phase unification [MM82] approach to solve type constraints. All of these three algorithms do not support incomplete programs with parametrically polymorphic types.

Adjacent to our work is the extensive work done in *type inference*, for which typically some form of unification [Rob65, MM82] is employed, for example, Dolan and Mycroft's biunification [DM17] 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.

Next, we discuss the emerging trend of general-purpose use of LLMs. AI-powered program synthesis is not a new concept [MW71], with several works [OSD+21, JVI+22] leveraging Pre-Trained LLMs like GPT-3 [BMR+20] 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 [Ope] are currently not suited for our purposes. As of writing this paper, LLMs do not understand program semantics and treat programs as text [OSD+21]—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* [JLF+23] 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 [ZW17] that locates Java archive files to resolve unknown constructs and build System Dependency Graphs [FOW87] for analysis of partial programs, ParseWeb [TX07] and PRIME [MSY12] 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 [DGTS22] which determines matching import statements where they are missing.

7 CONCLUSION

This paper described a novel algorithm that extends type checking and type inference by inferring the missing class declarations of incomplete programs. The algorithm was formulated based on three key insights: (1) the domain of unknown types is known: we exploit this fact to create disjunctive types that capture this domain, (2) a way to reduce constraints by inference rules, and a strategy to add information to missing/unknown types so that irreducible constraints can be reduced: for this we force some missing types to extend other types so that a previously irreducible subtype constraint can be reducible, and (3) a way of pruning the domain of possible types without needing to consider the type arguments of unknown class types: for this we exploit properties of well-formed type hierarchies. This allowed the algorithm to systematically search for types that complete the incomplete program while preserving type safety.

Having kept our ideas rather general, we postulate that what we have proposed can support the formulation of algorithms that (likely nontrivially) extends ours, to achieve the same objective for incomplete programs written in programming languages with similar type systems, such as C++, Rust and others. We look forward to future work in this direction.

DATA-AVAILABILITY STATEMENT

JavaCIP source code, benchmark datasets (input incomplete programs and outputs produced by JavaCIP) and statistics are hosted publicly on `https://github.com/yonggqiii/javacip`, which also contains a Python script which allows users to run the benchmarks themselves.

REFERENCES

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, page 805–806, 2007.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[BPZK08] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, page 27–30, 2008.

[CD09] Nicholas Cameron and Sophia Drossopoulou. On subtyping, wildcards, and existential types. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, New York, NY, USA, 2009. Association for Computing Machinery.

[CDE08] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for java with wildcards. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, pages 2–26, 2008.

[CMJL09] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 50–62, 2009.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, feb 2013.

[DB96] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[DGTS22] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. Snr: Constraint-based type inference for incomplete java code snippets. In *Proceedings of the 44th International Conference on Software Engineering*, page 1982–1993, 2022.

[DH08] Barthelemy Dagenais and Laurie Hendrenis. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 313–328, 10 2008.

[DM17] Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, page 60–72, 2017.

[DR08] Barthélémy Dagenais and Martin Robillard. Recommending adaptive changes for framework evolution. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 481–490, 2008.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 7 1987.

[GaMadSP19] Breno C F Guimarães, José Wesley de S Magalhães, Anderson Faustino da Silva, and Fernando M Q Pereira. Synthesis of benchmarks for the c programming language by mining software repositories. In *Proceedings of the XXIII Brazilian Symposium on Programming Languages*, page 62–69, 2019.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[GMP20] Piyush Gupta, Nikita Mehrotra, and Rahul Purandare. Jcoffee: Using compiler feedback to make partial code snippets compilable. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 810–813, 2020.

[God14] Patrice Godefroid. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering*, page 539–549, 2014.

[Gri17] Radu Grigore. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 73–85, New York, NY, USA, 2017. Association for Computing Machinery.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, may 2001.

[JLF+23] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Comput. Surv.*, 55(12), mar 2023.

[JVI+22] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1219–1231, New York, NY, USA, 2022. Association for Computing Machinery.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[KLDM99] G. Knapen, B. Lague, M. Dagenais, and E. Merlo. Parsing c++ despite missing declarations. In *Proceedings Seventh International Workshop on Program Comprehension*, pages 114–125, 1999.

[KP07] Andrew Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, January 2007.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 4 1982.

[MRdAP17] Leandro T. C. Melo, Rodrigo G. Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. Inference of static semantics for incomplete c programs. *Proceedings of the ACM on Programming Languages*, 2(POPL), 12 2017.

[MSY12] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, page 997–1016, 2012.

[MW71] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, mar 1971.

[OAC+06] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[Ope] OpenAI. Introducing chatgpt.

[OSD+21] Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. Program synthesis with large language models. In *n/a*, page n/a, n/a, 2021. n/a.

[PBMH13] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Adoption and use of java generics. *Empirical Software Engineering*, 18(6):1047–1089, 12 2013.

[PGBG12] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 275–286, 2012.

[RGP19] Marcus Rodrigues, Breno Guimarães, and Fernando Magno Quintão Pereira. Generation of in-bounds inputs for arrays in memory-unsafe languages. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 136–148, 2019.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1 1965.

[RV05] Giovanni Rimassa and Mirko Viroli. Understanding access restriction of variant parametric types and java wildcards. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, page 1390–1397, New York, NY, USA, 2005. Association for Computing Machinery.

[SCDCD10] Alexander J. Summers, Nicholas Cameron, Mariangiola Dezani-Ciancaglini, and Sophia Drossopoulou. Towards a semantic model for java wildcards. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, New York, NY, USA, 2010. Association for Computing Machinery.

[SIK+14] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, 2014.

[TEH05] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild fj. 2005.

[THE+04] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the java programming language. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, page 1289–1296, 2004.

[TX07] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, page 204–213, 2007.

[VRCG+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, page 214–224, 2010.

[XZX22] Zhipeng Xue, Yuanliang Zhang, and Rulin Xu. Clone-based code method usage pattern mining. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, page 543–547, 2022.

[ZW17] Hao Zhong and Xiaoyin Wang. Boosting complete-code tool for partial program. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, page 671–681, 2017.

[ZZL+06] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approace to feature location. *ACM Transactions on Software Engineering and Methodology*, 15(2):195–226, 4 2006.

A PROOFS OF THEOREMS IN SECTION 3

Recall that the construction of $\mathbb{T}_n(\mathbb{S})$ is as follows:

1. if $P \in \mathbb{S}$ then $P \in \mathbb{T}_1(\mathbb{S})$
2. if S is a class type with arity 0 then $S \in \mathbb{T}_1(\mathbb{S})$
3. if S is a class type with arity 1 then $S\texttt{<?>} \in \mathbb{T}_2(\mathbb{S})$
4. if $A \in \mathbb{T}_{n-1}(\mathbb{S})$ and S is a class type with arity 1 then $S\texttt{<}A\texttt{>}, S\texttt{<? }\textbf{extends}\ A\texttt{>}, S\texttt{<? }\textbf{super}\ A\texttt{>} \in \mathbb{T}_n(\mathbb{S})$.

We can now prove the two theorems in Section 3.

Theorem 3.1. *For all $\tau = \bigvee\left(\mathbb{S} \cup \{\alpha\texttt{<}\ldots\texttt{>}^{C}_{\mathbb{S}}\}\right)$ where $\mathbb{S}$ are type parameters, for any sequence of valid replacements χ_1 to χ_m, if $\tau\chi_1 \ldots \chi_m$ has height n and does not contain unknown types, $\tau\chi_1 \ldots \chi_m \in \mathbb{T}_n(\mathbb{S})$.*

Proof. We proceed by straightforward induction. We first prove the statement where $n = 1$. By definition, any valid replacement must replace τ with (case **A**) one of the type parameters in $\mathbb{S}$, or (case **B**) $\alpha\texttt{<}\ldots\texttt{>}^{C}_{\mathbb{S}}$. By rule (1), all types in $\mathbb{S}$ are in $\mathbb{T}_1(\mathbb{S})$, so (case **A**) has been shown. For (case **B**), since $\alpha\texttt{<}\ldots\texttt{>}^{C}_{\mathbb{S}}$ itself is unknown, it must be replaced with a concretization into a class type which must have height 1, so it must be replaced with its concretization into a class S with arity 0; By rule (2), $S \in \mathbb{T}_1(\mathbb{S})$ for all such types S, so (case **B**) is also shown.

We now show that the statement where $n = k$ will imply the statement where $n = k + 1$, for $k \geq 1$. Any valid replacement χ_1 must replace τ with $\alpha\texttt{<}\ldots\texttt{>}^{C}_{\mathbb{S}}$; it cannot replace τ with a type parameter in $\mathbb{S}$ because those have height 1. That necessarily means that there must be another valid replacement χ_2 that replaces $\alpha\texttt{<}\ldots\texttt{>}^{C}_{\mathbb{S}}$ with its concretization to a class S with arity 1, which is $S\texttt{<}\tau_i\texttt{>}$ where $\tau_i = \bigvee\left(\mathbb{S} \cup \{\alpha_i\texttt{<}\ldots\texttt{>}^{\emptyset}_{\mathbb{S}}\}\right)$ (we omit the wildcard arguments because they can similarly be argued by

rules (3) and (4)). We know by the induction hypothesis that all valid replacements χ_3 to χ_m on τ_i that produce a type T of height k that does not contain unknown types will be in $\mathbb{T}_k$, so by rule (4), $\tau\chi_1\chi_2\chi_3\ldots\chi_m = \alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}}\chi_2\chi_3\ldots\chi_m = S\texttt{<}\tau_i\texttt{>}\chi_3\ldots\chi_m = S\texttt{<}T\texttt{>} \in \mathbb{T}_{k+1}(\mathbb{S})$ for all $\chi_1\chi_2\chi_3\ldots\chi_m$ and S. □

Theorem 3.2. *For all $\tau = \bigvee\left(\mathbb{S} \cup \{\alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}}\}\right)$ where $\mathbb{S}$ are type parameters, for all $S \in \mathbb{T}_n(\mathbb{S})$, there exists a sequence of valid replacements χ_1 to χ_m such that $\tau\chi_1\ldots\chi_m = S$.*

Proof. We proceed by construction and induction. We first prove the statement where $n = 1$. There are two cases for a type S to be an element of $\mathbb{T}_1(\mathbb{S})$, (case **A**) where $S = P$ is a type parameter in $\mathbb{S}$, and (case **B**) where $S = T$ is a class type of arity 0. (Case **A**) can be constructed with $\chi_1 = \{\tau \mapsto P\}$, thus $\tau\chi_1 = P = S$. (Case **B**) can be constructed with $\chi_2 = \{\tau \mapsto \alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}}\}$ and $\chi_3 = \{\alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}} \mapsto \mathit{concretize}(\alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}}, T)\}$ where $\mathit{concretize}(\alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}}, T) = T$. This is so that $\tau\chi_2\chi_3 = T = S$.

Next we show that the statement for $n = k$ implies the statement where $n = k + 1$ for $k \geq 1$. We let $S = T\texttt{<}U\texttt{>} \in \mathbb{T}_{k+1}(\mathbb{S})$ where U has height k. We let $\chi_1 = \{\tau \mapsto \alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}}\}$ and $\chi_2 = \{\alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}} \mapsto \mathit{concretize}(\alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}}, T)\}$ where $\mathit{concretize}(\alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}}, T) = T\texttt{<}\tau_i\texttt{>}$ where $\tau_i = \bigvee\left(\mathbb{S} \cup \{\alpha\texttt{<}\ldots\texttt{>}^C_{\mathbb{S}}\}\right)$ (we omit showing the case for wildcard type arguments but the proof for those is similar). Therefore, $\tau\chi_1\chi_2 = T\texttt{<}\tau_i\texttt{>}$. Since $U \in \mathbb{T}_k(\mathbb{S})$ by rule (4), by the induction hypothesis we know that there exists a sequence of replacements $\chi_3\ldots\chi_m$ such that $\tau_i\chi_3\ldots\chi_m = U$, so $\tau\chi_1\chi_2\chi_3\ldots\chi_m = T\texttt{<}\tau_i\texttt{>}\chi_3\ldots\chi_m = T\texttt{<}\tau_i\chi_3\ldots\chi_m\texttt{>} = T\texttt{<}U\texttt{>}$. □

B PROOFS OF THEOREMS IN SECTION 4

We first begin with some supporting lemmas.

Lemma B.1. *If for any S we have $|\mathbb{E}_c(S)| > 1$, constraint solving should fail.*

Proof. Suppose $\mathbb{E}_c(S)$ contains two distinct non-abstract class types S and T. By definition, these two types are in a cycle in $\mathcal{G}$, meaning they cyclically inherit each other, which is not permissible in a well-typed Java program. □

Lemma B.1 gives us Lemma B.2 and Lemma B.3.

Lemma B.2. *If any α_i is concretized to an instance of S, then all types in $\mathbb{E}_a(\alpha_i)$ must each be concretized to an instance of S.*

Proof. Suppose not; suppose $\alpha_j \in \mathbb{E}_a(\alpha_i)$, $i \neq j$, and α_i is concretized to an instance of S, but α_j is concretized to an instance of T where $\epsilon(S) \neq \epsilon(T)$. Then by Lemma B.1 this is also not permissible. □

Lemma B.3. *If for any α_i we have $\mathbb{E}_c(\alpha_i) = \{S\}$, then α_i must be concretized to an instance of S.*

Proof. Suppose not (α_i is replaced with its concretization an instance of some other type $T \neq S$); therefore, we will have $\mathbb{E}_c(\alpha_i) = \{S, T\}$, which Lemma B.1 this is not permissible. □

These three Lemmas give a straightforward proof of Theorem 4.2.

Theorem 4.2. *For any α_i, if $\mathbb{E}_c(\alpha_i) = \{S\}$, then all types in $\mathbb{E}_a(\alpha_i)$ must all be concretized to instances of S.*

Proof. Lemma B.3 tells us that α_i must be replaced with its concretization to an instance of S, and Lemma B.2 generalizes this statement to all types in $\mathbb{E}_a(\alpha_i)$. □

C ADDITIONAL EXPERIMENTAL RESULTS

In this section we show additional experimental results not shown earlier in the main paper. Detailed file-by-file statistics and JavaCIP outputs are available in JavaCIP's public repository and supplementary material. JCoffee [GMP20] is also publicly available.

We first show statistics from running JCoffee with the BigCloneBench dataset in Table 2. While the authors of JCoffee reported a better success rate of 90% where only around 1K incomplete programs failed to be completed by JCoffee out of 9133, we were not able to replicate this result on our system.

Description	# Programs	Avg. Time	Avg. LoC	Polymorphic?
Completed	133 (30.5%)	2.12s	31.5	Yes
Failed	303 (69.5%)	4.83s	37.5	
Completed	2292 (51.2%)	1.37s	21.8	No
Failed	2184 (48.8%)	4.55s	28.1	

Table 2: JCoffee on BigCloneBench dataset

Next, we show results from executing JavaCIP on our custom dataset in Table 3. The leftmost three columns show statistics from the incomplete programs that do not contain type errors. JavaCIP successfully completed all of these programs in 1.58s on average, and on average these programs contained 14.9 lines of code. The rightmost three columns show statistics from the incomplete programs that could never be completed due to type errors. JavaCIP cited the existence of type errors for all these programs in 1.41s on average, and on average these programs contained 14.3 lines of code.

Filename	Time	LoC	Filename	Time	LoC
1.java	1.62s	13	8.java	1.82s	14
2.java	1.47s	29	13.java	0.92s	12
3.java	1.57s	34	14.java	3.88s	17
4.java	1.07s	10	15.java	0.62s	9
5.java	2.72s	18	18.java	0.61s	10
6.java	0.92s	11	19.java	0.92s	18
7.java	2.32s	15	20.java	0.82s	14
9.java	1.62s	11	21.java	0.61s	8
10.java	1.62s	12	22.java	1.87s	24
11.java	0.82s	10	23.java	1.17s	22
12.java	0.97s	7	24.java	3.42s	13
16.java	0.97s	11	25.java	0.87s	12
17.java	4.12s	10	26.java	1.62s	13
29.java	1.07s	12	27.java	1.32s	17
30.java	1.37s	20	28.java	0.72s	11

Table 3: JavaCIP on custom dataset

Finally, we show results from running JCoffee on our custom dataset in Table 4. Only the 15 incomplete programs not containing type errors were used to test JCoffee, because JCoffee requires the assumption that all input programs can be completed. Of the 15 incomplete programs that can be completed, JCoffee only successfully completed two, spending 2.25s on average for these two programs.

Filename	Time	LoC	Completed
1.java	10.48s	13	No
2.java	10.53s	29	No
3.java	10.58s	34	No
4.java	1.67s	10	Yes
5.java	2.82s	18	Yes
6.java	10.43s	11	No
7.java	10.48s	15	No
9.java	10.48s	11	No
10.java	10.28s	12	No
11.java	10.38s	10	No
12.java	10.58s	7	No
16.java	10.28s	11	No
17.java	10.38s	10	No
29.java	10.58s	12	No
30.java	10.43s	20	No

Table 4: JCoffee on custom dataset