Type-Safe Auto-Completion of Incomplete Polymorphic Programs

Yong Qi Foo
yongqi@nus.edu.sg
National University of Singapore

Siau-Cheng Khoo
khoosc@nus.edu.sg
National University of Singapore

1 OVERVIEW

Incomplete programs are ubiquitous in web repositories, evolving software projects and beyond, but are difficult to work with due to references to undeclared constructs. Program auto-completion enables static analysis on incomplete programs and boosts developer productivity. However, earlier efforts cannot handle parametrically polymorphic types (which are frequently used) and do not make guarantees on type safety.

We present a new algorithm that receives an **incomplete polymorphic Java program** P and **reconstructs its surrounding dependencies** R in a **type-safe** manner such that P and R together form a **complete and well-typed program**. Our algorithm extends **constraint generation** and **constraint solving** used by many type-checking and type inference algorithms.

2 CONSTRAINT GENERATION

- Constraints are generated by analysis of the Abstract Syntax Tree (AST) of the incomplete program: `a = b` in P_1 produces constraint `B<D>` <: `A<? extends C>`
- Type of `a.x` is unknown because class declaration for `A` is missing: we create a new declaration class `A<V>`, and since `x` occurs in `A<V>`, `x` *must be* `V` *or some other class type that can only contain type parameter* `V`
- We encode this as $\tau_1 = \mathsf{V}\{\mathsf{V}, {\alpha_1}_{\{\mathsf{V}\}}\}$ where α_1 is some unknown class type
- Assigning τ_1 as type of `x` lets us soundly generate type constraints on type of `a.x`:
  - `a.x = it` gives `Iterable<String>` <: τ_1`{V ↦? extends C}`
  - `a.x = "Hello APLAS!"` gives `String` <: τ_1`{V ↦? extends C}`

```
class B<T> extends A<T> {
  Iterable<String> it;
  void main() {
    A<? extends C> a = new A<>();
    B<D> b = new B<>();
    a = b;
    a.x = it;
    a.x = "Hello APLAS!";
  }
}
```

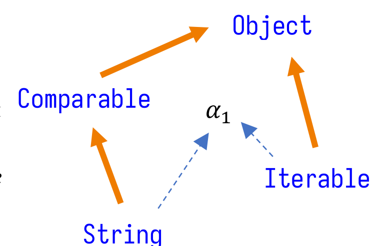
Incomplete program P_1

3 CONSTRAINT SOLVING

- Constraint solving is usually done by *constraint reduction*: if **B** implies **A** then **A** *reduces* to constraint **B**. For example, since `B<T>` extends `A<T>`, because `D` <: `C` implies `A<D>` <: `A<? extends C>` which implies `B<D>` <: `A<? extends C>`, we reduce `B<D>` <: `A<? extends C>` into `D` <: `C`
- *If a constraint cannot be reduced due to missing class declarations, we add more information to the program so that it can be reduced.* `D` <: `C` cannot be reduced further since `D` is missing, so we make `D` extend `C`, which makes the constraint reduce to True and therefore solved
- Types like τ_1 can be resolved by replacing it with any one of its choices and seeing if the constraints hold, for example we know τ_1 cannot be `V` since the constraint `String` <: `? extends C` does not hold, therefore τ_1 must be α_1.
- α_1 can be any possible class type that may be parametrically polymorphic but we do not know what its arity or type arguments are, so we analyse the *erasure graph* to eliminate some possible selections of α_1; the erasure graph is an *abstraction of the program's type hierarchy* that *does rely on knowledge of type arguments*
- Since there must be a path from `String` to α_1 and `Iterable` to α_1, α_1 must be `Object`. Replacing α_1 with `Object` solves last two constraints, and we have arrived at R_1, such that P_1 and R_1 together forms a complete and well-typed program

```
class A<V> {
    Object x;
}
class C { }
class D extends C { }
```

Program R_1 completes P_1



Erasure subgraph of P_1

4 EXPERIMENTAL EVALUATION

- Implementation of the algorithm tested on 4912 incomplete programs, 436 of them included parametrically polymorphic types
- Each program has ~30 unknown types; time limit was 1 minute
- Programs that could not be completed were actually impossible to complete, therefore no false positives
- Algorithm suffers from path explosion due to significant branching in search space

Description	# Programs	Avg. Time
Complete	4895 (99.7%)	1.7s
Cannot complete	8 (0.2%)	1.3s
Timeout	9 (0.2%)	-

5 CONCLUSION

We created a new algorithm that completes incomplete polymorphic Java programs that extends traditional constraint generation and constraint solving and type checking and type inference. We postulate that the ideas we presented allow future work to extend (likely nontrivially) ours to *complete incomplete polymorphic programs in other languages* like C++ and beyond.

Prototype is hosted on `https://github.com/yonggqiii/javacip`

NUS National University of Singapore

APLAS 2023
taipei, taiwan