API-Driven Program Synthesis for Testing Static Typing Implementations

THODORIS SOTIROPOULOS, ETH Zurich, Switzerland
STEFANOS CHALIASOS, Imperial College London, UK
ZHENDONG SU, ETH Zurich, Switzerland

We introduce a novel approach for testing static typing implementations based on the concept of *API-driven program synthesis*. The idea is to synthesize type-intensive but small and well-typed programs by leveraging and combining *application programming interfaces (APIs)* derived from existing software libraries. Our primary insight is backed up by real-world evidence: a significant number of compiler typing bugs are caused by small test cases that employ APIs from the standard library of the language under test. This is attributed to the inherent complexity of the majority of these APIs, which often exercise a wide range of sophisticated type-related features. The main contribution of our approach is the ability to produce small client programs with increased feature coverage, without bearing the burden of generating the corresponding well-formed API definitions from scratch. To validate diverse aspects of static typing procedures (i.e., soundness, precision of type inference), we also enrich our API-driven approach with fault-injection and semantics-preserving modes, along with their corresponding test oracles.

We evaluate our implemented tool, THALIA, on testing the static typing implementations of the compilers for three popular languages, namely, Scala, Kotlin, and Groovy. THALIA has uncovered 84 typing bugs (77 confirmed and 22 fixed), most of which are triggered by test cases featuring APIs that rely on parametric polymorphism, overloading, and higher-order functions. Our comparison with state-of-the-art shows that THALIA yields test programs with distinct characteristics, offering additional and complementary benefits.

CCS Concepts: • **Software and its engineering → Compilers**; **Software testing and debugging**.

Additional Key Words and Phrases: compiler bug, compiler testing, type system, API, library, enumeration

ACM Reference Format:
Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-Driven Program Synthesis for Testing Static Typing Implementations. *Proc. ACM Program. Lang.* 8, POPL, Article 62 (January 2024), 32 pages. https://doi.org/10.1145/3632904

1 INTRODUCTION

Type safety is a fundamental property contributing to the correct execution of computer programs. *Static typing*, an integral process in every statically-typed programming language implementation (typically part of the compiler), verifies that a source program is type-correct and type-safe based on a *type system*. A type system lies at the heart of the design of a language—it outlines a set of rules regarding the language's types, and how these types can be used and combined [Pierce 2002]. Language designers and researchers strive to build sound type systems [Cortier et al. 2017; Milano et al. 2022; Tate et al. 2011], which guarantee type safety, i.e., being able to identify all potential type errors during compilation.

Authors' addresses: Thodoris Sotiropoulos, ETH Zurich, Switzerland, theodoros.sotiropoulos@inf.ethz.ch; Stefanos Chaliasos, Imperial College London, UK, s.chaliasos21@imperial.ac.uk; Zhendong Su, ETH Zurich, Switzerland, zhendong.su@inf.ethz.ch.

ACM 2475-1421/2024/1-ART62
https://doi.org/10.1145/3632904

At the same time, modern languages continuously evolve with new (and often sophisticated) features to provide users with a smoother programming experience. However, integrating these new features into a language poses significant challenges and considerations for implementing sound and practical type systems. Indeed, recent work [Chaliasos et al. 2021, 2022] has shown that the static typing implementations of well-established compilers suffer from a substantial number of bugs [Amin and Tate 2016]. These implementation flaws lead to (1) reliability and security ramifications on the programs compiled with the faulty compilers, and (2) degradation of the programmers' experience and productivity. In particular, compiler typing bugs typically cause frustrating rejections of well-typed programs, dangerous acceptances of erroneous, type-unsafe programs, or annoying crashes and compilation performance issues.

In spite of the sharp rise of compiler testing research, improving the reliability of compilers' type checkers has been until recently a neglected problem. Indeed, most of the existing compiler testing techniques target optimizing compilers [Donaldson et al. 2017; Le et al. 2014; Livinskii et al. 2020, 2023; Yang et al. 2011; Zhang et al. 2017]. As shown in the empirical work of Chaliasos et al. [2021], detecting typing bugs demands new compiler testing methods, because typing bugs exhibit distinct characteristics (e.g., symptoms, root causes) compared to optimization bugs.

Currently, there are only a few approaches to validating static typing implementations. Focusing on Rust's type checker, Dewey et al. [2015] have introduced a fuzzing approach based on constraint logic programming. More recently, Chaliasos et al. [2022] have developed HEPHAESTUS, which produces type-intensive programs written in an intermediate language (IR) that supports parametric polymorphism and type inference. This high-level IR allows targeting multiple languages (i.e., Java, Groovy, and Kotlin).

All the aforementioned approaches rely on *generative compiler testing*: constructing programs entirely from scratch based on the syntactic and semantic rules of a language under test. However, generative compiler testing involves a significant limitation: it is unable to test features beyond what the underlying code generator can handle and synthesize. Therefore, the utility of generative techniques can easily saturate [Amalfitano et al. 2015], producing programs that exhibit the same programming idioms, unless new language constructs are implemented in the program generator.

Approach: We introduce a novel approach for testing static typing implementations based on the concept of *API-driven program synthesis*. Our approach is motivated by the findings of a recent empirical study [Chaliasos et al. 2021]: around one third of compiler typing bugs are triggered by small test cases that employ *application programming interfaces (APIs)* from the standard library of the language being tested. This finding is well-justified, considering the inherent complexity of the included API definitions (e.g., functions, variables, types), which heavily rely on advanced type-related features such as parametric polymorphism. Building upon this finding, our approach leverages the abundance of existing APIs, and produces small (yet complex) client programs, *without* the burden of producing the corresponding well-formed definitions from scratch. Our approach exploits the fact that although libraries are pre-compiled, a compiler still performs type checking on every client program that refers to components of a library API. This ensures that the features provided by the API are used in a type-safe manner.

At the high level, our approach works as follows. The input is an API, which is modeled as an *API graph*, a structure that captures the dependencies and relations among API components. Then, our approach proceeds with the concept of *API enumeration*: for every component d (i.e., method/field) of the input API, it systematically explores all unique, type-safe usages of d w.r.t. the signature of d. Invoking an API component through different typing patterns lets us exercise the implementation of many type-related operations in the compiler (e.g., subtyping rules). The outcome of the API enumeration process is a finite set of *abstract typed expressions*. An abstract typed expression is

encoded as a sequence of types that represents which specific types are combined to employ a certain API entity. In turn, our approach concretizes each abstract typed expression by replacing each type in the sequence with an *inhabitant* [Urzyczyn 1997] that re-uses code from the input library. To synthesize an inhabitant, the approach consults the API graph to identify sequences of method calls or field accesses that match the given type. By default, our method yields well-typed client programs. Consequently, a compiler is expected to successfully accept these generated programs. Failures to do so indicate potential bugs in the compiler.

Our method is also equipped with two modes that allow the discovery of type inference and soundness bugs respectively. Specifically, the first mode produces well-typed programs with omitted type information. To do so, when encountering a polymorphic call, our method provides no explicit type arguments whenever these can be inferred by the surrounding context. The second mode produces ill-typed programs by enumerating all those abstract-typed expressions that employ a specific API component in a way that violates typing rules. This mode enables the detection of soundness bugs by finding cases where the compiler mistakenly accepts ill-typed programs.

Results: Our implementation, which we call THALIA,[1] produces programs written in three popular languages: Scala, Groovy, and Kotlin. In our evaluation, we collected a large corpus of popular APIs taken from Maven's central software repository. Based on the collected APIs, THALIA produced small client programs that were able to trigger 84 unique bugs, of which 77 have been either confirmed or fixed. Our two modes helped identify 27 type inference bugs, and 10 bugs triggered by wrongly-typed code. When comparing our work with the state-of-the-art framework HEPHAESTUS [Chaliasos et al. 2022], we find that, despite producing test programs one order of magnitude smaller than prior work, THALIA is able to detect at least 42 bugs missed by it. Furthermore, THALIA's programs exercise previously unexplored compiler regions, leading to an increase in HEPHAESTUS' code coverage by up to 9% (2,567) for lines of code, 10% (17,548) for branches, and 8% (581) for functions.

Contributions: Our work makes the following contributions.

- The notion of API enumeration, which systematically examines API component usages via diverse valid (or invalid) typing patterns (aka abstract-typed expressions). This allows the exploration of complex type-related functionalities in the compiler, making our approach suitable for testing.
- A novel API-driven program synthesis approach for producing small yet complex client programs based on well-typed (or ill-typed) abstract-typed expressions via API enumeration.
- An open-source implementation called THALIA, which is able to produce client code written in three different languages: Scala, Kotlin, and Groovy.
- An extensive evaluation of THALIA covering several aspects, including bug-finding capability, characteristics of test cases, code coverage, and comparison with state-of-the-art. Overall, THALIA uncovered 84 new faults triggered in three widely-used compilers. These failures were caused by a plethora of language features available in the input APIs.

2 BACKGROUND AND ILLUSTRATIVE EXAMPLES

This section presents the background and motivation of our API-driven approach by discussing the limitations of the existing techniques in generative compiler testing, especially those of HEPHAESTUS, and how we address them.

Limitations of Hephaestus: HEPHAESTUS [Chaliasos et al. 2022] is the state-of-the-art tool for finding compiler typing bugs. It adopts a generative process similar to that of Csmith [Yang et al. 2011]. Specifically, HEPHAESTUS creates a bunch of random class/method/field definitions that

[1] In Greek mythology, Thalia was a nymph daughter of the smithing god Hephaestus.

```
import java.util.LinkedList
import com.google.common.collect.LinkedHashMultiset
def test(): Unit {
   val x: Tuple1[LinkedHashMultiset[String]] = ???
   val res: Any = x._1.retainAll(new LinkedList())
}
```

(a) Test case written in Scala.

```
package com.google.common.collect;
public class LinkedHashMultiset<T> extends AbstractMultiSet<T> { }
class AbstractMultiSet<T> implements MultiSet<T> {
  public void retailAll(Collection<?> p) {}
}
```

(b) Definition of the LinkedHashMultiset API in the guava library.

Fig. 1. DOTTY-17391: This program triggers a crash in Dotty. The program exercises the guava API.

respect the syntax and the semantics of a target language (in this case HEPHAESTUS' IR). This generative process involves three major limitations:

- The generated programs contain *only* a limited set of features. This means that the program generator is unable to exercise a certain feature, if this feature is not supported by the implementation. Extending the program generator with new features is technically hard, and often only works for a specific language. This is because the program generator needs to apply a set of language-specific semantic checks to ensure the validity of the generated programs. For example, creating a well-formed class that implements multiple interfaces requires checking that the interfaces have no conflicting method signatures.
- Even after using diverse configurations, the generative process easily comes to a saturation point [Amalfitano et al. 2015], because the resulting programs are somewhat biased in generating programs that exhibit the same programming idioms.
- To test a new language (e.g., Rust), someone typically needs to engineer a new program generator.

The benefits and power of APIs: To tackle these limitations of HEPHAESTUS, we introduce a complementary approach that relies on real-world *APIs*, which are well-engineered, expressive, and utilize diverse language features. Rather than generating large programs consisting of random definitions, we synthesize small but intricate client programs that invoke components (e.g., method) from a given API. This allows us to (1) effortlessly exercise a rich set of features without the added complexity of creating complex definitions from scratch, and (2) combine individual features in interesting and unexpected ways. In contrast to conventional generative processes (see HEPHAESTUS), API-based test programs provide the following unique benefits:

- *Rich feature coverage for free*: As APIs are typically designed for use in a wide variety of contexts, they often combine diverse sophisticated and demanding language features, such as bounded polymorphism, subtyping, or overloading. Given that their interfaces are also likely to be well-tested and widely used, they are free from type errors. Therefore, a feature-agnostic and potent generation process can be achieved by creating test programs centered around APIs without creating these complex and well-typed API definitions ourselves. Furthermore, the huge variety of real-world APIs avoids, or at least delays, the saturation of the generation process.
- *Applicability*: APIs are ubiquitous in mainstream languages. Generating type-intensive programs for a new language mostly requires the collection of its APIs (see Section 3.6—Generalizability).
- *Efficient testing*: Generating small programs improves the throughput of testing, as the compilers under test require considerably less time to process small inputs than large ones.

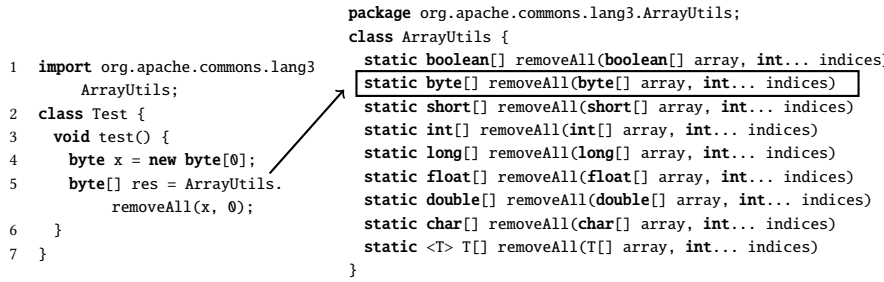


Fig. 2. GROOVY-11053: A well-typed program rejected by the Groovy compiler. The program exercises the apache-commons-lang3 library.

- *Simple test-case reduction*: Test-case reduction is highly important for compiler testing campaigns [Regehr et al. 2012]. By construction, API-driven program synthesis yields small, self-contained programs that require minimal test-case reduction.

Consider the following two bugs found by API-driven programs synthesized by our tool.

Internal compiler error in the Scala 3 compiler: Figure 1a shows a valid program that triggers a crash in the compiler of Scala 3, also known as Dotty. The program imports the `LinkedHashMultiset` class from `com.google.guava:guava` (line 2), a popular library that extends the Java collections framework. On line 4, the code uses the Scala standard library to create a tuple containing an element of type `LinkedHashMultiset<String>`. Next, by accessing this element, the program invokes the method `retainAll` found in the `LinkedHashMultiset` class.

Figure 1b shows the `LinkedHashMultiset` class as defined in the `guava` library. The class inherits `retainAll` from base class `AbstractMultiSet`. Notably, `AbstractMultiSet` involves a `default` access modifier (line 3), meaning that the class is inaccessible from code outside the package `com.google.common.collect`. However, its method `retainAll` features a `public` access modifier (line 4), which suggests that the method can be accessed by any public subclass of `AbstractMultiSet`, including `LinkedHashMultiset`. Although `retainAll` belongs to the API of `LinkedHashMultiset`, Dotty mistakenly treats the method as inaccessible, which in turn leads to a compiler crash while typing the method call on line 5 (Figure 1a). Surprisingly, replacing the receiver expression `x._1` with any other expression of type `LinkedHashMultiset<String>` successfully compiles the program.

Unexpected method ambiguity error in `groovyc`: Figure 2 shows another bug where `groovyc` erroneously rejects a well-typed program. The code first defines an array of `byte` (line 4), and then calls a static method from the `ArrayUtils` API of the `org.apache.commons:commons-lang3` library to remove the element of the array at position 0 (line 5).

The API of `ArrayUtils` is quite complex: it provides nine overloaded methods all named `removeAll`. Eight of them operate on primitive arrays, while the remaining one is a polymorphic method operating on arrays of reference types. When a compiler encounters a call to an overloaded method, it chooses to invoke the most specific one based on a set of criteria, such as the types of the provided arguments. Although it is clear from the context that the code intends to call the method variant that takes an array of bytes, a bug in `groovyc`'s method resolution causes the compiler to accidentally reject the program with an error of the form: *"reference to method removeAll is ambiguous. Cannot choose between candidate methods."* The root cause of the failure lies in the way variable arguments are matched against formal parameter types related to primitive types.

As the two examples illustrate, although the invoked APIs seem simple at a first glance, it is in fact challenging for the compilers to correctly handle these APIs. Indeed, the `guava` API exercises

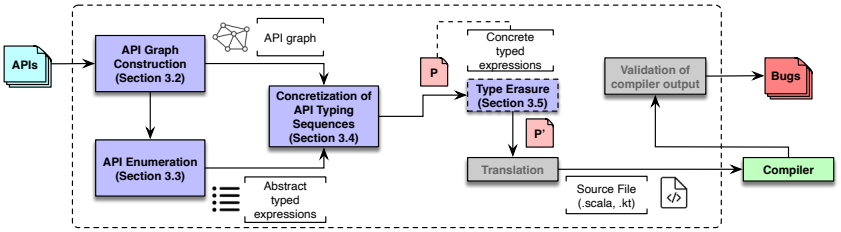


Fig. 3. The API-driven program synthesis approach for finding compiler typing bugs.

access modifiers and complex inheritance scenarios, while the `apache-commons-lang3` API has many overloaded methods mixed with plenty of type-related features (e.g., primitive types, array types, and variable arguments). Interestingly, API components might hide other typing features. For example, although calling method `retainAll` in Figure 1a knows nothing about what an access modifier stands for, or the inheritance chain of the receiver, the corresponding test case triggers a compiler bug associated with these "hidden" features.

3 API-DRIVEN PROGRAM SYNTHESIS

Figure 3 gives an overview of our API-driven program synthesis approach. The process begins with a corpus of APIs extracted from either the standard library of the language being tested or its third-party libraries. These APIs are represented in an *API graph* which captures the dependencies and type signature of each API component (Section 3.2). Next, our approach performs *API enumeration* which systematically explores all possible invocations of the encompassed API components through unique typing patterns (Section 3.3). A typing pattern is a sequence of types that corresponds to *abstract typed expressions*. Conceptually, an abstract typed expression represents a combination of types used to invoke a particular API entity (e.g, method). An abstract expression can be either well-typed or ill-typed with regards to the type signature of the corresponding API component. Then, the approach yields well-typed or ill-typed programs by concretizing each abstract-typed expression into a concrete one written in a reference language called API-IR. To do so, it examines the API graph to find type inhabitants by enumerating the set of paths that reach a specific type node via standard graph reachability algorithms (Section 3.4).

As an optional step, our approach employs the type erasure process whose purpose is to remove the type arguments of polymorphic calls, while maintaining the type correctness of the original expression (Section 3.5). The insight is that constructing polymorphic calls without explicit type arguments helps exercise the implementations of type inference operations. The final step is to employ language-specific translators that transform an expression in API-IR into a source file written in an actual language (e.g., Scala). Ultimately, the generated source files are given as input to the compiler under test, whose output is checked against the given oracle for potential bugs. In particular, we expect the compiler to accept the programs derived from well-typed typing sequences, and reject those that come from ill-typed ones. We now describe our approach in detail.

3.1 Preliminary Definitions

Figure 4 presents the core language (called API-IR) that we use as a base to explain our approach. API-IR is a basic language that supports three major programming constructs: classes, functions, and fields/variables. [2] API-IR is also equipped with parametric polymorphism in the style of Java-like generics. The API-IR is designed with simplicity in mind, ensuring that the principles of our approach (discussed in subsequent sections) can be adapted to any language compatible with API-IR.

[2]In our implementation, the core language is extended with more sophisticated features, such as lambdas, or wildcard types.

$$
\begin{aligned}
\langle \Lambda \in \mathit{API} \rangle &::= \overline{v} \\
\langle v \in \mathit{APIMember} \rangle &::= c \mid d \\
\langle c \in \mathit{Class} \rangle &::= \texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{extends}\ \overline{t}\ :\ \overline{v} \\
\langle d \in \mathit{Def} \rangle &::= \texttt{fun}\ m\texttt{<}\overline{\alpha}\texttt{>}(\overline{x : t}) : t \mid \texttt{var}\ f : t \\
\langle p \in \mathit{Program} \rangle &::= \overline{e} \\
\langle e \in \mathit{Expr} \rangle &::= \epsilon \mid \mathit{constant}(t) \mid e.f \\
&\quad \mid \texttt{local var}\ x : t = e \\
&\quad \mid e.m\texttt{<}\overline{t}\texttt{>}(\overline{e}) \\
\langle f, x \in \mathit{VariableName} \rangle &::= \textit{the set of field names in lib } \Lambda \\
\langle m \in \mathit{MethodName} \rangle &::= \textit{the set of method names in lib } \Lambda \\
\langle C \in \mathit{ClassName} \rangle &::= \textit{the set of class names in lib } \Lambda
\end{aligned}
$$

(a) Syntax

$$
\begin{aligned}
\langle t \in \mathit{Type} \rangle &::= \top \mid \bot \mid \alpha \mid N \mid N\texttt{<}\overline{t}\texttt{>} \\
\langle \alpha \in \mathit{TypeVariable} \rangle &::= \phi : t \\
\langle N \in \mathit{ClassType} \rangle &::= \mathcal{T}\texttt{<}\overline{\alpha}\texttt{>} : \overline{t} \\
\langle \phi \in \mathit{TypeVarName} \rangle &::= \textit{the set of type variable names in lib } \Lambda \\
\langle \mathcal{T} \in \mathit{TypeName} \rangle &::= \textit{the set of type names in lib } \Lambda
\end{aligned}
$$

(b) Types

$$
\mathit{UpBound} : \mathit{TypeVariable} \longrightarrow \mathit{Type} \qquad \mathit{UpBound}(\phi : t) = t
$$

$$
\mathit{type} : \mathit{Class} \longrightarrow \mathit{Type} \qquad \mathit{type}(\texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{extends}\ \overline{t} : \overline{v}) = C\texttt{<}\overline{\alpha}\texttt{>} : \overline{t}
$$

(c) Auxiliary functions

Fig. 4. The syntax and the types in the API-IR.

In what follows, we use the vector notation to represent a sequence of elements. For example, $\overline{t}$ means a sequence of t elements.

API-IR is parameterized by an API denoted as $\Lambda \in \mathit{API}$. An API is a set consisting of (polymorphic) classes, (polymorphic) functions, and field/variables. A class defined as $\texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{extends}\ \overline{t} : \overline{v}$ defines a sequence of formal type variables $\overline{\alpha}$, and extends a sequence of types $\overline{t}$. The body of a class represented by $\overline{v}$ contains other API members, including nested class definitions. A function $\texttt{fun}\ m\texttt{<}\overline{\alpha}\texttt{>}(\overline{x : t_1}) : t_2$ with name m: (1) introduces a set of type variables $\overline{\alpha}$, (2) takes a sequence of formal parameters, and (3) outputs a value of type t_2. A class or a function is considered *polymorphic* only when the sequence $\overline{\alpha}$ in their syntax is not empty. A program in API-IR is a sequence of expressions that use components from the given API Λ. An expression is either a constant of type $t \in \mathit{Type}$ denoted as $\mathit{constant}(t)$, a local variable definition, a field access, or a function call. The expression ϵ represents an empty expression used to model the invocation of top-level methods and fields, e.g., methods with no explicit receiver.

The type system in API-IR is nominal. The set of types includes the usual $\bot$ and $\top$ types, or a type variable $\phi : t$ with an upper bound t. A class type $\mathcal{T}\texttt{<}\overline{\alpha}\texttt{>} : \overline{t}$ is labeled with a name $\mathcal{T}$, a sequence of type variables $\overline{\alpha}$, and a sequence of supertypes $\overline{t}$. A class type is derived from a class $c \in \mathit{Class}$ based on function $\mathit{type} : \mathit{Class} \longrightarrow \mathit{Type}$ shown in Figure 4. When the sequence $\overline{\alpha}$ is not empty, the class type $\mathcal{T}\texttt{<}\overline{\alpha}\texttt{>} : \overline{t}$ is referred to a *type constructor*. The type system of API-IR also features a type instance ($N\texttt{<}\overline{t}\texttt{>}$) that instantiates a type constructor N with a sequence of type arguments $\overline{t}$. Finally, the type system of API-IR follows the usual typing rules shown in the extended version of our paper [Sotiropoulos et al. 2023b], using the judgment $\Lambda \vdash e : t$. In our setting, the given API Λ acts as our global typing environment, and the type of an empty expression is $\bot$ ($\Lambda \vdash \epsilon : \bot$). In the text and examples, we use the shorthand (1) ϕ for $\phi : \top$, (2) $\mathcal{T} : \overline{t}$ for $\mathcal{T}\texttt{<}\emptyset\texttt{>} : \overline{t}$, (3) $\mathcal{T}$ for $\mathcal{T} : \top$, and (4) $\mathcal{T}\texttt{<}\overline{t_2}\texttt{>}$ for $(\mathcal{T}\texttt{<}\overline{\alpha}\texttt{>} : \overline{t_1})\texttt{<}\overline{t_2}\texttt{>}$.

Our language also defines the usual type substitution and type unification operations. A type substitution $\sigma \in \Sigma$ is a mapping that replaces all occurrences of a type variable α with a given type t. We use the symbol σt to denote the application of a substitution σ on type t. In this work, type unification (unify: $\mathit{Type} \times \mathit{Type} \longrightarrow \Sigma$) takes two types ($t_1, t_2 \in \mathit{Type}$) and identifies a substitution σ so that the type σt_2 is equal subtype of t_1. In the following, the symbol $<:$ indicates the subtype relation, and $\mathit{UpBound}(\alpha)$ gives the upper bound of a type variable α as shown in Figure 4.

Definition 3.1 (Validity of type substitution). A type substitution $\sigma \in \Sigma$ is called *valid* when $\forall \alpha \in Dom(\sigma).\ \sigma(\alpha) <: UpBound(\alpha)$.

This definition expresses that a type substitution is considered valid, when every type variable in the substitution is instantiated with a type that respects the upper bound of the type variable. For example, the substitution $s = [\alpha \mapsto t_1]$ is valid when α has the upper bound $\top$, because $t_1 <: \top$. On the contrary, the substitution is invalid when α is bounded to type t_2, and t_1 is not a subtype of t_2.

Definition 3.2 (Subsumption). Consider two type substitutions $\sigma_1, \sigma_2 \in \Sigma$. We say that substitution σ_1 subsumes σ_2, denoted as $\sigma_1 \sqsubseteq \sigma_2$, when $\forall \alpha \in Dom(\sigma_1).\ \sigma_1(\alpha) = \sigma_2(\alpha)$.

The subsumption relation holds between two type substitutions, when all type variables in a substitution σ_1 are instantiated with exactly the *same* type as in another substitution σ_2. The subsumption relation is reflexive, and for an empty substitution ϵ, we have $\forall \sigma \in \Sigma.\epsilon \sqsubseteq \sigma$.

Definition 3.3 (Type decomposition). Type decomposition ($Type \longrightarrow \Sigma \times Type$) is an operation that decomposes a given type $t_1 \in Type$ into a substitution σ and another type $t_2 \in Type$, so that $\sigma t_2 = t_1$. It is defined as:

$$decompose(t) = \langle [\overline{\alpha} \mapsto \overline{t_2}], \mathcal{T}\text{<}\overline{\alpha}\text{>} : \overline{t_1} \rangle \qquad \text{if } t = (\mathcal{T}\text{<}\overline{\alpha}\text{>} : \overline{t_1})\text{<}\overline{t_2}\text{>}$$
$$decompose(t) = \langle \epsilon, t \rangle \qquad \text{otherwise}$$

In essence, type decomposition allows us to decompose a type instance $N\text{<}\overline{t}\text{>}$ into (1) the type constructor N, and (2) the type substitution that replaces all formal type variables in N with the provided type arguments $\overline{t}$. For example, consider the type constructor $N = \mathcal{T}\text{<}\alpha\text{>} : \top$ and its type instance $t = N\text{<}t_2\text{>}$. In this example, $decompose(t)$ returns $\langle [\alpha \mapsto t_2], N \rangle$. For a non polymorphic type t, type decomposition simply returns an empty substitution and the input type t.

3.2 API Graph

We define an API directed graph as $G = (V, E)$, where V is the set of nodes corresponding to either a type $t \in Type$ or an API definition $d \in Def$ (i.e., a method or a field), $E \subseteq V \times V \times L$ is the set of edges whereas $L = \Sigma$ is the set of edge labels (representing the set of valid type substitutions). To construct the API graph we examine a given API $\Lambda \in API$ and proceed as follows.

- Iterate over the set of classes in topological order with regards to their inheritance chain. Convert every class c into a type t based on function *type* defined in Figure 4, and add the resulting type t into the graph. Then, we iterate over each member (i.e., a function or a field) $d \in Def$ belonging to class c and proceed as follows.
- Add node d to the API graph.
- Add edge $t \xrightarrow{\epsilon} d$ to the API graph, if d is an *instance* method or a field of the class c and $t = type(c)$.
- Add edge $d \xrightarrow{\sigma} r'$ to the API graph, if r is the return type of d and $decompose(r) = \langle \sigma, r' \rangle$ according to Definition 3.3.

Conceptually, the set of edges determines the following relationships. The edge $t \xrightarrow{\epsilon} d$ indicates that the definition d is applied to a value of type t (receiver type). The edge $d \xrightarrow{\sigma} r'$ denotes that the application of the API definition d returns the type given by $\sigma r'$. For example, consider a method m whose return type is `List<String>`. In this scenario, we add the following edge to the API graph: $m \xrightarrow{\alpha \mapsto \texttt{String}}$ `List`$\text{<}\alpha\text{>}$, where the target node of the edge is the type constructor `List`. In a similar manner, when the return type of method m is a non-polymorphic type, such as `Int`, *decopose(`Int`)* yields $\langle \epsilon, \texttt{Int} \rangle$. Therefore, we add the edge $m \xrightarrow{\epsilon}$ `Int` to the graph. In our examples, we use the shorthand $v_1 \longrightarrow v_2$ for $v_1 \xrightarrow{\epsilon} v_2$.

```
1  class Utils {
2    static <X> List<X> createList();
3  }
4  class List<T> {
5    List(int size);
6    boolean add(T elem);
7    Set<T> toSet();
8  }
9  class Set<E> {
10   Set(int size);
11   boolean add(E elem);
12   List<E> toList();
13 }
```

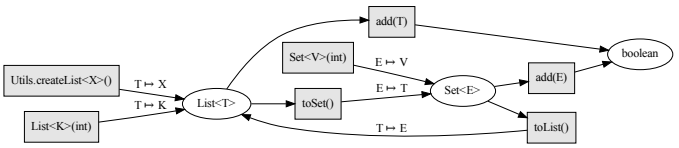


Fig. 5. An example API written in Groovy and its API graph. Oval nodes represent types, while rectangular nodes denote definitions (e.g., methods).

The API graph is directly inspired by the *signature graph* introduced in the program synthesis work of Mandelin et al. [2005]. In a similar manner to our API graph, the signature graph encodes API components as unary functions taking one input type (receiver type), and producing an output type (return type). The intuition behind this is the identification of chains of method calls/field accesses (e.g., `x.f1.m1().f2...`) that lead to a specific type t_{out}. This can be achieved by simply querying the graph for paths between an input type t_{in} and the target type t_{out} using standard graph reachability algorithms, such as Dijkstra's algorithm.

However, the presence of polymorphic components complicates this straightforward approach. Our API graph addresses this by refining the signature graph: we label edges with type substitutions, allowing us to handle parametric polymorphism without compromising the size of the API graph. Our detailed approach for identifying chains of method calls/field accesses in the presence of parametric polymorphism is explained later in Section 3.4.

Example: Figure 5 shows an example API (written in Groovy) and its API graph. The API graph contains seven definitions (i.e., methods) depicted with gray color. Three definitions have no incoming edges, as their application does not require a receiver object. Two of them (i.e., `List<K>(int)`, `Set<V>(int)`) stand for the constructors of the generic classes `List` and `Set` respectively (lines 4, 9), while one of them is for the static polymorphic method of class `Utils`. Finally, the example API graph includes three types represented by oval nodes, two of which denote the type constructors `List` and `Set`, while one type node corresponds to `boolean`.

3.3 API Enumeration Problem Formulation

Having presented the notion of API graph (Section 3.2), we now formulate the problem of API enumeration. API enumeration systematically explores all the unique typing combinations that can be used to invoke a particular API component, such as a function or a field. The intuition is that invoking an API component through different typing patterns lets us exercise the implementation of many type-related operations in the compiler, including, subtyping rules, or method resolution. In what follows, we use an API graph as our typing environment. In fact, an API graph can help type expressions that use API components, because the type signatures of every API entity is included in G. For example, hereafter, $G \vdash e.f : t$ means that the type of the field access $e.f$ is t. This typing process is based on the type signature of field f found in G.

We first introduce *abstract typed expressions*, an abstraction over the domain of expressions defined in API-IR (Figure 4). An abstract typed expression is given by:

$$\langle \hat{e} \in \hat{Expr} \rangle ::= [t] \;\mid\; [t].f \;\mid\; [t_1].m\langle \bar{t} \rangle(\overline{[t_2]}) \;\mid\; \texttt{local var}\ x : t = \hat{e}$$

The notation $[t]$ represents an *inhabitant* of type t [Urzyczyn 1997], that is any concrete expression $e \in Expr$ whose type is t. An abstract typed expression hides the contents and the value of a

concrete expression e, and considers only the type of e. For example, the abstract expression $[t].f$ indicates a field f accessed through *any* expression of type t. We encode every abstract typed expression $\hat{e}$ using *typing sequences*. A typing sequence succinctly captures the types found within the holes of abstract typed expressions. A typing sequence is denoted by the symbol $[\![\,]\!]$:

$$\begin{aligned}
[\![\,[t]\,]\!] &\rightarrow \langle t \rangle \\
[\![\,[t].f\,]\!] &\rightarrow \langle t, \bot \rangle \\
[\![\,[t_1].m\overline{<t>}(\overline{[t_2]})\,]\!] &\rightarrow \langle t_1, \overline{t_2} \rangle \\
[\![\,\texttt{local var}\ x : t = \hat{e}\,]\!] &\rightarrow [\![\,\hat{e}\,]\!] \cdot t
\end{aligned}$$

In the preceding rules, the symbol $\cdot$ means appending an element to the end of a sequence. A singleton sequence consisting of type t represents an inhabitant of t. When a typing sequence contains more than two types, the first element of the sequence stands for the receiver type of a field access/method call, and the remaining elements correspond to the parameter types of the application (if any). Finally, in the case of local variable definitions, the final element of the typing sequence is the expected type of the entire abstract expression found on the right-hand side.

We define a concretization function γ that allows us to map a typing sequence and an API definition d into a set of concrete expressions written in API-IR under a given type substitution σ.

Definition 3.4. Let $\gamma : G, \mathit{Def}, \mathit{Type} \times \cdots \times \mathit{Type} \times \Sigma \longrightarrow \mathcal{P}(\mathit{Expr})$ such that:

$$\begin{aligned}
\gamma(G, \bot, \langle t \rangle, \sigma) &= \{e \mid G \vdash e : t\} \\
\gamma(G, \texttt{var}\ f : t, \langle t, \bot \rangle, \sigma) &= \{e.f \mid G \vdash e : t\} \\
\gamma(G, \texttt{fun}\ m\overline{<\alpha>}(\overline{x : p}) : t, \langle r, \overline{p'} \rangle, \sigma) &= \left\{ e_1.m\overline{<t>}(\overline{e_2}) \;\middle|\; \begin{array}{l} G \vdash e_1 : r, G \vdash \overline{e_2} : \overline{p'}, \\ \overline{t} = (\sigma(\alpha_i))_{i=1}^{n} \end{array} \right\} \\
\gamma(G, d, s \cdot t, \sigma) &= \{\texttt{local var}\ x : t = e \mid e \in \gamma(G, d, s, \sigma)\}
\end{aligned}$$

The function γ concretizes abstract typing sequences and definitions into specific expressions that are typed under a given API graph and a type substitution. When no API entity is provided to γ (i.e., its second parameter is $\bot$), the function returns all the inhabitants of type t. This translates to every expression $e \in \mathit{Expr}$ that satisfies $G \vdash e : t$. Conversely, if an API component d is provided, γ yields all the possible expressions that invoke the definition d with respect to the given typing sequence s. For example, consider the field $\texttt{var}\ f : t$ and the typing sequence $s = \langle r, \bot \rangle$. In this case, the function γ gives all accesses of field f via every inhabitant of type r. When encountering a polymorphic function, γ maps every formal type variable of the function into actual type arguments attached to the resulting method calls based on the provided type substitution, in particular $(\sigma(\alpha_i))_{i=1}^{n}$. Later, in Section 3.4, we present an under-approximation of γ called $\hat{\gamma}$, introduced to address the practical concerns associated with exhaustively enumerating all potential expressions in γ.

Definition 3.5 (API typing sequence). Given an API component $d \in \mathit{Def}$, the sequence s_d is called an *API typing sequence* of d when there is *an abstract typed expression* $\hat{e}$, such that (1) $[\![\,\texttt{local var}\ x : t = \hat{e}\,]\!] = s_d$, and (2) the abstract expression $\hat{e}$ invokes the component d. We say that an expression $e \in \mathit{Expr}$ *realizes* the API typing sequence s_d under the substitution σ, if $e \in \gamma(G, d, s_d, \sigma)$.

Based on Definition 3.5, an API typing sequence s_d reveals two key details: First, *how* a set of types are combined together to invoke and use a certain API component d (represented by all elements of s_d except the last). Second, *what* is the expected type that derives from the usage of d (indicated by the last element of s_d).

Example: Consider two typing sequences for the method d = `add(T)` defined in Figure 5 (line 6): $s_{d_1} = \langle \bot, \texttt{int}, \texttt{boolean} \rangle$ and $s_{d_2} = \langle \texttt{List<Int>}, \texttt{int}, \texttt{boolean} \rangle$. Listing 1 contains three expressions that realize s_{d_1} and s_{d_2}. The first expression (although type incorrect) realizes s_{d_1}, because the first element of s_{d_1} (denoted as $s_{d_1}\downarrow_1$) is $\bot$, which represents the absence of receiver. The last two expressions of the listing realize the same API typing sequence s_{d_2}, because the type of both `new List<Int>(10)` and `Utils.<Int>createList()` is $s_{d_2}\downarrow_1$= `List<Int>`.

```
1 boolean x = add(1);
2 boolean y = Utils.<Int>createList().add(1);
3 boolean z = new List<Int>(10).add(1);
```

Listing 1. Expressions that realize typing sequences of method `List.add(T)`.

Definition 3.6 (API signature). API signature ($G \times Def \longrightarrow Type \times \cdots \times Type$) is a function that maps an API graph $G = (V, E)$ and one API definition $d \in V$ to a typing sequence as follows:

$$
\begin{aligned}
sig(G, \texttt{var}\ x{:}\ t) &= \langle r, \bot, t \rangle && \text{if } r \xrightarrow{l} d \in E \\
sig(G, \texttt{var}\ x{:}\ t) &= \langle \bot, \bot, t \rangle && \text{if } d \text{ has no incoming edges in } G \\
sig(G, \texttt{fun}\ m\texttt{<}\overline{\alpha}\texttt{>}(\overline{x : p}) : t) &= \langle r, \overline{p}, t \rangle && \text{if } r \xrightarrow{l} d \in E \\
sig(G, \texttt{fun}\ m\texttt{<}\overline{\alpha}\texttt{>}(\overline{x : p}) : t) &= \langle \bot, \overline{p}, t \rangle && \text{if } d \text{ has no incoming edges in } G
\end{aligned}
$$

For example, the API signature of the `Utils.createList()` method of Figure 5 (line 2) is $\langle \bot, \bot, \texttt{List<}X\texttt{>} \rangle$. The method is static and takes no parameters. That is why the first element (receiver type) and the second element (parameter type) of the resulting sequence are $\bot$.

Definition 3.7 (Well-typed API typing sequence). Given an API graph G, a definition $d \in Def$, and a type substitution $\sigma \in \Sigma$, we say that an API typing sequence $s_d = \langle r, \overline{p}, t \rangle$ is *well-typed* under G and σ, if $sig(G, d) = (r', \overline{p'}, t')$ and $r <: \sigma r'$, $\overline{p} <: \sigma\overline{p'}$, and $t >: \sigma t'$.

In essence, this definition captures precisely the notion that given an API component d, a typing sequence s_d is well-typed under σ, if s_d describes a usage of d with: (1) a receiver whose type r is a subtype of the formal receiver type extracted from the signature of d ($r <: \sigma r'$), and (2) arguments (if present) whose types are subtypes of the formal parameter types of d. Finally, the expected type of d's application should be any supertype of d's formal return type.

Theorem 3.8. *Consider an API component $d \in Def$, one well-typed typing sequence s_d of d, and a substitution σ. The programs derived from s_d, that is $\gamma(G, d, s_d, \sigma)$, are well-typed.*

Proof. This follows straightforwardly from Definition 3.4 and the typing rules of API-IR. □

Example: Consider again the method d = `add`(T) defined in the `List` class of our example API (Figure 5, line 6). Also, consider a substitution $\sigma = [T \mapsto \texttt{Int}]$. The API typing sequence $s_d = \langle \texttt{List<Int>}, \texttt{Int}, \texttt{boolean} \rangle$ is well-typed under σ, because (1) `List<Int>` $<:$ $[T \mapsto \texttt{Int}]$`List<`T`>`, (2) `Int` $<:$ $[T \mapsto \texttt{Int}]T$, and (3) `boolean` $>:$ $[T \mapsto \texttt{Int}]$`boolean`. Hence, all the concrete expressions that come from s_d, such as the last two expressions of Listing 1, are well-typed.

Definition 3.9 (API enumeration). Consider an API graph G and an API component d included in G. For every type substitution $\sigma \in \Sigma$, API enumeration computes exhaustively a set of *well-typed* API typing sequences S_d under σ such that for every $s_d \in S_d$, we obtain a program $p \in \gamma(G, d, s_d, \sigma)$.

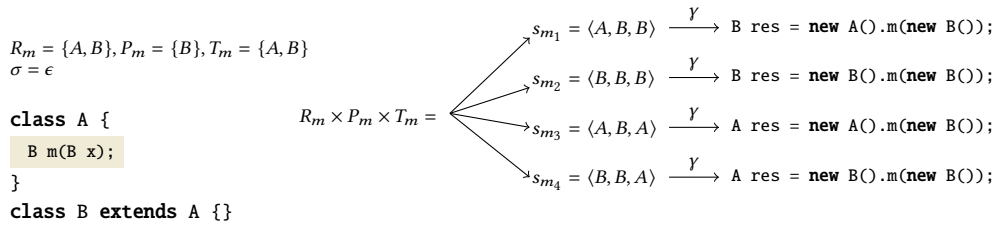


Fig. 6. Enumerating well-typed API typing sequences for method `B m(B x)` using substitution $\sigma = \epsilon$. The set R_m contains the subtypes of the formal receiver type of `m`, P_m contains the subtypes of the first formal parameter type of `m`, while T_m contains the supertypes of the return type of `m`. Using the function γ, each typing sequence leads to programs that invoke method `m` via a unique typing pattern. This, in turn, triggers various type-related operations in the compiler.

The algorithm of API enumeration first enumerates *all* valid type substitutions that instantiate the type variables of an API definition. Given that in the presence of polymorphic types, substitutions are infinite (consider `List<List<List<...>>>`), API enumeration can be set to produce all type substitutions up to a certain depth. This is done by instantiating each type variable with every concrete type found in a given API, including type instances. For example, when it comes to enumerating all type substitutions of length two, the result set includes a type of the form `List<List<Int>>`, but not `List<List<List<Int>>>` (as the depth in this case is three). Based on a substitution σ and a library component d with signature $\langle r, p_1 \dots p_n, t \rangle$, API enumeration generates the Cartesian product of sets $R, P_1, \dots P_n, T$, where R is the set consisting of subtypes of σr, P_i is the set consisting of subtypes of σp_i (with $1 \leq i \leq n$), and T is the set containing the supertypes of σt. API enumeration is exponential in terms of the number of types included in the signature of d. For example, consider a substitution σ and an API signature that specifies four types: $\langle r, p_1, p_2, t \rangle$. Each of σr, σp_1, and σp_2 contains ten subtypes, while there are also ten supertypes of σt. API enumeration gives $10^4 = 10,000$ well-typed typing sequences under σ.

Example: Figure 6 illustrates the concept of API enumeration for the non-polymorphic method `B m(B x)`. Based on the signature of method `m`, we compute the sets R_m, P_m, and T_m, and then we take their Cartesian product. Each element of this product corresponds to a well-typed API typing sequence, which eventually leads to concrete programs that invoke method `m` through a unique typing combination. For example, the type of the receiver in the first program is `A`, while the type of the receiver in the second program is type `B`. API enumeration for polymorphic components follows a similar process by enumerating well-typed API sequences under every valid type substitution.

Enumerating ill-typed API typing sequences: An API typing sequence s_d of a definition d is considered ill-typed when at least one enclosing type of the sequence is *incompatible* with the signature of d. Consider an API graph G, a definition d, and a substitution σ. An API typing sequence $s_d = \langle r, \overline{p}, t \rangle$ is ill-typed under G and σ, if $sig(G, d) = \langle r', \overline{p'}, t' \rangle$ and $r \not<: \sigma r'$ and $r \not>: \sigma r'$, or $\overline{p} \not<: \sigma \overline{p'}$, or $t \not>: \sigma t'$. Notice that a receiver type is incompatible when it is neither a supertype nor a subtype of the formal receiver type. This happens to avoid constructing a well-typed sequence due to inheritance. In a similar manner to Theorem 3.8, the programs derived from ill-typed API typing sequences are ill-typed. Consider again the method $d =$ `List.add(T)` of Figure 5. Under the substitution $\sigma = [T \mapsto \texttt{Int}]$, the sequence $s_d = \langle$`List<Int>`, `String`, `boolean`$\rangle$ is ill-typed, because `String` $\not<:$ $[T \mapsto \texttt{Int}]T$. An expression that realizes this sequence is deemed ill-typed:

```
boolean x = Utils.<Int>createList("str value") // String is incompatible to Int
```

```
1  class Utils {
2    static <X, Y> Map<X, Y> mapOf();
3    static Map<String, String>
         mapOfStrs();
4  }
5  class Map<K, V> {
6    Set<K> keySet();
7  }
8  class Set<E> {}
```

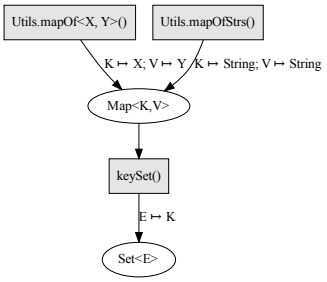


Fig. 7. An API relying on parametric polymorphism and its API graph.

3.4 Concretization of API Typing Sequences

Every API typing sequence that arises from API enumeration is converted into a concrete test program. We present a method that *under-approximates* the concretization function γ presented in Definition 3.4. To identify type inhabitants, our method examines the API graph and enumerates all paths that reach a certain type t. Each path represents an inhabitant consisting of a chain of method calls or field accesses. In the presence of parametric polymorphism, some paths are infeasible, as the underlying type variables found in these paths are instantiated with incompatible types that do not ultimately lead to a given type t. We explain the details on how we determine the feasibility of each path and properly instantiate the corresponding type variables.

Finding type inhabitants via API graph reachability: In the simplest scenario, where parametric polymorphism is not present, finding inhabitants of a type t is given by standard graph reachability algorithms that compute the set of paths that reach node t [Mandelin et al. 2005]. However, parametric polymorphism complicates the process, making the simple path search yield incorrect results. To illustrate this, consider the API and its corresponding API graph of Figure 7. Now assume that we want to find an inhabitant of polymorphic type `Set<Int>`. The naïve path search identifies two potential paths that reach the type constructor `Set`: path (1) and path (2). The expression `mapOfStrs().keySet()`, which originates from path (1) `Utils.mapOfStrs()` $\rightarrow$ `Map<`K, V`>` $\rightarrow$ `keySet()` $\rightarrow$ `Set<`E`>`, leads to an incompatible type `Set<String>`. In contrast, path (2) `Utils.mapOf<`X, Y`>()` $\rightarrow$ `Map<`K, V`>` $\rightarrow$ `keySet()` $\rightarrow$ `Set<`E`>` involves a polymorphic function (`Utils.mapOf`). The expression from path (2) is correctly identified as an inhabitant of `Set<Int>` *only* if the type variable X defined in `Utils.mapOf` is instantiated with type `Int`. To identify infeasible paths and properly instantiate type variables of polymorphic components, we now present a refined path search approach that deals with parametric polymorphism.

Every path in an API graph may consist of polymorphic API components. Every polymorphic component introduces some type variables. In the above example, path (1) introduces type variables K, V and E. Similarly, path (2) introduces type variables, X, Y, K, V, and E. The labels (aka substitutions—recall Section 3.2) found on top of each edge indicate instantiations of the corresponding type variables. However, a path may also contain free type variables whose instantiation is not given by such substitutions. For example, path (1) of Figure 7 contains no free type variables, as both type variables K and V are instantiated with `String` while type variable E is instantiated with type K. On the other hand, path (2) involves two free type variables, namely, X and Y. These type variables do not participate in the left-hand side of a substitution. From now onwards, given a path p, we use the notation *TypeVar(p)* to obtain the set of type variables of path p. To take the set of free type variables of a path p, we use the symbol $FV(p)$.

Algorithm 1: Algorithm for finding paths that form inhabitants of type t

```
1  fun find_API_paths(G, t)=
2      ⟨σ, t'⟩ ← decompose(t)
3      paths ← {⟨⟨t⟩, ⊥⟩}
4      for p ∈ Paths(G, t') do
5          sub ← gather all substitutions of path p
6          σ' ← perform constant propagation on σ based on subs
7          t_2 ← σ't'
8          σ_1 ← unify(t, t_2)
9          if σ_1 = ϵ then continue
10         σ_2 ← ∀α ∈ FV(p). instantiate α   if α ∉ σ_1
11         σ' ← σ_1 ∪ σ_2
12         paths ← paths ∪ {⟨p, σ'⟩}
13     return paths
```

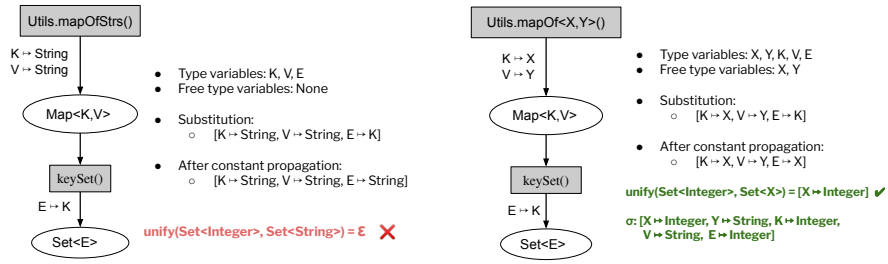
Definition 3.10. Let a type t and its decomposition $decompose(t) = \langle \sigma, t' \rangle$. Given an API graph G, we say that a path p *forms* an inhabitant of type t, when (1) the path p leads to target node t', and (2) there is a *a valid* substitution σ', such that $\forall \alpha \in TypeVar(p).\ \alpha \in Dom(\sigma')$ and $\sigma \sqsubseteq \sigma'$. Our goal is then to find the set of paths P so that every path $p \in P$ forms an inhabitant of type t.

The definition above summarizes the problem of path search, which aims to find type inhabitants even in the presence of polymorphic types. Given a target type t, we initially decompose it into a substitution σ and a type t' according to Definition 3.3. Then, we search the API graph G to find the set of paths that reach node t'. A path p forms a type inhabitant if there is a *valid* substitution σ' (recall Definition 3.1) that includes every type variable in *TypeVar(p)*, and subsumes the original substitution σ according to Definition 3.2 ($\sigma \sqsubseteq \sigma'$). Conceptually, the substitution σ that we obtain after performing the type decomposition on t, *constrains* the instantiation of some type variables found in path p. Therefore, we need to ensure that the substitution σ' contains compatible assignments for the type variables included in σ. If no such valid substitution σ' exists, then we consider the path as *infeasible*: it cannot yield an expression of the target type t.

Example: Consider again the API graph of Figure 7 and the problem of finding inhabitants of type `Set<Int>`. This type is decomposed into the substitution $\sigma = [E \mapsto \texttt{Int}]$ and the type constructor `Set<E>`. Again, there are two paths that reach the type constructor `Set`. Path (1) is infeasible as the only substitution σ that stems from this path is $\sigma' = [K \mapsto \texttt{String}, V \mapsto \texttt{String}, E \mapsto \texttt{String}]$, which is incompatible with σ, because $\sigma(E) \neq \sigma'(E)$. On the other hand, path (2) is feasible because there exists a substitution σ' such that $\sigma \sqsubseteq \sigma'$. Specifically, we have $\sigma' = [X \mapsto \texttt{Int}, Y \mapsto \texttt{String}, K \mapsto \texttt{Int}, Y \mapsto \texttt{String}, E \mapsto \texttt{Int}]$, as $\sigma(E) = \sigma'(E) = \texttt{Int}$.

Algorithm: Algorithm 1 outlines our method for identifying paths that form inhabitants of a given type. We describe the algorithm in the context of finding inhabitants of `Set<Int>` based on the API graph shown in Figure 7. The algorithm takes as input an API graph G and a type t, and returns a set of pairs, where each pair consists of a path and a substitution. The algorithm starts with decomposing the given type t into a substitution σ and another type t'. Then, it enumerates all *acyclic* paths that reach node t'. In our example, the are two paths that reach the type constructor `Set`—each of those paths is shown individually in Figure 8.

It is now time to determine which of the available paths are feasible. For every path p, the algorithm examines and gathers all substitutions (i.e., edge labels) found in p (line 5). For example, the first path (Figure 8a) contains three substitutions (e.g., $[K \mapsto \texttt{String}, V \mapsto \texttt{String}, E \mapsto T]$), while the second path (Figure 8b) includes the following substitutions: $[K \mapsto X, V \mapsto Y, E \mapsto K]$.



(a) An infeasible path.

(b) An inhabitant of `Set<Int>`.

Fig. 8. Determining which of the paths in the API graph of Figure 7 form an inhabitant of type `Set<Int>`.

Algorithm 2: Algorithm for converting paths into concrete expressions

```
1 fun inhabitants(G, t)=
2 |  inhabitants ← ∅
3 |  for ⟨p, σ⟩ ∈ find_API_paths(G, t) do
4 |  |  inhabitants ← inhabitants ∪ 2expr(p, σ)
5 |  return inhabitants
```

Next, the algorithm performs constant propagation on the collected substitutions and creates a new substitution σ' (line 6). For example, the substitution of path (1) becomes $\sigma' = [K \mapsto \texttt{String}, V \mapsto \texttt{String}, E \mapsto \texttt{String}]$, while the substitution when handling path (2) is $[K \mapsto X, V \mapsto Y, E \mapsto X]$. Using the new substitution σ', we create a new type t_2 that is given by $\sigma' t'$. For example, for the path of Figure 8a t_2 is `Set<String>`, while t_2 stands for `Set<X>` when dealing with the path of Figure 8b. Then, the algorithm unifies the type t_2 with the target type t (line 8). Essentially, this type unification operation identifies assignments for the free type variables of path p so that the resulting type of the path expression is compatible with the target type t. If the two types are not unifiable, the path is considered infeasible, and the algorithm proceeds to the next iteration (line 9). For example, the type t_2 = `Set<String>` of Figure 8a is not unifiable with the target type `Set<Int>`. This means that path (1) is an infeasible path, not leading to an expression of type `Set<Int>`. When the types are unifiable, the outcome of type unification (σ_1—line 8) instantiates some of the free type variables of p. For example, consider again Figure 8b: the algorithm unifies types `Set<Int>` and `Set<X>` by returning the substitution $\sigma_1 = [X \mapsto \texttt{Int}]$.

In the final step, the algorithm instantiates any free type variables of p, not instantiated by the previous unification operation (line 10). Such free type variables do *not* affect the type of the underlying expression and therefore, the algorithm is free to instantiate them with any valid type that respects their upper bounds. For example, Y is another free type variable that stems from the path of Figure 8b. The algorithm instantiates it with a randomly selected type, e.g., `String`. The final substitution σ' is then the union of substitutions σ_1 and σ_2 (line 11). The final substitution and the corresponding path p are added to the set P (line 12). For example, the path of Figure 8b forms an inhabitant of type `Set<Int>` if using the substitution $[X \mapsto \texttt{Int}, Y \mapsto \texttt{String}, K \mapsto \texttt{Int}, V \mapsto \texttt{String}, E \mapsto \texttt{Int}]$.

Termination: The function *Paths(G, t)* on line 4 of Algorithm 1 enumerates all paths in graph G that reach the target node t. Since an API graph may contain cycles, there might be an infinite number of paths. To tackle this, we consider only acyclic paths, which are guaranteed to be finite.

Converting a path into an expression: Having computing the set of paths that form inhabitants of a type t, we employ Algorithm 2 to convert every path into a concrete expression. The algorithm

employs the function *2expr*, which given an API graph G, and a substitution σ, converts a path p into an expression as follows:

$$
\begin{aligned}
2expr(G, p \cdot \texttt{fun}\ \ m\texttt{<}\overline{\alpha}\texttt{>}(\overline{x : p}) : t, \sigma) &= 2expr(G, p, \sigma).m\texttt{<}\overline{t}\texttt{>}(\overline{e}) && \text{where}\\
&\quad \overline{e} = e_i \in \texttt{inhabitants}(G, \sigma p_i), \forall i \in \{1 \dots n\}\\
&\quad \overline{t} = (\sigma(\alpha_i))_{i=1}^{n}\\
2expr(G, p \cdot \texttt{var}\ \ f : t, \sigma) &= 2expr(G, p, \sigma).f\\
2expr(G, d, \sigma) &= constant(d) && \text{if } d \in Type\\
2expr(G, p \cdot d, \sigma) &= 2expr(G, p, \sigma) && \text{if } d \in Type\\
2expr(G, [], \sigma) &= \epsilon
\end{aligned}
$$

For a path consisting of a single type node, the function *2expr* simply returns a constant expression $constant(t)$, which is typically translated into a cast null expression (e.g., `???.asInstanceOf[T]` in Scala). Type nodes found in intermediate positions within a path are ignored. To illustrate this path conversion process, consider the path of Figure 8b. When using the substitution $[X \mapsto \texttt{Int}, Y \mapsto \texttt{String}]$, this path results in the expression `Utils.mapOf<Int, String>().keySet()`.

Realization of γ: Based on the aforementioned definitions and algorithms, we finally present the realization of the concretization function γ, previously shown in Definition 3.4. The function γ is an abstract concept that generates *all* API-IR expressions that originate from a given typing sequence and an API graph. Our realization function called $\hat{\gamma}$ under-approximates the behavior of γ, meaning that for a given API graph G, a typing sequence s_d, an API component d, and a type substitution, we have $\hat{\gamma}(G, s_d, d, \sigma) \subseteq \gamma(G, s_d, d, \sigma)$. To do so, the $\hat{\gamma}$ function relies on Algorithm 2 as follows:

Definition 3.11. Let $\hat{\gamma} : G, Def, Type \times \cdots \times Type \times \Sigma \longrightarrow \mathcal{P}(Expr)$ such that:

$$
\begin{aligned}
\hat{\gamma}(G, \bot, t, \sigma) &= \{e \mid e \in \texttt{inhabitants}(G, t)\}\\
\hat{\gamma}(G, \texttt{var}\ f : t, \langle t, \bot \rangle, \sigma) &= \{e.f \mid e \in \texttt{inhabitants}(G, t)\}\\
\hat{\gamma}(G, \texttt{fun}\ m\texttt{<}\overline{\alpha}\texttt{>}(\overline{x : p}) : t, \langle r, \overline{p'} \rangle, \sigma) &= \left\{ e_1.m\texttt{<}\overline{t}\texttt{>}(\overline{e_2}) \;\middle|\; \begin{array}{l} e_1 \in \texttt{inhabitants}(G, r)\\ \overline{e_2} = e_{2_i} \in \texttt{inhabitants}(G, p_i'),\\ \forall i \in \{1 \dots n\}, \overline{t} = (\sigma(\alpha_i))_{i=1}^{n} \end{array} \right\}\\
\hat{\gamma}(G, d, s \cdot t, \sigma) &= \{\texttt{local var}\ x : t \mid e \in \hat{\gamma}(G, d, s, \sigma)\}
\end{aligned}
$$

Theorem 3.12 (Soundness). *Given an API graph G, an API definition $d \in Def$, a typing sequence s, and a substitution σ, if $\hat{\gamma}(G, d, s, \sigma)$ returns E, then $E \subseteq \gamma(G, d, s, \sigma)$.*

Proof. The theorem follows straightforwardly from lines 8–9 of Algorithm 1 as we consider only paths that result in a type unifiable with the target type t. □

Theorem 3.13 (Completeness). *Given an API graph G, an API definition $d \in Def$, a typing sequence s, and a substitution σ, $\hat{\gamma}(G, d, s, \sigma) \neq \emptyset$.*

Proof. This can be proven by Algorithm 1. The algorithm always returns a non-empty solution, even if there is no path to the target type t. In this case, the algorithm returns a singleton path containing the given type t (line 3, Algorithm 1), which is in turn translated into a constant expression according to function *2expr*. □

3.5 Type Erasure

Next, an API-IR expression obtained from an API typing sequence (Section 3.4) is passed as input to the type erasure process. The objective is to remove types from an API-IR expression, while

$$\textsc{constant}\quad \frac{}{erasure(G, constant(t), t') = constant(t)} \qquad \textsc{field access}\quad \frac{}{erasure(G, e.f, t) = erasure(G, e, \bot).f}$$

$$\textsc{method call}\quad \frac{e = e_1.m\langle\overline{t}\rangle(\overline{e_2}) \qquad G \vdash \overline{e_2} : \overline{p} \qquad \forall \alpha \in TypeVar(m).\textit{the type argument of } \alpha \textit{ can be safely erased from } e \textit{ when the target type is } k}{erasure(G, e, k) = erasure(G, e_1, \bot).m(erasure(G, \overline{e_2}, \overline{p}) \ldots)}$$

$$\textsc{local var}\quad \frac{}{erasure(G, \texttt{local var } x : t = e, t') = \texttt{local var } x : t = erasure(G, e, t)}$$

Fig. 9. Defintion of type erasure. The *erasure* function ($G \times Expr \times Type \longrightarrow Expr$) takes an API graph, an expression e, and a target type t, and yields another expression e' with erased type information.

still maintaining the type correctness of the input expression. Type erasure helps our approach stress-test the implementation of compiler type inference procedures [Chaliasos et al. 2022].

Type erasure aims to construct polymorphic invocations with no explicit type arguments. Our method relies on local type inference algorithms [Pierce and Turner 2000], which are commonly supported by mainstream programming languages. In the context of local type inference, the type arguments of polymorphic invocations are deduced in a manner that satisfies two conditions: (1) the method arguments must be compatible with the formal parameter types of the method being invoked, and (2) the type of the entire polymorphic invocation must be compatible with a target type derived from the surrounding context.

Specifically, our type erasure process leverages the following key insight: a type argument of a polymorphic invocation can be safely erased if it can be inferred by either the arguments of the invocation, or the target type of the invocation. At the same time, the explicit type argument should be the same as its inferred counterpart. More formally:

Definition 3.14. Consider an API graph G, a polymorphic method m with signature $sig(G, m) = \langle r, \overline{p}, t\rangle$ and its invocation $e = r.m\langle\overline{t}\rangle(\overline{e'})$, a target type k for e, and a substitution σ that maps every type variable of method m to their type arguments in e. We say that the type argument of a type variable of method m can be *safely removed* from expression e, when:

- $\sigma(\alpha) = \sigma'(\alpha)$, where $\sigma' = unify(k, t)$
- or $\sigma(\alpha) = \sigma'(\alpha)$, where $\sigma' = \bigcup_{i=1}^{n} unify(p_i', p_i)$ and $G \vdash \overline{e'} : \overline{p'}$

Based on the preceding property, we define the function *erasure* ($G \times Expr \times Type \longrightarrow Expr$), which takes an API graph, an input expression e, and a target type t. It outputs another expression with type information removed. The full definition of *erasure* is shown in Figure 9. The function *erasure* is recursively applied to any sub-expression included in e. When handling a method call e, the type arguments are removed by *erasure*, but only when *every* type argument of the call can be safely erased according to Definition 3.14 ([method call]).

Example: Consider the following code snippet:

```
1  <T> void m1(T x) {}                          <T> void m1(T x) {}
2  <X, Y> Y m2(X p1)                            <X, Y> Y m2(X p1)
3  m1<Object>("str");                           m1<Object>("str"); // Not erased
4  m1<String>("str");                           m1("str"); // Erased
5  m2<String, String>("f");                     m2<String, String>("f"); // Not erased
6  String x = m2<String, String>("str");        String x = m2("str"); // Erased
```

After applying the (optional) function *erasure*, we get the method calls shown on the right. *erasure*

removes the type arguments from the second and fourth polymorphic invocation (lines 4, 6), while the first and the third call remain the same. Specifically, if we chose to erase the type argument of the first call (line 3), the inferred type of `T` would become `String`, which is not equivalent with the explicit type argument `Object`. Similarly, we do not remove the type arguments from the method call on line 5, because the type argument of type variable `Y` cannot be safely erased. This is because there is no target type for the entire call that helps with the inference of `Y`.

Relation to HEPHAESTUS' type erasure approach: Erasing types has proven useful in finding bugs in type inference implementations, as demonstrated by the HEPHAESTUS tool [Chaliasos et al. 2022]. HEPHAESTUS comes with a mutation that erases types from an *existing* program. In contrast, our approach differs by incorporating type erasure directly into the synthesis process. During the generation of a method call, we check whether the method call's type arguments can be omitted based on the inference rules outlined in Figure 9. Notably, integrating type erasure into the synthesis process makes our approach efficient, as we further show in Section 4.5 of our evaluation. One distinct difference is the overhead in HEPHAESTUS, which is attributed to an intra-procedural analysis that preserves the type correctness of the input program during the type erasure mutation.

3.6 Implementation and Discussion

Having presented the theory behind the main components of the approach, this section focuses on noteworthy technical details. We have implemented our techniques on top of HEPHAESTUS, the modern framework for testing compilers' type checkers [Chaliasos et al. 2022]. Our implementation, which we call THALIA, extends HEPHAESTUS using roughly 5k additional lines of Python code.

The input of THALIA is a set of JSON files that describe a given API. We have developed an auxiliary script that automatically produces such JSON files by parsing a library's API documentation web pages (e.g., generated by `javadoc`) via the `beatifulsoup4` Python package. THALIA then examines the input JSON files and builds the corresponding API graph.

Producing typing sequences: API enumeration is exponential in terms of the number of types found in the signature of an API component. To make API enumeration practical, THALIA employs randomization. When dealing with a polymorphic API component d, THALIA first generates a *valid* type substitution σ at random. Based on the randomly generated substitution σ, THALIA then computes the set S_d containing the typing sequences of d as described in Section 3.3. THALIA generates one test case for every $s_d \in S_d$ by applying function $\hat{\gamma}(G, d, s_d, \sigma)$ as detailed in Section 3.4.

Enumerating API paths: To compute all simple paths that reach a specific node, THALIA employs Yen's algorithm [Yen 1971], which computes the k-shortest loopless paths in a graph. During the concretization of an API typing sequence, THALIA invokes a variant of Algorithm 1 presented in Section 3.4. THALIA iterates over the paths given by Yen's algorithm in a *random* order, and rather than returning all feasible paths, THALIA returns the first random path that forms a type inhabitant.

API graph size: Intuitively, the input API affects the size of the underlying API graph, and thus the scalability of our approach. However, as we show in our evaluation, our tool can easily handle real-world APIs with more than 25k edges, and synthesize programs in milliseconds.

Incomplete APIs: Although our realization function $\hat{\gamma}$ (Section 3.4) always returns a non-empty set of expressions, we may be unable to exercise a specific API component due to the presence of recursive bounds and the use of an API with missing type information. To illustrate this, consider:

```
class A<T extends A<T>> { int getSize(); }
```

We are unable to produce a typing sequence that invokes `getSize` as we cannot construct a proper receiver type derived from the type constructor `A`. This is because, there is no valid instantiation of type variable `T` that is compatible with the bound `A<T>`. This is unavoidable because the preceding

API does not contain subtypes of `A` (e.g., a subclass `class B extends A<B>`). To tackle this, our enumeration skips all API entities for which we cannot produce well-typed typing sequences.

Generalizability: THALIA currently produces programs written in three popular languages: Scala, Kotlin, and Groovy. Adding a new language requires (1) the collection of its APIs, (2) the implementation of a parser that transforms the string representation of a type (as it appears in the input JSON) into its in-memory counterpart supported by API-IR, and (3) a translator to convert API-IR programs into source files written in the target language.

Although our approach enables increased feature coverage (see Section 4.3), we might encounter an API that exhibits type system-related features that are not currently understood by API-IR. By default, we skip exercising API entities that involve such unsupported features. As a result, our current implementation might not be as effective for languages, such as Rust, OCaml, or TypeScript because the type system of API-IR does not currently support many of their core type system features, including currying, type aliases, structural types, associated types or more expressive bound constraints. However, the fundamental concepts of our approach (e.g., API enumeration) are still applicable to any modern language, because APIs are ubiquitous. For example, one has the option to enhance our API-IR language with new type-related features. We have already done so to accommodate Scala's higher-kinded types, and Kotlin's nullable types.

4 EVALUATION

Our evaluation answers the following research questions:

RQ1 Is THALIA effective in finding new compiler typing bugs? (Section 4.2)

Short answer: During a five-month period of developing THALIA and testing industrial-strength compilers, a total of 84 bugs were detected by THALIA. Out of these, 77 bugs have been confirmed or fixed.

RQ2 What are the characteristics of the test cases generated by THALIA? (Section 4.3)

Short answer: THALIA's test cases are concise, averaging only 11–13 lines of code across various languages. To reveal bugs, these compact test cases effectively combine key features, such as parametric polymorphism, overloading, and higher-order functions. Rather than solely relying on a generative process, these features originate from *existing* API definitions.

RQ3 What is the impact of libraries and synthesis modes on the effectiveness of THALIA? (Section 4.4)

Short answer: Using diverse libraries as input seeds increases code coverage across all tested compilers. Different synthesis settings further boost (1) the bug-finding capability by uncovering 36 additional bugs, and (2) the line coverage by up to 5%–11.9% across all compilers. THALIA scales well even with libraries containing large APIs, with an average synthesis time (per program) of 161 to 256 milliseconds, depending on the target language.

RQ4 How does THALIA compare to the state-of-the-art, namely HEPHAESTUS? (Section 4.5)

Short answer: THALIA and HEPHAESTUS complement each other effectively. THALIA has detected at least 42 bugs missed by prior work. HEPHAESTUS and THALIA achieve similar code coverage. However, combining the results of both tools significantly increases code coverage (up to 4.4%), demonstrating THALIA's ability to test previously unexplored components.

4.1 Experimental Setup

Hardware and compiler versions: We performed all experiments on commodity servers (32 cores and 64 GB of RAM per machine) running Ubuntu 22.04 (x86_64). Our testing efforts focused on the compilers of Groovy, Kotlin, and Scala. Although THALIA can produce Java programs, we did

Table 1. Characteristics of ten selected libraries. Each table entry indicates the number of nodes and edges of the API graph, the number of methods (including the polymorphic ones), the number of fields, the number of constructors, the number of types (including type constructors), the average size of the inheritance chain, and the average size of API signatures.

Library	Nodes	Edges	Poly. M/Methods	Fields	Constr	Type Con/Types	Inhr size	Sig size
com.fasterxml.jackson.core:jackson-core	6,775	10,158	16/1,442	54	82	4/98	3.80	2.47
com.google.guava:guava	10,235	14,469	975/4,363	188	0	214/412	3.31	2.68
org.apache.commons:commons-lang3	8,559	11,514	272/2,643	307	193	93/211	2.80	2.60
org.apache.logging.log4j:log4j-api	7,207	10,805	49/1,771	66	122	12/134	3.77	4.05
org-assertj-assertj-core	11,321	16,169	633/5,250	179	0	151/656	4.41	2.67
org.clojure:clojure	9,032	12,532	1/2,676	466	169	2/624	3.85	3.23
org.mockito:mockito-core	5,783	8,048	98/510	9	0	27/136	3.39	2.28
groovy-stdlib	14,817	22,303	461/10,982	1,108	1,159	164/1,276	4.21	2.46
kotlin-stdlib	6,020	8,986	205/3,808	1,108	438	70/395	5.47	2.37
scala-stdlib	3,933	5,928	390/2,439	688	246	93/331	4.23	2.49
other	6,900	9,974	43/1,330	150	97	16/204	3.32	2.37
Avg	**7,052**	**10,192**	**71/1,563**	**177**	**112**	**23/227**	**3.38**	**2.4**

not consider Java in our evaluation. The reason is that the issue tracker of OpenJDK is not open to the public, and therefore we were unable to directly interact with its compiler's developers. For each compiler, we tested (1) its latest development version by regularly building the compiler on its latest commit, or (2) its most recent stable version.

API collection: We collected APIs from two sources. Initially, we examined the APIs found in the standard library of the corresponding language, such as collections API, I/O API. Beyond standard libraries, we also considered APIs from third-party libraries found in the Maven central repository. Specifically, we examined the Maven repository and Scala index [Scaladex 2023] to gather the group ID, artifact ID, and the latest versions of the most popular Kotlin, Java, and Scala libraries. Rather than selecting libraries solely based on their popularity, we could establish a feature coverage criterion. This criterion would (1) exclude libraries that exhibit the same features with previously explored libraries, or (2) prioritize libraries with new and complex typing features. This is not a straightforward task, so we leave it as future work.

For each library, we searched in the Maven central repository to fetch: (1) the API documentation of the library, (2) the JAR files of the library and its dependencies. The documentation was converted into JSON files given as input to THALIA. Then, we built a classpath that pointed to the retrieved JAR files so that the compiler under test could locate the external symbols defined in external libraries. In total, we selected 95 libraries whose characteristics are shown in Table 1. The entry "other" shows the average metrics of the libraries not shown in the table.

We ran THALIA on each selected API under four different synthesis modes. In particular, on each API, we used THALIA to synthesize (1) well-typed programs with and without type erasure, and (2) wrongly-typed programs with and without type erasure.

Configuration: THALIA comes with a set of constants that affect the number of the synthesized programs (Section 3.6). For enumerating ill-typed typing sequences, we configured THALIA so that every type set contains at most five incompatible types selected randomly. Regarding type inhabitant selection, we configured Yen's algorithm to return the shortest path between two nodes. This is the equivalent to calling Dijkstra's algorithm for the shortest path calculation. Finally, the maximum depth of the generated programs is two. Based on our exploratory experiments, these configurations provide a good balance between bug-finding capability and performance.

4.2 RQ1: Bug-Finding Results

Table 2a summarizes the bug-finding results of THALIA. Overall, within five months of concurrent development and testing, we reported 84 bugs to developers, 77 of which are either confirmed or

Table 2. (a) Status of the reported bugs in `groovyc`, `kotlinc`, and Dotty, (b) number of bugs with unexpected compile-time error (UCTE), unexpected runtime behavior (URB), crash, or compilation performance (CPI) symptom, (c) bugs revealed by well-typed programs, well-typed programs with erased types (TE), or ill-typed ones.

Status	groovyc	kotlinc	Dotty	Total
Confirmed	38	9	8	55
Fixed	20	1	1	22
Duplicate	2	0	1	3
Won't fix	2	1	1	4
Total	**62**	**11**	**11**	**84**

(a)

Symptom	groovyc	kotlinc	Dotty	Total
UCTE	47	7	7	61
URB	6	0	0	6
Crash	8	3	4	15
CPI	1	1	0	2

(b)

Program type	groovyc	kotlinc	Dotty	Total
Well-typed	32	8	7	47
Well-typed (TE)	21	3	3	27
Ill-typed	9	0	1	10

(c)

fixed. To prevent duplicate bug reports, we conducted a comprehensive search in the issue tracker prior to submission, to identify potential related bugs. After reporting a small number of bugs to each development team, we waited a couple of weeks for the developers to fix or triage them. This helped us avoid overwhelming developers with an extreme number of unfixed/untriaged bugs. In the end, four of the reported bugs were marked as "won't fix". One "won't fix" issue is a compiler crash in Dotty. Scala developers think that it is not worth fixing it, although the issue is a regression introduced in Scala 3. Another one is a known limitation issue of `groovyc`.

Symptoms of compiler typing bugs: The symptoms of the discovered bugs are shown in Table 2b. The majority (61/84) lies in unexpected compile-time errors (UTCE): a compiler *unintentionally* rejects a well-typed program. UTCE is followed by compiler crash (15) and unexpected runtime behavior (6). An unexpected runtime behavior (URB)[3] is the symptom of a compiler bug that becomes evident at runtime. For example, soundness bugs typically lead to URB symptoms, e.g., runtime errors that should have been caught at compile-time. Two discovered bugs trigger severe compilation performance issues (CPI) in the type checkers of `groovyc` and `kotlinc`.

Bug-finding under different synthesis modes: Table 2c shows the bug-finding results of each synthesis mode. Most of the bugs (47/84) are triggered in the base mode, where THALIA produces well-typed programs that contain full type information (i.e., type erasure is disabled). This suggests that THALIA is effective in detecting compiler typing bugs, even when generating simple client code (at least at first blush). Using the type erasure process, THALIA has identified 27 additional bugs. All of these bugs are related to issues found in the type inference procedures of the compilers. Finally, THALIA has discovered 10 bugs that originate from ill-typed code. The low number of bugs triggered by ill-typed code in relation to the number of bugs that arise from well-typed programs is consistent with the findings of Chaliasos et al. [2021, 2022].

Intricacy of typing bugs: As highlighted by the study of Chaliasos et al. [2021], fixing compiler typing bugs often becomes particularly challenging. This insight is indeed confirmed by most of the compiler development teams we have interacted with: many reported bugs require much expertise in the areas of overload resolution and type inference.

Our interaction with developers has revealed another noteworthy observation: typing bugs take a fair amount of effort to understand their root causes, particularly identifying which specific components in type checking procedure break down. Therefore, additional tooling for root cause analysis would be beneficial for compiler developers. Furthermore, THALIA has helped us reveal interesting bugs that stem from issues in the language design. These bugs are exceptionally challenging to fix, as their fix requires pervasive changes in the compiler code base. These changes are susceptible to breaking other language features.

[3]We follow the same terminology used in the study of Chaliasos et al. [2021]. THALIA detects URB bugs at compile-time, and not at runtime, based on our test oracle for soundness bugs, that is, erroneous acceptance of ill-typed programs.

Table 3. The language features that appear in the minimized test cases of our reported bugs. A check mark indicates whether the corresponding feature is supported by the state-of-the-art tool HEPHAESTUS.

Feature type	Feature	Frequency	Supported by HEPHAESTUS
Declaration	Polymorphic class	58/84	✓
	Polymorphic function	45/84	✓
	Single Abstract Method (SAM)	30/84	✓
	Overloading	22/84	✗
	Inheritance/Implementation of multiple interfaces	8/84	✗
	Variable argument	5/84	✓
	Access modifier	2/84	✗
	Inner class	3/84	✗
	Bridge method	1/84	✗
	Default method	1/84	✗
	Static method	2/84	✗
	Operator	1/84	✗
Type inference	Type argument inference	23/84	✓
	Variable type inference	2/84	✓
Type	Polymorphic type	58/84	✓
	Wildcard type	19/84	✓
	Bounded type parameter	15/84	✓
	Array type	7/84	✓
	Subtyping	7/84	✓
	Recursive upper bound	3/84	✗
	Primitive type	3/84	✓
	Nullable type	2/84	✗
Expression	Function reference	21/84	✓
	Lambda	10/84	✓
	Conditionals	4/84	✓

4.3 RQ2: Test Case Characteristics

We manually identified what language features appear in every minimized test case that accompanies our bug reports. Table 3 shows the frequency of the identified features. Features related to parametric polymorphism (e.g., polymorphic function, wildcard type, etc.) are the top features in terms of bug-finding capability. Roughly 85% (70/84) of the bug-triggering test cases contain at least one feature related to parametric polymorphism. This finding is in line with the work of Chaliasos et al. [2021, 2022]. Our finding is further supported by compiler developers who have acknowledged that working with generics is a highly demanding task that requires a high level of expertise. Parametric polymorphism is followed by functional programming features (e.g., lambdas, function references), type inference features, and overloading. Figure 10 shows the overlap of parametric polymorphism, functional programming, and overloading. Overall, these features are found in all but eight test cases. Finally, Table 3 confirms that THALIA increases feature coverage, especially for definition-related features. This means that THALIA can exercise the implementation of these features in an agnostic way, as long as they they are part of the input APIs.

Size of test cases: : The test cases given by THALIA are consistently small across the examined languages. On average, a Groovy test case measures 1.6kB and contains 13 lines of code (LoC), while a Scala test case measures 1.6kB and consists of 11 LoC. Similarly, Kotlin test cases have an average size of 1.4kB and an average of 11 LoC. The compact size of these test cases facilitates efficient testing with higher throughput and simplifies the generation of minimal test-cases.

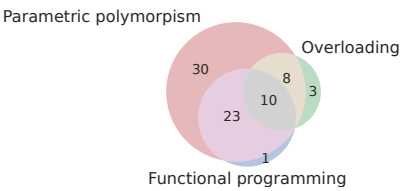


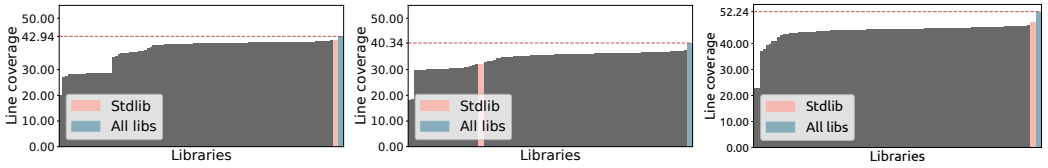
Fig. 10. The Venn diagram of features related to parametric polymorphism, functional programming, and overloading.

Overall, the above results indicate that although the test cases produced by THALIA are small in size, they are able to discover bugs triggered by a combination of complicated languages features.

4.4 RQ3: Impact of Library Selection and Synthesis Modes

To evaluate the impact of library selection and investigate the effectiveness of different synthesis modes, we conducted a comprehensive evaluation using 95 top libraries found in the Maven repository, including the standard library of each language.

Figure 12 shows how many well-typed and ill-typed test cases (typing sequences) have been produced by our API enumeration technique. In total, the examined 95 libraries yielded roughly 1.6



(a) groovyc (b) Dotty (c) kotlinc

Fig. 11. Compiler line coverage when using top 95 Maven libraries. Notably, when combining the results of all libraries (highlighted in blue), there is an increase in line coverage, ranging between 1.6%–4% when compared to the best individual library of each language.

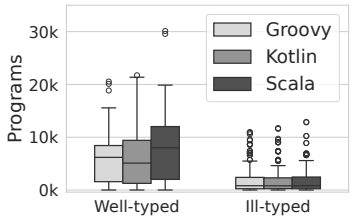


Fig. 12. Number of test cases produced by THALIA categorized by their type.

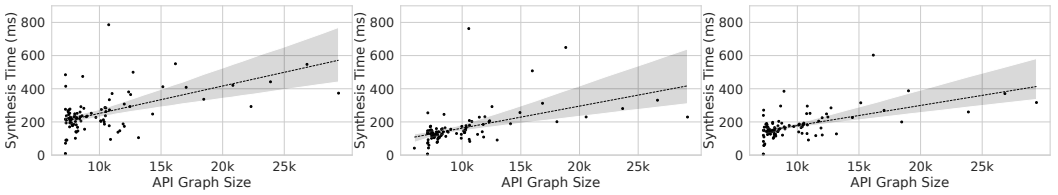
Table 4. Statistics regarding the increase in line coverage achieved by each synthesis mode. The baseline is when THALIA synthesizes well-typed programs with complete type information. Numbers outside parentheses indicate absolute change, while percentages in parentheses show relative change. The type erasure mode (TE) exhibits the highest line coverage increase, followed by producing ill-typed programs with missing type information (Both).

Compiler	Mode	Statistics				
		Min	Max	Median	Mean	Std
`groovyc`	TE	+0 (0%)	+4,306 (11.9%)	+166 (0.5%)	+265 (0.7%)	1.6%
	Ill-typed	+0 (0%)	+2,952 (8.1%)	+45 (0.1%)	+118 (0.3%)	1.1%
	Both	+0 (0%)	+3,087 (8.5%)	+163 (0.5%)	+195 (0.5%)	0.9%
Dotty	TE	+0 (0%)	+5,548 (7.1%)	+934 (1.2%)	+1087 (1.4%)	1.1%
	Ill-typed	+7 (0.01%)	+3,595 (4.6%)	+475 (0.6%)	+615 (0.8%)	0.7%
	Both	+117 (0.2%)	+4,192 (5.3%)	+852 (1.1%)	+975 (1.24%)	0.8%
`kotlinc`	TE	+3 (0.01%)	+2,144 (3.7%)	+523 (0.9%)	+566 (1%)	0.5%
	Ill-typed	+1 (0%)	+3,030 (5.2%)	+175 (0.3%)	+291 (0.5%)	0.7%
	Both	+45 (0.1%)	+2,175 (3.7%)	+347 (0.6%)	+420 (0.7%)	0.6%

million programs for each language. Based on these programs, we performed an analysis on the time spent synthesizing them, as well as the code coverage achieved by each library and mode.

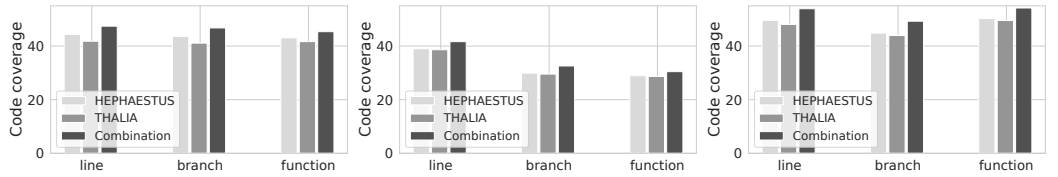
Code coverage analysis: We used the JaCoCo library [EclEmma 2023] to measure the code coverage in each compiler. Figure 11 shows the line coverage achieved by programs derived from each library in all synthesis modes. We observe that the input library and its corresponding API significantly affects the resulting line coverage. Libraries with limited methods and polymorphic components tend to have lower line coverage, indicating that the extent of code coverage depends on the richness of the input API. When combining the programs that arise from all the input libraries (see blue bars), there is a noteworthy code coverage increase in all the examined compilers. For example, in `groovyc`, the combination of libraries surpasses the line coverage achieved by the best individual library by 1.6%. This percentage translates to hundreds of additional covered lines. Similarly, in Dotty and `kotlinc`, the line coverage improvement ranges between 3% and 4%. Similar patterns have been observed when considering branch and function coverage. Based on the above, using APIs with distinct characteristics avoids saturation and exercises unexplored code.

Table 4 illustrates the impact of each mode on line coverage in comparison to the base mode, where THALIA produces well-typed programs with type erasure disabled. THALIA's different modes contribute to a line coverage increase up to 5%–11.9% across all compilers. While the goal is not to explore the entire code base, but rather to exercise specific compiler parts, such as type inference, employing multiple modes is crucial in maximizing the effectiveness of THALIA. This is further supported by the number of bugs (36/84) that were discovered when using a mode other than the base mode (see Table 2c). Many of them concern important type inference and soundness issues.



(a) groovyc (b) Dotty (c) kotlinc

Fig. 13. The synthesis time per program (in milliseconds) as a function of the size of the API graph.



(a) groovyc (b) Dotty (c) kotlinc

Fig. 14. Comparison between code coverage of HEPHAESTUS and THALIA.

Synthesis time: Figure 13 shows how the size of the underlying API graph influences the time spent on the synthesis of a *single* test case. The correlation between the size of the API graph and the synthesis time is almost linear for all the compilers. THALIA takes less than 400 milliseconds to synthesize one program coming from large libraries with more than 25k edges in the corresponding API graph. The average synthesis time is 256, 161, and 178 milliseconds for Groovy, Scala, and Kotlin programs respectively. This demonstrates the practicality of THALIA in synthesizing client programs in milliseconds, even if they come from large libraries and complex APIs.

There are some exceptional cases where the synthesis time is relatively high (see data points above the regression lines). This is explained by the high number of polymorphic definitions with bounded type parameters included in the corresponding APIs. For example, the org.assertj:assertj-core library has a larger number of type constructors and polymorphic methods that define type parameters with recursive upper bounds (e.g., `T extends A<T>`) In such cases, THALIA spends significant time seeking appropriate type arguments to instantiate these polymorphic definitions.

4.5 RQ4: Comparison of Thalia vs. Hephaestus

We compared our work with HEPHAESTUS [Chaliasos et al. 2022], which is the state-of-the-art framework for validating static typing implementations. Originally, HEPHAESTUS had support for three languages, two of which were also included in our evaluation (Kotlin, Groovy). To make HEPHAESTUS also generate Scala programs, we followed the guidelines in the work Chaliasos et al. [2022] and implemented a translator that allows the conversion of programs written in the HEPHAESTUS's IR into concrete Scala programs.

Bug-finding capability: We manually analyzed the bug-revealing test cases of THALIA to identify how many bugs are missed by HEPHAESTUS. Looking at Table 3, it is clear that the bug-revealing test cases exercise a plethora of characteristics, such as overloading and the use of inner classes, which fall outside the capabilities of HEPHAESTUS. Specifically, out of the 84 bugs identified by THALIA, HEPHAESTUS fails to detect 42 of them, accounting for a 51% miss rate. Interestingly, these 42 bugs were found *effortlessly* by THALIA, as it leverages existing compiled code instead of explicitly supporting the generation of intricate definitions.

For the remaining detected bugs, we also examined whether these bugs existed in the compiler versions used in the testing campaign by Chaliasos et al. [2022]. Remarkably, out of these bugs, 9

were found to exist in those versions, and HEPHAESTUS failed to detect them despite the thorough efforts. This is explained by the expressiveness of real-world APIs that made THALIA produce code featuring complex combinations of typing features. For example, GROOVY-11020 is triggered by passing a function reference to a polymorphic method that defines a type parameter bounded with a SAM type. Although HEPHAESTUS supports polymorphic methods and SAM types, there is an extremely low probability of generating code with such a programming idiom.

Nevertheless, there are certain bugs triggered by HEPHAESTUS that cannot be detected by THALIA. Most of these bugs have to do with implementation flaws that are related to the semantic validation of definitions. Since THALIA employs pre-compiled API definitions, it fails to detect these types of compiler errors. For example, KT-49583 is a bug, previously-reported by the team of HEPHAESTUS. The accompanying test case contains an anonymous function that encompasses another nested function declaration. This issue is out of THALIA's reach, because rather than producing new declarations (e.g., a function in a lambda), THALIA re-uses existing code.

Code coverage analysis: We conducted a 12-hour run for both tools in every language, using all available modes and generating as many test cases as possible within that time frame. We ran HEPHAESTUS using the same settings as described in the work of Chaliasos et al. [2022]. We assessed the code coverage in each compiler using the programs generated by each tool.

Figure 14 illustrates the code coverage attained by HEPHAESTUS and THALIA. HEPHAESTUS slightly outperforms THALIA on all the examined compilers. This outcome aligns with our expectations, as HEPHAESTUS produces larger programs and heavily emphasizes the utilization of declarations. However, THALIA offers the advantage of easily testing arbitrary features (Section 4.3). Consequently, when combining the two approaches, we observe a percentage increase in HEPHAESTUS' line coverage of 7% for `groovyc` and Dotty, and 9% for `kotlinc`. This translates to 1,118–2,567 additionally covered lines of code. For branch coverage, the combination of THALIA and HEPHAESTUS leads to a 7% increase for `groovyc`, 9% for Dotty, and 10% for `kotlinc`. This translates to 5,395–22,334 additionally covered branches. Finally, when considering function coverage, THALIA contributes to calling 5% more functions in `groovyc` and Dotty, and 8% more functions in `kotlinc`. This translates to 173–654 functions that were previously unexplored by HEPHAESTUS. We looked further into some packages and classes to get an insight about that newly-explored code. Some noteworthy examples include: (1) the `org.jetbrains.kotlin.resolve.*` package, which implements the name resolution procedure in `kotlinc`, (2) a couple of classes in Dotty (e.g., `dotty.tools.dotc.core.TypeComparer`) that handle polymorphic types and type comparisons. As expected, THALIA missed a lot of code associated with declarations and types of expressions (e.g., binary operators) supported only by HEPHAESTUS. Overall, our findings indicate that THALIA helps explore new code in the compilers' code base, addressing the current limitations of HEPHAESTUS.

Table 5. Average time of compiling and generating HEPHAESTUS and THALIA programs. Generation time is per program, whereas compilation time is per batch of 45 programs.

Compiler	Compilation time (sec)		Generation time (sec)	
	THALIA	HEPHAESTUS	THALIA	HEPHAESTUS
`groovyc`	3.1	6.6	0.3	0.7
Dotty	5.4	7.9	0.3	1
`kotlinc`	6.5	26.5	0.2	1.3

Code size: Within the 12-hour time period, the average size of THALIA's programs written in Groovy, Scala, and Kotlin is 15, 12, and 11 LoC, respectively. HEPHAESTUS generated programs that are one order of magnitude larger, with an average size of 304, 262, and 257 LoC for Groovy, Scala, and Kotlin respectively. The smaller size of THALIA's programs simplifies the generation of minimal test-cases, which are invaluable for compiler writers [Regehr et al. 2012].

```
1  def test() {
2    val x: String = "strVal"
3    Predef.identify[Function1[? >: Int, String]](x.
           substring)
4  }
5  // Definition of the API
6  class String {
7    def substring(begin: Int): String
8    def substring(begin: Int, end: Int): String
9  }
10 object Predef {
11   def identity[A](x: A): A
12 }
```

(a) DOTTY-17310: Fail to resolve the reference to method `String.substring`.

```
1  fun test() {
2     val x: Iterable<HashSet<Number>> = TODO()
3     val y: HashSet<Number> = TODO()
4     // Type mismatch: inferred type is Number
5     val res: List<HashSet<Number>> = x.minus(y)
6  }
7  // Definition of the API
8  package kotlin.collections;
9
10 operator fun <T> Iterable<T>.minus(element: T): List<
       T>
11 operator fun <T> Iterable<T>.minus(elements: Iterable
       <T>): List<T>
```

(b) KT-57596: Resolving wrong method in the presence of operator overloading.

```
1  import com.google.common.collect.HashBasedTable;
2   void test() {
3     Number x = HashBasedTable.<Number, Number, Number>
          create().get(null, null);
4    }
5  }
6  // Definition of the API
7  package com.google.common.collect;
8  interface Table<R,C,V> {
9    default V get(Object x, Object y);
10 }
11 class HashBasedTable<R,C,V> implements Table<R,C,V> {
12   static <R,C,V> HashBasedTable<R,C,V> create();
13 }
```

(c) GROOVY-11012: A bug in the treatment of default methods leads to an UTCE.

```
1  import java.util.Map
2  import org.apache.commons.lang3.Validate.notEmpty;
3  def test(): Unit = {
4    val x: Map[String, String] = ???
5    val res: Map[String, String] = notEmpty[Map[String,
         String]](x, "foo");
6  }
7  // Definition of the API
8  package org.apache.commons.lang3
9  object Validate {
10   def notEmpty[T](x: Array[T]): T
11   def notEmpty[T <: CharSequence](chars: T): T
12   def notEmpty[T <: Map[?, ?]](map: T): T
13 }
```

(d) DOTTY-17412: Dotty is unable to call an overloaded method with bounded type parameters.

Fig. 15. Sample test programs that trigger typing bugs.

Generation and compilation time: Table 5 presents the average time spent by each tool on generating and compiling programs. HEPHAESTUS features higher times compared to THALIA, with varying performance across different languages. For example, `groovyc` spends an average time of 6.6 seconds for compiling 45 HEPHAESTUS' programs, while it needs only 3.1 seconds, on average, to compile 45 programs given by THALIA. At the same time, THALIA synthesizes a Groovy test case in 0.3 seconds, while HEPHAESTUS requires 0.7 seconds to generate a single Groovy program. These findings highlight that THALIA can achieve greater throughput than the state-of-the-art work.

In short, our comparative analysis highlights four distinct differences between THALIA and HEPHAESTUS. (1) THALIA's programs are considerably smaller than those of HEPHAESTUS (code size). (2) THALIA's programs involve faster synthesis and compilation times (generation and compilation time). (3) Despite the reduced size of its test cases, THALIA still explores compiler regions that HEPHAESTUS does not reach (code coverage analysis). (4) Relying on existing compiled libraries makes THALIA effortlessly uncover 51 bugs missed by HEPHAESTUS (bug-finding capability).

4.6 Bug-Revealing Test Samples

Figure 15a: This is a regression introduced in Scala 3. The code creates a reference to an overloaded method of class `String` named `substring` and passes it as an argument to the higher-order function `identity` which expects something of type `Function1[? >: Int, String]` (line 3). Although the intention is to make a reference to the overloaded method defined on line 7, Dotty rejects the program with an error of the form: *"None of the overloaded alternatives of method substring*

match expected type". Interestingly, when the expected type is `Function1[Int, String]`, Dotty resolves the correct method as expected.

Figure 15b: The code demonstrates an issue in the design of Kotlin. As with C++ and other languages, Kotlin supports operator overloading. The standard library of Kotlin defines two overloaded methods for the operator - (minus). When calling method `minus`, `kotlinc` decides to resolve the method defined on line 11. However, this decision leads to a type mismatch, as the inferred type of the method call becomes `List<Number>`, while the expected type is `List<HashSet<Number>>`. The compiler should have resolved the overloaded variant on line 10, where the type variable `T` is instantiated by the type `HashSet<Number>`. This issue prevents the removal of polymorphic items from mutable collections (e.g., lists, sets) via the corresponding operator, i.e., `x - y`.

Figure 15c: In this example, the compiler erroneously rejects a well-typed program. The root cause of this failure lies in the way Groovy treats interfaces with default methods (see lines 8–10). In Groovy, an interface with a default method is implemented as a trait, a structural construct that allows composition of behaviors and implementation of interfaces. `groovyc` fails to propagate the given type arguments when calling `create` (line 3) to the trait that implements the interface `Table` (lines 8, 12). Consequently, while checking the method call on line 3, `groovyc` infers the return type of the method `get` as `Object`. This leads to an UCTE, as the expected type is `Number`.

Figure 15d: This program calls an overloaded method named `notEmpty` that comes from the `org.apache.commons:commons-lang3` library (line 5). The library contains a few polymorphic variants of method `notEmpty`; each of them defines a type parameter with a unique upper bound (lines 10–12). The intention is to call the third overloaded alternative (line 12), as the explicit type argument of the method call on line 5 comes in line with the bound of the underlying type parameter (i.e., `Map[?,?]`). Nevertheless, an issue in Dotty's overload resolution procedure makes the compiler reject the program by reporting an ambiguous method call. This issue stems from the design choice of Dotty developers to check bounds only after type checking (to avoid cycles due to recursive upper bounds). Therefore, bounds do not influence overload resolution. In this context, it is impossible to properly call the intended method from the given library. Surprisingly, when the definition on line 10 is removed, the compiler behaves as expected.

5 RELATED WORK

Generative compiler testing: *Generative compiler testing* is an umbrella term for methods that construct test programs completely from scratch. Csmith [Yang et al. 2011] is the most influential program generator which produces well-typed C programs while avoiding undefined behaviour. Many subsequent generators have leveraged the power of Csmith to test other compilers (e.g., OpenCL) [Lidbury et al. 2015] and compiler components [Le et al. 2015b], or enhance the diversity of the generated programs [Even-Mendoza et al. 2020, 2022]. A more recent program generator for C/C++ programs, YARPGen [Livinskii et al. 2020], produces programs with complex arithmetic expressions that are more likely to trigger optimization bugs. Its re-implementation [Livinskii et al. 2023] focuses on validating loop optimizers.

A key challenge associated with generative compiler testing is the construction of code generators for syntactically- and semantically-valid programs that help test beyond the front-end of a compiler. This typically involves a large amount of engineering effort, while the corresponding implementation is usually crafted for a single target language. Exploiting well-formed definitions taken from existing software libraries allows us to synthesize programs from scratch, but at the same time, easily port our program generator to multiple languages.

Mutation-based compiler testing: Mutation-based compiler testing produces new test programs by modifying existing ones. The most effective compiler testing method that lies in this category is

equivalence modulo input (EMI) [Le et al. 2014, 2015a; Sun et al. 2016]. EMI profiles the execution of a seed program and applies a set of transformations so that the resulting programs have the same output as the original one. Applying semantics-preserving transformations to existing programs has been shown highly effective in the graphics driver domain [Donaldson et al. 2017, 2021].

A primary limitation of mutation-based testing is that its effectiveness is limited to the quality of the available seed programs. Our work tackles this limitation, as mainstream languages come with a rich library ecosystem. Library APIs expose a wide variety of advanced features that are more likely to trigger typing bugs [Chaliasos et al. 2021].

Program enumeration for compiler testing: A number of *enumeration techniques* have been developed for detecting issues in deep compiler phases, such as optimizations and code generation. *Skeletal program enumeration (SPE)* [Zhang et al. 2017] takes a program skeleton and enumerates all program variants that involve unique variable usage combinations. SPE aims at exhibiting diverse control- and data-dependence. *Type-centric enumeration (TCE)* [Stepanov et al. 2021] finds crashes in the Kotlin compiler by first generating a reference program of a certain structure and then replacing all program expressions with other compatible expressions of the same type. In contrast to our work, the outcome of TCE is not guaranteed to be type correct.

In this work, we propose a different enumeration technique: API enumeration focuses on exposing diverse typing patterns of API invocations to exercise interesting compiler behaviours with regards to subtyping and compile-time name resolution.

Finding compiler typing bugs: Most of the aforementioned techniques target optimizing compilers. The CLP-based (*Constraint Logic Programming*) program generator proposed by Dewey et al. [2015] and HEPHAESTUS [Chaliasos et al. 2022] represent the work most closely related to our approach. The CLP-based method works by encoding the semantics rules of the language under test into a set of logical constraints. Querying a CLP engine under the given constraints enables the generation of well-typed or ill-typed programs. Despite its application to the Rust compiler, CLP-based program generation comes with many caveats that limit its generality and practicality, e.g., poor performances, and even non-termination.

HEPHAESTUS employs *type graphs* with the primary goal of capturing (1) the dependencies between type variables (i.e., determining whether a type variable can be inferred by another), and (2) the inferred or declaration type of local variables. A type graph is constructed using an intra-procedural analysis on an existing program. HEPHAESTUS consults type graphs to maintain type correctness in its type erasure mutation (Section 3.5). In contrast, our API graph captures (1) all API definitions (methods, fields) found in an API, (2) what types are needed to perform an application or a field access, and (3) what is the type of each application or field access. An API graph is constructed by traversing a given API specification, rather than a program. The main purpose of API graphs is to identify type inhabitants.

Section 4.5 thoroughly compares our work with HEPHAESTUS. In short, THALIA's underlying process, which relies on API enumeration rather than randomized program generation, yields programs with distinct characteristics (smaller code size) and enables exercising compiler regions that have been previously unexplored by HEPHAESTUS.

Component-based program synthesis: Our work is also related to program synthesis techniques that generate small code fragments using components of existing libraries. The purpose of these techniques is not to test compilers, but rather to assist developers in programming tasks via library code reuse. In this context, a developer provides an incomplete expression [Gvero et al. 2013; Perelman et al. 2012] or a method signature [Feng et al. 2017; Guo et al. 2022, 2019]. A program synthesis tool then produces a ranked list of implementation sketches that better match the developer's intent.

The idea of labelling API graphs with type substitutions shares similarities with a graph data structure called *equality-constrained tree automata (ECTA)* [Koppel et al. 2022]. In ECTA, nodes are annotated with equality constraints. While our API graph constrains type variable instantiations, ECTA goes further by constraining arbitrary types, such as matching an argument type with its corresponding formal parameter type. ECTA is more expressive than API graphs, because it solves the more general program synthesis problem, where the relevancy of the proposed solutions matters. In contrast, the primary use of API graphs is much simpler: we aim to identify chains of method calls/field accesses of a certain type. This allows for (1) a compact representation, especially the treatment of polymorphic types, (2) an efficient enumeration done by *standard* graph reachability algorithms (e.g., Yen's algorithm), and (3) identification of variable-length chains of method calls/field accesses, in contrast to ECTA's fixed size approach.

API graph is inspired by PROSPECTOR [Mandelin et al. 2005]. PROSPECTOR introduces the *signature graph*, which treats every API component as a unary function that takes an input type (receiver type), and produces an output type (return type). Contrary to our API graph, PROSPECTOR does not handle parametric polymorphism which is a key language feature for revealing typing bugs [Chaliasos et al. 2021]. To handle parametric polymorphism, we enrich PROSPECTOR with Algorithm 1 and type substitution labels. A generalization of PROSPECTOR is SYPET [Feng et al. 2017], which models the structure of an API using a Petri-net. Guo et al. [2019] extend SYPET by introducing the *type-guided abstraction refinement* that allows program synthesis over polymorphic types using an SMT encoding. Finally, InSynth [Gvero et al. 2013] assigns weights to both API definitions and types, modeling program synthesis as an optimization problem. InSyth leverages these weights to guide its search for type inhabitants, selecting expressions that minimize a specific weight function.

The goal of these techniques is different from ours. They strive to synthesize the most optimal solution e.g., in terms of code size, number of API method invocations, type distance, or users' intent. Also many of these tools lack support for parametric polymorphism, or exhibit high running times. Such challenges can make these tools less suitable for our compiler testing scenarios.

SyRust [Takashima et al. 2021] is a semantic-aware program synthesis technique for testing Rust libraries. Using a manually-generated code template and inputs, it encodes the typing rules of Rust (w.r.t. ownership, variable lifetime) into a satisfiability problem and synthesizes test cases of increasing size. Contrary to our work, SyRust is semi-automatic and faces scalability issues, because it is capable of handling 15 API definitions (e.g., methods) per library. Also, SyRust's focus lies in identifying memory-safety bugs in library implementations, rather than producing test cases that showcase diverse typing patterns (e.g., type inference reasoning) to test compilers.

6 CONCLUSION

We have presented an API-driven program synthesis approach for testing the implementation of compilers' static typing procedures. Our method harnesses the ubiquity and complexity of APIs extracted from established software libraries, and synthesizes concise client programs that employ API entities (e.g., types, functions, fields) through different typing patterns. This helps us exercise a broad spectrum of type-related compiler functionalities, without the need to generate complex API definitions ourselves. Our evaluation on the compilers of Scala, Kotlin, and Groovy has shown the effectiveness of our approach. Indeed, our implementation has uncovered 84 bugs (77 bugs are either confirmed or fixed), 51 of which could not have been detected by prior work.

Our API-driven program synthesis approach opens up further opportunities for combining synthetic and real-world code to discover bugs in compiler implementations. For example, in addition to generating client programs, we can extend our approach to create intricate definitions (e.g., classes) that build upon existing ones found in an API (e.g., through inheritance constructs). We believe that this extension will enable the identification of new compiler bugs.

ACKNOWLEDGMENTS

We are grateful to Diomidis Spinellis, Michel Weber, Katerina Vlachaki, Alastair Donaldson, and the anonymous POPL reviewers for their detailed feedback on earlier versions of the paper.

DATA AVAILABILITY STATEMENT

THALIA is available as open-source software under the GNU General Public License v3.0 at https://github.com/hephaestus-compiler-project/thalia. The research artifact [Sotiropoulos et al. 2023a] is also available under the same license.

REFERENCES

Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. 2015. Exploiting the Saturation Effect in Automatic Random Testing of Android Applications. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems* (Florence, Italy) *(MOBILESoft '15)*. IEEE Press, 33–43.

Nada Amin and Ross Tate. 2016. Java and scala's type systems are unsound: the existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 838–848. https://doi.org/10.1145/2983990.2984004

Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (Oct. 2021), 30 pages. https://doi.org/10.1145/3485500

Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding Typing Compiler Bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 183–198. https://doi.org/10.1145/3519939.3523427

Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. 2017. A Type System for Privacy Properties. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 409–423. https://doi.org/10.1145/3133956.3133998

Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 482–493. https://doi.org/10.1109/ASE.2015.65

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133917

Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-Case Reduction and Deduplication Almost for Free with Transformation-Based Compiler Testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1017–1032. https://doi.org/10.1145/3453483.3454092

EclEmma. 2023. EclEmma Jacoco. https://www.eclemma.org/jacoco/. Online accessed; 04-07-2023.

Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2020. Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative about Undefined Behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1219–1223. https://doi.org/10.1145/3324884.3418933

Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2022. CsmithEdge: More Effective Compiler Testing by Handling Undefined Behaviour Less Conservatively. *Empirical Softw. Engg.* 27, 6 (nov 2022), 35 pages. https://doi.org/10.1007/s10664-022-10146-1

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-Based Synthesis for Complex APIs *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 599–612. https://doi.org/10.1145/3009837.3009851

Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. 2022. Type-Directed Program Synthesis for RESTful APIs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 122–136. https://doi.org/10.1145/3519939.3523450

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.* 4, POPL, Article 12 (dec 2019), 28 pages. https://doi.org/10.1145/3371080

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2491956.2462192

James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. *Proc. ACM Program. Lang.* 6, ICFP, Article 91 (aug 2022), 29 pages. https://doi.org/10.1145/3547622

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

Vu Le, Chengnian Sun, and Zhendong Su. 2015a. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

Vu Le, Chengnian Sun, and Zhendong Su. 2015b. Randomized Stress-Testing of Link-Time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/2771783.2771785

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428264

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (jun 2023), 22 pages. https://doi.org/10.1145/3591295

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 48–61. https://doi.org/10.1145/1065010.1065018

Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A Flexible Type System for Fearless Concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 458–473. https://doi.org/10.1145/3519939.3523443

Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-Directed Completion of Partial Expressions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2254064.2254098

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (jan 2000), 1–44. https://doi.org/10.1145/345099.345100

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

Scaladex. 2023. The Scala library index. https://index.scala-lang.org/. Online accessed; 04-07-2023.

Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2023a. *Artifact for "API-driven Program Synthesis for Testing Static Typing Implementations"*. https://doi.org/10.5281/zenodo.10077754

Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2023b. Extended Paper: API-driven Program Synthesis for Testing Static Typing Implementations. *arXiv preprint arXiv:2311.04527* (2023). https://doi.org/10.48550/arXiv.2311.04527

Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. 2021. Type-Centric Kotlin Compiler Fuzzing: Preserving Test Program Correctness by Preserving Types. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 318–328. https://doi.org/10.1109/ICST49551.2021.00044

Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2021. SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 899–913. https://doi.org/10.1145/3453483.3454084

Ross Tate, Alan Leung, and Sorin Lerner. 2011. Taming Wildcards in Java's Type System. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*.

Association for Computing Machinery, New York, NY, USA, 614–627. https://doi.org/10.1145/1993498.1993570

Pawel Urzyczyn. 1997. Inhabitation in Typed Lambda-Calculi (A Syntactic Approach). In *Proceedings of the Third International Conference on Typed Lambda Calculi and Applications (TLCA '97)*. Springer-Verlag, Berlin, Heidelberg, 373–389.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Jin Y. Yen. 1971. Finding the K Shortest Loopless Paths in a Network. *Manage. Sci.* 17, 11 (jul 1971), 712–716. https://doi.org/10.1287/mnsc.17.11.712

Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 347–361. https://doi.org/10.1145/3062341.3062379

Received 2023-07-11; accepted 2023-11-07