Mutation-Based Fuzzing of the Swift Compiler with Incomplete Type Information

Sarah Canto Hyatt
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, USA
sarahcantohyatt@ucsb.edu

Kyle Dewey
Department of Computer Science
California State University, Northridge
Northridge, USA
kyle.dewey@csun.edu

***Abstract*—Fuzzing statically-typed language compilers practically necessitates the generation of well-typed programs, which is a major challenge for fuzzing modern languages with rich type systems. Existing fuzzers only handle languages with simplistic type systems (e.g., C), only generate programs from small language subsets, or frequently generate ill-typed programs. In this work, we propose a mutation-based method for guaranteed well-typed program generation, even with minimal type system knowledge. With our method, we take a known well-typed seed program and annotate understood nodes with their types. We then try to replace annotated nodes with new type-equivalent ones, using a generator which fails if the input type is not understood. The end result is guaranteed overall well-typed as long as the original program was well-typed, even if most nodes are unannotated or the generator usually fails.**

We applied this approach to fuzzing the Swift compiler, and we are the first to fuzz Swift to the best of our knowledge. To implement our generator, we adapted constraint logic programming (CLP)-based fuzzing to work in a mutation-based context without a CLP engine, and this is the first such adaptation. Our fuzzer generates ∼22k programs per second, and we used it to find 13 Swift bugs, of which 7 have been confirmed or fixed by developers to date. Five bugs involved the compiler rejecting a well-typed program, and were only discoverable thanks to our well-typed generation guarantee.

***Index Terms*—Swift, Fuzzing, Compilers, Testing.**

I. Introduction

Compilers are critical pieces of computing infrastructure, and compiler bugs can silently change code behavior. This makes finding compiler bugs of utmost importance. Fuzzing is a popular technique for testing compilers (e.g., [1]–[6]), wherein a *fuzzer* automatically generates test input programs.

Technically any inputs will do, even random strings [7]. However, we must get past the compiler's front-end in order to test deeper components, and these deeper components usually account for the majority of the compiler's size and complexity (e.g., ∼82% by LOC for Swift). For statically-typed languages, this means generating well-typed programs. Modern statically-typed languages have many features complicating well-typed program generation, including subtyping, generics, higher-order functions, type classes [8] (e.g, Haskell, Swift's protocols, Rust's traits), or even affine types [9] (e.g., Rust). To maximize the number of discoverable bugs, a fuzzer must be able to produce well-typed programs using all these features; any bugs reliant on features not covered will not be found.

However, handling all typing features is a tall task. Modern languages generally lack complete formal or even informal specifications, and constantly evolve to add new features. This makes reasoning about well-typedness a blurry, moving target. To the best of our knowledge, fuzzers for statically-typed languages with complex typing features can either only generate programs according to small language subsets (e.g., Dewey et al. [10], SyRust [11], Chaliasos et al. [12]), or frequently generate ill-typed programs (e.g., Stepanov et al. [13]). While prior works phrase these issues merely as limitations of the specific fuzzing techniques or implementations devised, our key insight is these are actually manifestations of a fundamental limitation of what is realistically possible in well-typed fuzzing. Even with total type system knowledge, which is already unlikely, this knowledge alone does not provide a way to efficiently *generate* well-typed programs. For example, merely typechecking arbitrary Java is undecidable [14], so generating similarly arbitrary well-typed Java is likely at least as difficult. Statically-typed language fuzzers will therefore always be limited in general, necessitating fuzzing approaches that are highly tolerant of limited information.

With such limitations in mind, in this work, **we devise a fuzzing approach guaranteeing well-typed generation, even with incomplete type information and program generation capabilities**. Our approach is *mutation-based* (e.g., Holler et al. [2]), wherein an existing set of known well-typed programs (the *seed set*) is modified to produce new well-typed programs. A compiler's existing test suite can serve as a suitable seed set. From there, our approach uses three components: a parser, a possibly-incomplete typechecker-like type analysis, and a possibly-incomplete generator of well-typed expressions. The parser needs to be only complete enough to parse in the seed set, and the parser from the compiler under test may serve this purpose. The type analysis annotates AST nodes it understands with their respective types, and must guarantee that these annotations are correct. Any node it cannot handle for whatever reason (e.g., behavior still unimplemented, poorly-understood type system features, decidability issues, etc.) is left unannotated. The typechecker from the compiler under test may serve this purpose, if isolatable and trustworthy. The expression generator takes an input type and either produces an expression of that type, or fails to generate anything;

failure may occur for similar reasons as the type analysis. The generator must guarantee that the given expression generated is a well-typed expression of the input type.

Using these components, a program from the seed set is first parsed in and subsequently undergoes type analysis. From there, for every annotated node, we attempt to use the generator with the annotated type as input. The original node is either replaced by the new node on success, or simply left in place if the generator fails. This process preserves the original types, guaranteeing type-equivalence to the original program overall. Since the original program was well-typed, this means the new program is also well-typed, even if most nodes are unannotated or left in place. This approach generates programs using all the features in the seed set, even if some (or even most) of these features cannot be reasoned about or directly generated by the fuzzer. A more advanced typechecker or generator will lead to more possible output programs, but crucially this is not needed for the well-typedness guarantee.

We applied this approach to fuzzing Apple's Swift, which nearly 5% of all programmers reported using in 2024 [15]. Despite Swift's popularity, its only specification is incomplete user-facing documentation, based primarily on English and short code snippets [16]. Furthermore, there are no alternative Swift implementations, so we cannot run the same input on multiple compilers to find discrepancies (per differential testing [1]). These properties make Swift an excellent fuzzing candidate for our approach, as only limited typing information is available. We can also take advantage of the well-typed generation guarantee as an oracle, without needing other Swift implementations: all generated programs are well-typed, therefore any rejection indicates a bug. To the best of our knowledge, we are the first to fuzz Swift.

Swift's parser and typechecker were not designed to be separated from the compiler, nor did we consider them trustworthy, so we wrote our own. Our type analysis and generator only handle a subset of Swift with straightforward typing rules. For the generator, we innovated upon constraint logic programming (CLP)-based fuzzing [5], [10], which was originally designed to produce whole programs from scratch using CLP languages (e.g., SWI-Prolog [17]). We heavily modified CLP-based fuzzing to work for mutation-based fuzzing *without* CLP languages, and wrote a corresponding generator in Scala that could handle the same Swift subset as our type analysis. Despite our limited type analysis and generator, we found 13 bugs in the Swift compiler. Of these, 5 were cases where the compiler incorrectly rejected an input program; these would not have been discoverable without our well-typed generation guarantee. Nearly a quarter of all expression nodes are nonetheless understood and eligible for replacement, despite our limitations, suggesting only minimal type system knowledge is necessary to build a basic fuzzer with our approach. We also study the effect of additional type system knowledge on bug-finding power.

The contributions of this paper are:

- A mutation-based approach to fuzzing statically-typed languages guaranteeing well-typedness, even with limited type information or generation capabilities. (Section III)
- An adaptation of CLP-based fuzzing for mutation-based testing without a CLP language. (Section IV)
- The application of this approach to Swift. (Section V).
- An evaluation detailing the bugs we found, generation speed, how many nodes are replacable, and how bug-finding effectiveness changes as more type system information is provided. (Section VI)

II. Related Work

CSmith [3] generates well-typed C programs devoid of undefined behavior from scratch, via the help of a conservative program analysis. The analysis avoids issues related to C's *dynamic* semantics, whereas we focus on *static* semantics (i.e., types). C's type system is fairly trivial, simplifying well-typed C generation. Comparatively, avoiding undefined behavior is a major challenge, and is CSmith's major innovation. We posit that this innovation is possible only because C has a complete (albeit informal) semantics [18], making this fuzzing approach inaccessible for languages without complete specifications.

Dewey et al. [5], [10] introduce CLP-based fuzzing, wherein entire well-typed programs are generated from scratch via a CLP engine. They apply CLP-based fuzzing to JavaScript and Rust, finding 18 bugs in Rust. With CLP-based fuzzing, one writes a typechecker for the language being fuzzed in CLP. Normally, a typechecker is a predicate which determines if a given input program `P` typechecks (i.e., `typechecks(P)`), and CLP allows such execution. However, CLP also allows `typechecks` to *generate* satisfying programs, i.e., give all `P`s such that `typechecks(P)` holds. Dewey et al. state this is unique to CLP, making their work dependent on CLP engines. Furthermore, in practice, non-trivial optimizations are needed to write efficient CLP-based program generators, practically requiring significant CLP expertise. Overall, this severely limits CLP-based fuzzing's practical applicability [12], [19].

Our work is primarily inspired by LangFuzz [2], a mutation-based approach wherein AST nodes in a seed set are replaced with either existing nodes, or new nodes generated according to a context-free grammar (CFG). LangFuzz was applied to two dynamically typed languages, JavaScript and PHP, finding bugs in both. However, CFGs only guarantee syntactic validity, and less than 1% of randomly-generated syntactically-valid programs are well-typed [13], making LangFuzz inappropriate for fuzzing statically-typed languages like Swift.

Multiple mutation-based approaches have been devised which attempt to generate well-typed programs. JATTACK [20] allows developers to write template Java programs containing "holes", where the holes are filled-in using developer-written generators in a provided API. Developers must manually ensure their generators produce well-typed expressions of an appropriate type for the hole, and the API effectively forces generators to use a CFG, introducing the same issues as LangFuzz and severely limiting what can be generated. Their approach requires the template program to be executed to fill in holes, leading to poor generation rates between 5-77 programs per second, depending on the template.

Chaliasos et al. [12] developed a generator of well-typed expressions according to a custom IR. They then translate programs in this IR to programs in Java, Kotlin, or Groovy, and subsequently mutate the programs to either remove type annotations (testing type inference), or replacing type annotations with incompatible ones (to test if ill-typed programs are properly rejected). Only language features in the IR and translation are present in target programs, limiting the features that can be tested in the languages under consideration.

Sotiropoulos et al. [21] expand on the work of Chaliasos at al. [12] by introducing program generation based on API program synthesis. They use API graph enumeration to generate all possible calls to an API and ensure generation of well-typed programs by skipping any APIs for which they cannot infer concrete types. Their approach completely omits API calls that they cannot provide a well-typed call for, which they admit poses a challenge for applying this to languages with more sophisticated type systems like Rust or Typescript.

Zhang et al. [22] introduce *skeletal program enumeration* (SPE), wherein variables in a given input program are swapped with other in-scope variables of the same type. They found hundreds of bugs in multiple C compilers. SPE only can vary variable usage, limiting the sorts of programs it can produce.

Stepanov et al. [13] is arguably the most similar work to ours. They introduce *type-centric enumeration* (TCE), and use it to fuzz Kotlin. From a high level, TCE, like our approach, also identifies typed holes in a seed program, and attempts to fill those holes using expressions of the same type. However, the generation approach used is fixed in TCE, and is arguably far more complex. TCE starts with a generation phase, wherein expressions of known types from a seed set of programs are collected. TCE then constructs larger expressions using these initial seed expressions by applying rules describing call-like expressions in Kotlin. This means their generator itself requires a seed set, and it only generates call-like expressions. These constructed expressions are then used to fill in holes of the same type in the original seed set during a separate mutation phase. There is no guarantee that anything the generation phase produces will be used in the mutation phase, introducing possible performance issues. Furthermore, the generation phase uses snippets which may not work outside of their original context (e.g., using a variable), necessitating a Kotlin-specific semantics-aware merge operation to reintroduce missing components. This leads to large programs likely to contain lots of irrelevant code.

Most importantly, TCE does not guarantee well-typed generation, with only $\sim$63% of generated programs being well-typed. They also evaluate TCE with some additional Kotlin typing information (TCE + EM in their paper), which seems to be more effective at finding bugs, but with an even lower well-typed generation rate of $\sim$13%. While different numbers of bugs found are reported, looking specifically at TCE, TCE found 18 unique bugs considered "interesting" according to the Kotlin developers (21 are explicitly mentioned, but 3 of these were duplicates). No exact numbers for "uninteresting" TCE-discovered bugs are reported.

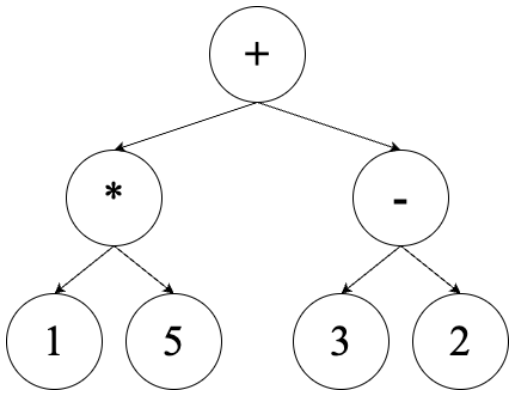

Fig. 1. `(1 * 5) + (3 - 2)` after initial parse to an AST.

III. Approach

In this section, we discuss our approach for generating guaranteed well-typed programs even in the absence of most typing knowledge. We take a two-phase approach: type analysis, and mutation. The rest of this section discusses these phases, as well as key properties this approach has overall. Throughout this section, we use an illustrative example wherein an input program `(1 * 5) + (3 - 2)` is ultimately used to generate a new well-typed program.

A. Type Analysis

We assume the existence of a seed set of programs that are known to be well-typed. A compiler's test suite should be adequate for this purpose, especially if it uses a large set of features found in the language. From there, we randomly select an input seed, and parse it to an AST. For our example, this leads to the AST in Figure 1.

We then apply a variation of typechecking to the AST in order to annotate AST nodes with their types. Unlike traditional typechecking, the goal is not to verify well-typedness, as we know the program is well-typed from our initial assumption. The goal instead is to annotate as many AST nodes as possible with their corresponding types. Any nodes we cannot reason about are left unannotated, instead of reporting a problem. This phase thus resembles a conservative program analysis instead of traditional typechecking. Similar to program analysis, soundness here is key: it is perfectly acceptable to leave nodes unannotated, but if a node is annotated, the annotation must be correct to guarantee downstream well-typed generation.

With respect to our running example, we apply our type analysis in Figure 2. For expository purposes, our analysis cannot reason about multiplication, and so the multiplication node remains unannotated. Its child nodes, however, are understood, and therefore annotated. Addition is understood by the type analysis only if the types of both children are known. In this case, since addition's left child lacks a known type, the addition overall lacks a known type.

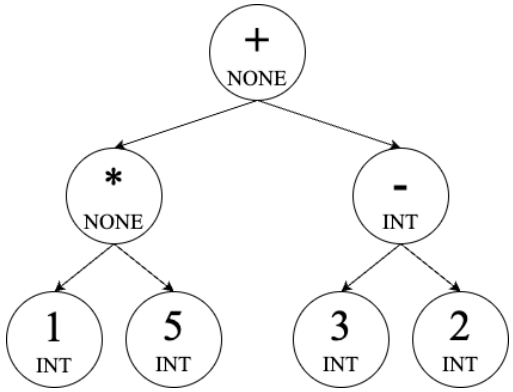

Fig. 2. `(1 * 5) + (3 - 2)` (Figure 1) after type annotation. `NONE` means that a node does not have a known type. The type analysis here cannot understand multiplication, nor addition with a child with an unknown type.

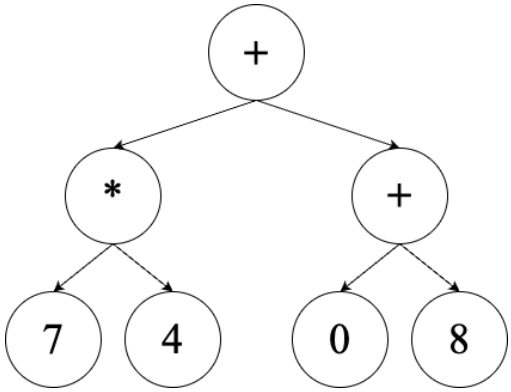

Fig. 3. `(1 * 5) + (3 - 2)` (Figure 2) after mutation.

B. Mutation

Next is the mutation phase, which assumes the existence of an expression generator with the following signature:

```
def generateExp(ofType: Type): Option[Exp]
```

Given a type `ofType`, `generateExp` will attempt to produce a new node of that type. The only requirements are that the generated node must be of type `ofType`, and the node must overall be well-typed. If the generator is unable to do this for whatever reason (e.g., incomplete implementation), the generator instead fails, hence the result type of `Option`.

For each annotated node, we can execute the generator with the node's type. If the generator produces a new node, then this new type-equivalent node replaces the original one. Otherwise, the original node is left as-is. As long as one node is replaced in this manner, a new, type-equivalent program is produced. Since the original program was well-typed, then the resulting program is also well-typed, even if most nodes lack known types, or the generator failed to generate something.

Figure 3 shows the result of the mutation phase on our running example. Unannotated nodes from Figure 2 remain in place, whereas all annotated nodes may be replaced. The entire subtraction node was replaced with the node for `0 + 8`, which is type-equivalent with type `INT`. Although the multiplication node was unannotated, its children were annotated, and so they were replaced with type-equivalent `7` and `4` nodes.

C. Key Properties

There are two key properties this approach has. For one, guaranteed well-typed outputs serve as their own testing oracles. That is, if the compiler rejects one, we know we have a compiler bug wherein the compiler failed to accept a well-typed program. This lets us detect more than just crashes, and does not require an alternative compiler version like differential testing [1] would. Fuzzers without this guarantee have no way to differentiate between an incorrect rejection from the target compiler, versus the fuzzer generating an invalid program, leading to missed bugs.

Another key property is that this approach is general, and amenable to growing. The type analysis can be improved to annotate more nodes, meaning more replaceable nodes for the mutation phase. Similarly, practically any generation approach could be used for our prior definition of `generateExp`; the more nodes generable, the larger the space of possible output programs. Crucially, such improvements neatly fit into our general fuzzing approach.

IV. Adapting CLP-based Fuzzing

In this section, we discuss our approach for generating well-typed expressions, and show how to generate such expressions via a code example. This generator can be used as part of our overall fuzzing approach (Section III). Our generation approach relies on a heavily-modified version of CLP-based fuzzing [5], [10]. CLP-based fuzzing only permits whole program generation and requires CLP languages, though our approach lifts these restrictions.

Looking at the CLP features enabling CLP-based fuzzing, we found that only two are essential: unification and nondeterministic execution. With those features, CLP-based fuzzing can be performed, independent of the fuzzer implementation language. We discuss unification's importance first, then nondeterministic execution. Unification is a way to implement equality constraints over data, where the data may contain variables. Unification has been used in typechecking and type inference for decades [23], [24], and is used in some recent work on well-typed program generation [12], [25].

We motivate unification for generation via an example generating well-typed Java. In Java, + is overloaded over multiple types, including `int`, `long`, and `double`. If we generate `l + r` for subexpressions `l` and `r`, then we must record that `typeof(l + r) = typeof(l) = typeof(r)`, ignoring implicit type coercions. Initially, multiple types may work for `l + r`, and so its type should be left unconstrained. Unification can record such equalities without prematurely choosing a particular type. With respect to our approach, unification libraries exist (e.g., `unification-fd` [26]), making CLP unnecessary for unification.

As for nondeterministic execution, this allows for a computation to have $0-N$ answers, instead of the usual $0-1$ (where 0 may indicate an error). For example, if solving for integer n in $0 \leq n \leq 10$, there are multiple possible solutions. Key to CLP's nondeterministic execution is that the computational cost of finding a given solution is proportional to that solution, as opposed to all solutions, in contrast to a traditional SMT solver with blocking clauses [27]. Nondeterministic execution is useful for generation because there are often many different expressions of a given desired type. Nondeterministic execution can be implemented as a library (e.g., MiMIs [28]), making CLP unnecessary for nondeterministic execution.

The challenge then is to combine unification and nondeterministic execution into a single coherent library. With such a library in hand, there is no need for CLP languages to perform CLP-based fuzzing. The rest of this section details our combination approach, the library we developed, and the application of this library to generate well-typed programs. Code in this section is written in Scala, as we used Scala to write our fuzzer, and the library itself is a paper contribution. However, the same ideas should be implementable in any other language supporting mutable state and higher-order functions.

A. Unification

Unification operates over *terms*, which obey this grammar:

```
v ∈ Variable     s ∈ StructureName

term ::= v | s `(` term* `)`
```

This term grammar mimics Prolog's, and can represent types as terms. For example, `Int` is representable with `Int()` (a structure named `Int` with no subterms). Type `List[String]` is representable with `List(String())`. Variables represent unknown types.

We implement unification via a map of variables to terms, similar to a union-find data structure [29]. For example, unifying variables `X` and `Y` stores $X \mapsto Y$ in the map. This map forms a graph where nodes are terms and edges dictate that two terms are equivalent. For any path through the graph, all nodes in the path are equivalent. Each term in the graph has a *representative*, which is the last node in a path. To unify two terms, we first find their representatives, and either add new edges and terms if the set representatives are compatible, or *fail* if they are incompatible. For example, variable `X` and term `Bool()` are compatible, but `Bool()` and `Int()` are not. Structures are compatible only if they have the same name, number of subterms, and their subterms recursively unify.

From a user's perspective, unification is representable in an immutable style as follows:

```
def unify(map: Map[Variable, Term],
          t1: Term, t2: Term):
  Option[Map[Variable, Term]]
```

That is, `unify` takes an existing map of variables to terms, as well as two terms to unify. On success, it returns a new map, or `None` if unification fails (hence the `Option` result type).

```
and(or(singleton(1), singleton(2)),
  a => and(or(singleton(3), singleton(4)),
        b => singleton((a, b)))))
```

Fig. 4. Code constructing an iterator over pairs of integers.

B. Nondeterminism

Dewey et al. [28] implements nondeterministic execution via iterators; this subsection covers background from that work. Instead of looking at type `Iterator[A]` (for some `A`) as a single deterministic value, we instead view it as a computation that nondeterministically produces values of type `A`, where iterating over the iterator retrieves individual `A`s. From there, larger nondeterministic computations can be composed from smaller ones using four helper functions:

```
def fail[A]: Iterator[A]
def singleton[A](a: A): Iterator[A]
def or[A](it1: Iterator[A],
          it2: Iterator[A]): Iterator[A]
def and[A, B](it: Iterator[A],
  f: A => Iterator[B]): Iterator[B]
```

`fail` returns an empty iterator, so named because it is used when there are no solutions. `singleton` takes a given element and returns an iterator over that element, putting deterministic values into a nondeterministic context. `or` creates a new iterator that initially iterates over everything in `it1`, and then seamlessly switches to `it2`, analogous to list append. Lastly, `and` takes an iterator over `A`s, and produces an iterator over `B`s with `f`'s help. Intuitively, `f` is applied to each `A` from `it`, and the resulting `Iterator[B]`s are merged together via `or`; this can be done lazily to delay calls to `f` until they are necessary. `and` is so named because it threads the result of one nondeterministic computation into another. Figure 4 shows an example constructing an iterator over integer pairs according to $\{(a,b)|a \in \{1,2\}, b \in \{3,4\}\}$.

Iterators can be constructed on demand, delaying actual work until iteration begins. The actual work performed is thusly proportional to the number of solutions iterated over, instead of the total number of solutions. This captures CLP's semantics, which similarly computes solutions on demand.

C. Combining Unification and Nondeterminism

The `unify` operation from Section IV-A and the four functions from Section IV-B can be used together as-is. However, it is cumbersome to do so, as the map must be threaded through all computations. This leads to awkward signatures wherein every nondeterministic computation takes a map and returns a new map along with the actual result of the computation. A possible solution is to make the map global and mutable, as CLP does internally [30]. However, this would require saving and restoring the map as different solutions are computed, which is complex. We instead use a state monad [31], [32] to track the map, hiding it from signatures. This is illustrated below, where `UIterator` represents our core abstraction:

```
exp ::= `0` | `"foo"` | exp1 `+` exp2
```

Fig. 5. Expression grammar over strings, integers, and an overloaded + that operates over a single type.

```
type UIterator[A] =
  (Map[Variable, Term]) =>
    Iterator[(Map[Variable, Term], A)]
def unify(t1: Term,
          t2: Term): UIterator[Unit]
```

A `UIterator[A]` (unification + iterator) is a higher-order function that takes a map and returns an iterator over map, `A` pairs. Intuitively, `unify` now returns a function that will later take and return the map, instead of requiring the map upfront. The helper functions have nearly the same signatures, but are over `UIterator` instead of `Iterator`. User code now constructs `UIterator`s but otherwise does not change; for example, Figure 4's code still works as-is, though it now constructs a `UIterator`. For `or`, the same input map is passed to both `it1` and `it2`. For `and`, the input map is first passed to `it` to yield a new map m', and m' is subsequently passed to the `UIterator` returned from `f`. The new two-argument version of `unify` internally checks the result of the three-argument version of `unify` from Section IV-A, and calls `fail` if unification failed. If unification succeeds, this only changes the map, which is now hidden away in `UIterator`'s definition. `Unit` is used as a dummy value; on unification success, there will be one unit value available, but on failure, there will be nothing to iterate over.

D. Performing Well-typed Generation

With `UIterator`, we can generate guaranteed well-typed code. Consider the grammar in Figure 5, where + is over either two integers or two strings, but not a mixed combination. In this grammar, `1 + 2` is well-typed, but not `3 + "foo"`.

Figure 6 shows a `UIterator`-based generator of well-typed expressions of depth $< k$ according to Figure 5's grammar. A line-by-line breakdown of Figure 6's code follows:

- Lines 2-3: If `k` is exceeded, outright fail.
- Lines 5-7: If our expected type `ofType` unifies with `Int()`, then generate integer literal `0`.
- Lines 8-10: If our expected type `ofType` unifies with `String()`, then generate string literal `"foo"`.
- Lines 11-14: Chain two recursive calls to `genExp` to make a `Plus` expression from the generated subexpressions. Since both `genExp` calls take `ofType`, both calls will produce expressions of the same type.
- Lines 15: The overall result is the nondeterministic choice of the last three bullets.

The initial `genExp` call must provide k and a term representation of the type to generate for. `genExp` will subsequently return a `UIterator` (a higher-order function), which can be called with an empty map (for unification) to return an `Iterator`. From there, the iterator can be traversed to produce different well-typed expressions of the provided type.

E. Code Cleanup

Figure 6 can be made more readable via helper methods to create structures representing our types. We can similarly introduce a multi-argument `or` which chains two-argument `or` calls, similar to what line 15 of Figure 6 does explicitly. We can also take advantage of Scala's *for comprehensions* [33], allowing for chained uses of `and` without nesting. For comprehensions require `UIterator` to be a separate trait (or class), making `and` a method which implicitly takes its `UIterator` via `this`, and renaming `and` to `flatMap`. These changes turn `UIterator` into a Scala-recognizable monad, where `and` is the bind operation [28]; Scala calls bind `flatMap`, and for comprehensions are designed to work with monads in the same way as Haskell's do notation [34]. Figure 7 is an updated version of Figure 6 reflecting these changes.

V. Implementation

We applied our technique to fuzzing Swift's compiler, `swiftc`. Swift's parser and typechecker are not designed to be separated from the compiler, nor did we consider these components trustworthy, so we wrote our own parser and type analysis. Parsing Swift, despite the presence of a CFG [35], is surprisingly challenging due to a combination of documentation issues and syntactic ambiguities. We implemented our parser via a combination of Scala's parser combinators [36], [37] and a custom library for handling ambiguous grammars.[1] We believe this initial obstacle heavily contributes to why no one has previously fuzzed Swift.

Our type analysis only handles a small Swift subset, namely literals, variables, infix expressions, ternary expressions, prefix expressions, assignment expressions, parenthesized expressions, arrays, and named functions. The only types natively understood are `Int`, `Double`, `Bool`, `Char`, `String`, and `Void`. Other types are handled by our type analysis by relying on type annotations in the input program. The analysis recursively traverses the AST and maintains a mapping of in-scope variables to their types and function names to their signatures, hereafter referred to as a *type environment*. For each AST node, the analysis annotates the node with its type, if possible. The analysis also saves the current type environment on the node, which is used later by the mutation phase. For annotated variable declarations, the variable is added to the type environment, even for unknown types. For example, with `var x: MyClass = MyClass(5)`, `x` $\mapsto$ `MyClass` is added to the type environment, even though `MyClass` is not natively understood. For unannotated variable declarations (e.g., `var x = 7`), if the initializer's type cannot be determined, then the variable is not added to the type environment. For named functions, the function's name is mapped to its signature.

Our generator operates over a similar subset of expressions as the type analysis, namely integer literals (as decimal, binary,

[1]The library is yet unpublished and not considered a contribution here.

```
def genExp(k: Int, ofType: Term): UIterator[Exp] = {
  if (k <= 0) {
    fail
  } else {
    val intLit =
      and(unify(ofType, Structure("Int", Seq())),
          _ => singleton(IntLiteral(0)))
    val stringLit =
      and(unify(ofType, Structure("String", Seq())),
          _ => singleton(StringLiteral("foo")))
    val plus =
      and(genExp(k - 1, ofType),
          e1 => and(genExp(k - 1, ofType),
                    e2 => singleton(Plus(e1, e2))))
    or(intLit, or(stringLit, plus))
  }
}
```

Fig. 6. Nondeterministic generator of well-typed expressions according to the grammar in Figure 5.

```
def intType: Term = Structure("Int", Seq())
def stringType: Term = Structure("String", Seq())
def genExp2(k: Int, ofType: Term): UIterator[Exp] = {
  if (k <= 0) {
    fail
  } else {
    or(
      for {
        _ <- unify(ofType, intType)
      } yield IntLiteral(0),
      for {
        _ <- unify(ofType, stringType)
      } yield StringLiteral("foo"),
      for {
        e1 <- genExp(k - 1, ofType)
        e2 <- genExp(k - 1, ofType)
      } yield Plus(e1, e2))
  }
}
```

Fig. 7. Equivalent version of the code from Figure 6, using helper functions and for comprehensions for improved readability.

octal or hex), double literals (as decimal or hex), boolean literals, strings, chars, parenthesized expressions, binary expressions, prefix expressions, ternary expressions, variables, assignment expressions, arrays, and named function calls. Our generator takes a bound on the depth of generated ASTs, and exhaustively generates all ASTs within that depth without duplicates. For `or`, our generator randomly chooses the order in which to explore child iterators. For literal nodes (e.g., `42`, `3.14`), it randomly selects one value instead of nondeterministically trying others, given how many values are possible.

To produce a new program, we recursively traverse the annotated AST, producing a new AST in the process. For each unannotated node, we recursively process the node's children, then copy the node with the possibly new children for the output AST. For each annotated node, we nondeterministically choose either to process the node as if it were unannotated, or select a replacement node from our generator. The generator has a nearly-identical signature to Figure 7, but a type environment is additionally passed in; the node being replaced provides this type environment, which came from the type analysis phase. The generator's body consists of a large `or` selecting between different kinds of generable AST nodes. This body is too large to show in its entirety, but key portions are below, with some modification to improve exposition.

The following code generates an integer literal in base 10:

```
for {
  _ <- unify(ofType, intType)
  n <- or(singleton(0), singleton(1))
} yield DecimalIntegerLiteral(n.toString)
```

This code first ensures we may generate something of type `Int` via the unification. It then nondeterministically selects $n \in \{0, 1\}$ via `or`, and outputs n's string representation.

The code below generates infix expressions of depth $\leq k$. `NewVariable` creates new unification variables.

```
def infixHelper(lt: Term, rt: Term,
  ofType: Term): UIterator[Operator] = {
  or(
    for {
      _ <- unify(lt, intType)
      _ <- unify(rt, intType)
      _ <- unify(ofType, intType)
    } yield Operator("+"),
    for {
      _ <- unify(lt, boolType)
      _ <- unify(rt, boolType)
      _ <- unify(ofType, boolType)
    } yield Operator("&&")
    ...) // more operators elided
}
val lt = NewVariable()
val rt = NewVariable()
for {
  op <- infixHelper(lt, rt, ofType)
  l <- genExp(lt, k - 1, env)
  r <- genExp(rt, k - 1, env)
} yield InfixExp(l, op, r)
```

`infixHelper` looks for an operator which returns `ofType`, given two input expressions of types `lt` and `rt` (representing the left and right subexpression types, respectively). While only two operators are shown here, the real `infixHelper` considers 43 possible operators, and is implemented with minimal code duplication. Most operators are overloaded (e.g., +), and are repeated for each set of possible types. Once `infixHelper` finds a suitable operator, this code recursively generates subexpressions of types `lt` and `rt`, subtracting one from the bound k to limit the overall expression size. Finally, once subexpressions `l` and `r` are generated, they are bundled together into a single AST node with the chosen operator.

The code below attempts to generate an in-scope variable of type `ofType`. The `toUIterator` helper converts a regular iterator to a `UIterator`, and the `iterator` method on `env` (the type environment) returns an iterator over variable, type pairs. The `typeToTerm` helper takes a type and returns its term-based representation, suitable for unification.

```
for {
  (name, typ) <- toUIterator(env.iterator)
  _ <- unify(ofType, typeToTerm(typ))
} yield VariableExp(name)
```

This code nondeterministically chooses a variable from the type environment, and attempts to unify the variable's type with `ofType`. If unification succeeds, then a node with the variable's name is emitted. If unification fails, a different variable will be nondeterministically selected for the same process. This continues until either a suitable variable is found, or all variables in the type environment have been considered. Since the type analysis phase includes variables in the type environment from explicit type annotations, even for unknown types, this generator can produce expressions of types it otherwise cannot generate. In other words, as long as variable x is in scope and is explicitly annotated to be of type τ, this generator can generate x if `ofType` unifies with τ, even if the generator otherwise cannot generate τ expressions.

VI. Evaluation

In this section, we evaluate our fuzzer with respect to four research questions:

- **RQ1**: How many and what kinds of bugs does it find in Swift? This speaks to overall bug-finding effectiveness.
- **RQ2**: How quickly can it generate Swift programs? This measures the computational cost of our approach, and is independent of bug-finding power.
- **RQ3**: How many nodes does it annotate with types? More nodes annotated mean more opportunities to generate new programs.
- **RQ4**: How does bug-finding effectiveness change as more type system knowledge is used? Our approach allows us to at least begin to answer such a question by incrementally adding more features to the same fuzzer.

To evaluate these questions, we use a seed set of 3,444 programs that were sourced from the Swift repository [38]. We configured our fuzzer to operate at increasing levels of generation capability, detailed below:

- **Level 1**: only literals (integers, floats, chars, static strings, booleans) and parenthesized expressions
- **Level 2**: level 1 plus prefix expressions, binary expressions, and ternary expressions
- **Level 3**: level 2 plus variables, assignment expressions, arrays, and named function calls

We ran the fuzzer on each seed file for each level, using a max depth of 2, producing one *batch* of files for each seed and level. Because the fuzzer uses bounded-exhaustive search with a large nondeterministic state space, many billions of unique programs are generable, which is prohibitive for experimentation. As such, we set a timeout on the fuzzer of 5 seconds for level 1, 10 seconds for 2, and 15 seconds for 3. We then ran each batch of files on `swiftc` version 5.8.1, with a timeout of one minute for the whole batch; this timeout was due to a limited testing budget. In total, $\sim$439M programs were generated, and $\sim$1.4M of these were run on `swiftc`. All experiments were run on a MacBook Air with an M2 processor and 24 GB RAM. We divide up the rest of our evaluation according to these research questions.

```
1 func example(_ f: (Int) throws -> Int)
2   rethrows -> Int { return try f(7); }
3
4 print(example({x in x})); // ok
5
6 let hof = example;
7 // hof inferred type:
8 // ((Int) throws -> Int) throws -> Int
9 print(hof({x in x})); // compile error
```

Fig. 8. Bug 3: `throws` and `rethrows` interacting poorly with higher-order functions, leading to program rejection. The original bug trigger was simplified for expository purposes.

A. RQ1: Number and Kinds of Bugs Found

We found 13 bugs (Table I), divided into three classes:

- Rejection (R): `swiftc` rejected a well-typed program.
- Crash (C): `swiftc` crashed.
- Specification (S): Swift's documentation (effectively Swift's specification) diverges from `swiftc`'s behavior.

S-class and R-class bugs are distinguished based on where the bugfix was placed or likely needs to be placed. To find R-class bugs, we must know ahead of time that the compiler should accept a program, making our guaranteed well-typed generation approach well-suited to finding R-class bugs. We also list the fuzzer level(s) which found the bug. All bugs occur on Swift versions 5.8.1 and 5.10, except for Bug 11. Bug 11 was fixed by 5.10, and to the best of our knowledge, this bugfix was unintentional. The rest of this subsection discusses a subset of the bugs we found, divided by bug class.

1) R-class Bugs: Figure 8 is a case where the compiler improperly rejected a program in a manner indicating a deeper issue in Swift's language design. Swift requires exception-throwing functions to be annotated with `throws`, and calls to exception-throwing functions must use `try`. Swift allows functions taking higher-order functions as parameters to be annotated with `rethrows`, meaning an exception is only thrown if the provided higher-order function throws an exception. If users pass higher-order functions that do not throw exceptions to `rethrows`-annotated functions, then `try` is unnecessary. However, an oversight is that `rethrows` can currently only be used on named (not higher-order) functions, leading to inconsistent behavior. For example, on line 4 of Figure 8, `example` can be called directly with a non-throwing higher-order function without `try`. However, if `example` is assigned to a variable to create a higher-order function (line 6), Swift infers a return type annotated with `throws` instead of `rethrows` (line 8). This causes a compilation failure on line 9, as `hof`'s call thus needs a superfluous `try`. This is not merely a type inference issue, but rather a design defect, as `hof` cannot be manually annotated with `rethrows`, either.

2) C-class Bug: The code in Figure 9 crashes `swiftc` during the Swift Intermediate Language generation phase on line 5. The parentheses around `try` are needed to trigger the crash, though they do not change the program's behavior.

```
1 class Class {
2   var num: Int = 1
3   init() throws {}
4 }
5 _ = (try Class().num)
```

Fig. 9. Bug 10: A compiler crash involving `try` in unnecessary parentheses.

```
1 enum A {}
2 struct MyStruct {  var a: A  }
3 func myFunc(_: inout A) {}
4 func myOtherFunc(s: inout MyStruct) {
5   myFunc(&s.a)
6 }
```

Fig. 10. Program combining in-out and dot (`.`) expressions.

Further analysis revealed this bug was present as early as Swift 5.5, so it lurked for nearly three years. Overall, our fuzzing efforts revealed that semantically meaningless parentheses had surprising impacts on the compiler's behavior, with bugs 6, 9, 10, and 11 all dependent on unnecessary parentheses.

3) S-class Bugs: Most S-class bugs we found are cases where Swift's grammar specification [35] deviates from the compiler's behavior. For example, according to Swift's original grammar, line 5 of Figure 10 should be rejected, as only identifiers were permitted to follow `&`. After reporting this to developers, the grammar was updated to allow any `primaryExp` to follow `&`. However, this fix is incomplete, as this code should *still* be rejected. To understand why, Figure 11 shows a snippet of the new grammar. The new grammar states that `s.a` is an `explicitMembExp`, which is only derivable from `postfixExp`, *not* `primaryExp`. Bug 4 in Table I represents this issue, and we mark it as only partially fixed.

B. RQ2: Generation Rate

The fuzzer took 5:26:25(hh:mm:ss) to produce 439,096,876 mutant programs across all levels, which is approximately 22,448 mutants generated per second. This far exceeds the rate at which `swiftc` can compile them; in total, only 0.32% of these (1,413,648) were compiled on `swiftc`, which

```
exp ::= [tryOp] [awaitOp] prefixExp [infixExps]
prefixExp ::= [prefixOp] postfixExp | inOut
postfixExp ::= primaryExp | explicitMembExp
explicitMembExp ::=
    postfixExp `.`
    identifier [genericArgumentClause]
```

Fig. 11. Subset of the new Swift expression grammar showing that Figure 10 should still be rejected. Names and irrelevant productions are edited for space.

TABLE I

BUGS FOUND, USING BUG CLASSES FROM SECTION VI-A. "LEVEL" INDICATES WHICH FUZZER LEVEL(S) FOUND THE BUG, USING THE LEVELS FROM SECTION VI; "N/A" MEANS THE BUG WAS FOUND DURING FUZZER DEVELOPMENT. ✓* SIGNIFIES EITHER AN INCOMPLETE OR UNRELEASED BUGFIX.

Bug ID	Bug Class	Level	Reported	Confirmed	Fixed	Description
1	S	N/A	✓	✓	✓	Unreachable production in grammar
2	S	N/A	✓	✓	×	Some "reserved words" are actually undocumented keywords.
3	R	1, 2, 3	✓	×	×	Named functions allow for `throws` and `rethrows`, but higher-order functions only handle `throws`, creating issues when using a named function as a higher-order function. (Figure 8)
4	S	N/A	✓	✓	✓*	Grammar for in-out-expressions is overly restrictive. (Figure 10)
5	S	N/A	✓	✓	×	Grammar is missing semicolons for many productions.
6	R	2, 3	✓	×	×	Type inference failure involving parenthesized expressions and implicit type coercions
7	S	N/A	✓	×	×	`final` is permitted in actor declarations despite Swift not supporting actor inheritance.
8	S	N/A	✓	×	×	Grammar disallows `mutation` modifier in front of `willSet` and `didSet` clauses
9	S	1, 2, 3	✓	×	×	Grammar disallows parentheses around type identifiers
10	C	1, 2, 3	✓	✓	✓*	Compiler crash on certain parenthesized try expressions (Figure 9)
11	R	1, 2, 3	×	×	✓	Type inference failure with unnecessary parentheses, unreported as it was fixed in version 5.10
12	R	2, 3	✓	✓	×	Inconsistent loop termination analysis inside functions or methods
13	R	2, 3	✓	×	×	Unable to determine overloaded operator exactly equals in some cases

comparatively took 43:40:15. We conclude that our fuzzer rapidly generates well-typed programs.

C. RQ3: Nodes Annotated

The fewer nodes that can be annotated, the fewer the opportunities to replace a node and generate a new program. Theoretically, with a poor enough type analysis, no nodes will be annotated and so no new programs are generable. The seed set may also be biased towards program features that the type analysis cannot handle. In our experimentation, we found that 1,578 programs in the seed set lack expression nodes (∼46%), and therefore will never generate any mutants no matter how advanced our type analysis is. Of the remaining programs, 469 contained exclusively expressions our type analysis could not reason about (∼14% of all, ∼25% of programs containing expressions). This still left ∼40% of programs across the entire seed set containing at least one annotated node. Looking at all expression nodes overall, ∼26% of them were annotated. Considering how simplistic our type analysis is, this was higher than expected, and indicates there are nonetheless a number of opportunities to generate new programs.

D. RQ4: Bug-Finding Effectiveness As Knowledge Increases

Looking at Table I's "Level" column, level 1 was sufficient to find four bugs (3, 9-11), but at least level 2 was needed to find an additional three bugs (6, 12, 13). No bug was exclusive to level 3, so level 2 alone could find all bugs. This shows that additional type system knowledge may find additional bugs (moving from level 1 to 2), but not necessarily so (level 2 to 3). However, our current evaluation is insufficient to generalize this conclusion. We would need many more bugs found for this sort of evaluation, ideally all generated programs should be run on `swiftc` instead of less than 1% (Section VI-B), more levels and type system knowledge should be considered, and other languages would need to be fuzzed to see if this is a Swift-specific trend. We provide this data as a preliminary exploration into fuzzer effectiveness as type system knowledge increases. The key point is that our technique can at least begin to answer this question.

VII. CONCLUSION

Effective fuzzing of statically-typed languages requires generating well-typed programs, which is exceptionally difficult. We offer a mutation-based approach for guaranteed well-typed program generation, even with only limited type information. We can add additional type system knowledge to a fuzzer made with this approach, all without modifying the fundamental approach. We also adapted CLP-based fuzzing to work in a mutation-based context, without needing CLP. We created the first Swift fuzzer and used it to generate hundreds of millions of programs, exposing 13 bugs overall, demonstrating its efficiency and bug-finding capability. We lastly conducted a preliminary investigation into how bug-finding power changes with greater type system knowledge, showing that such a question is at least answerable with our approach.

For future work, we plan to update our type analysis and generator to handle more complex type information and more Swift AST nodes. This will allow us to expand our preliminary investigation and possibly find more bugs. We also plan to apply this approach to other statically-typed languages, particularly Rust [39], given both the complexity of its type system and the recent community interest in fuzzing it [11], [40], [41].

REFERENCES

[1] W. M. McKeeman, "Differential testing for software." *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, December 1998.

[2] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 38–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362831

[3] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532

[4] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 65–76. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737986

[5] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 725–730. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642963

[6] K. Dewey, L. Nichols, and B. Hardekopf, "Automated data structure generation: Refuting common wisdom," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 32–43. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818761

[7] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[8] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: ACM, 1989, pp. 60–76. [Online]. Available: http://doi.acm.org/10.1145/75277.75283

[9] B. C. Pierce, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[10] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using CLP," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 482–493. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.65

[11] Y. Takashima, R. Martins, L. Jia, and C. S. Păsăreanu, "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*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 899–913. [Online]. Available: https://doi.org/10.1145/3453483.3454084

[12] S. Chaliasos, T. Sotiropoulos, D. Spinellis, A. Gervais, B. Livshits, and D. Mitropoulos, "Finding typing compiler bugs," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 183–198. [Online]. Available: https://doi.org/10.1145/3519939.3523427

[13] D. Stepanov, M. Akhin, and M. Belyaev, "Type-centric kotlin compiler fuzzing: Preserving test program correctness by preserving types," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 318–328.

[14] R. Grigore, "Java generics are turing complete," *CoRR*, vol. abs/1605.05274, 2016. [Online]. Available: http://arxiv.org/abs/1605.05274

[15] Stack Overflow, "2024 developer survey," 2024. [Online]. Available: https://survey.stackoverflow.co/2024/

[16] Apple, "The swift programming language (5.10)," 2024. [Online]. Available: https://docs.swift.org/swift-book/documentation/the-swift-programming-language

[17] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[18] C Standards Committee, "C17 standard," 2017. [Online]. Available: https://www.iso.org/standard/74528.html

[19] N. A. Awar, K. Jain, C. J. Rossbach, and M. Gligoric, "Programming and execution models for parallel bounded exhaustive testing," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi-org.libproxy.csun.edu/10.1145/3485543

[20] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, "Compiler testing using template java programs," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556958

[21] T. Sotiropoulos, S. Chaliasos, and Z. Su, "API-Driven program synthesis for testing static typing implementations," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, jan 2024. [Online]. Available: https://doi.org/10.1145/3632904

[22] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 347–361. [Online]. Available: https://doi.org/10.1145/3062341.3062379

[23] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Transactions of the American Mathematical Society*, vol. 146, pp. pp. 29–60, 1969. [Online]. Available: http://www.jstor.org/stable/1995158

[24] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348 – 375, 1978. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000078900144

[25] B. Fetscher, K. Claessen, M. Pałka, J. Hughes, and R. B. Findler, "Making random judgments: Automatically generating well-typed terms from the definition of a type-system," in *Programming Languages and Systems*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 383–405.

[26] W. G. Romano, "unification-fd: Simple generic unification algorithms," 2023. [Online]. Available: https://hackage.haskell.org/package/unification-fd

[27] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. [Online]. Available: http://books.google.com/books?id=anJsH3Dq5BIC

[28] K. Dewey, S. Hairapetian, and M. Gavrilov, "Mimis: Simple, efficient, and fast bounded-exhaustive test case generators," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 51–62.

[29] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Commun. ACM*, vol. 7, no. 5, p. 301–303, may 1964. [Online]. Available: https://doi.org/10.1145/364099.364331

[30] D. H. Warren, "An abstract prolog instruction set," *Technical report*, 1983.

[31] "Haskell/understanding monads/state," 2023. [Online]. Available: https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

[32] P. Chiusano and R. Bjarnason, *Functional Programming in Scala*, 1st ed. USA: Manning Publications Co., 2014.

[33] Scala Development Team, "Tour of scala: For comprehensions," 2024. [Online]. Available: https://docs.scala-lang.org/tour/for-comprehensions.html

[34] "Haskell/do notation," 2021. [Online]. Available: https://en.wikibooks.org/wiki/Haskell/do_notation

[35] Apple, "Summary of the grammar," 2024. [Online]. Available: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/summaryofthegrammar/

[36] R. Frost and J. Launchbury, "Constructing natural language interpreters in a lazy functional language," *Comput. J.*, vol. 32, no. 2, p. 108–121, apr 1989. [Online]. Available: https://doi.org/10.1093/comjnl/32.2.108

[37] "scala-parser-combinators," 2024. [Online]. Available: https://github.com/scala/scala-parser-combinators

[38] "swift," 2024. [Online]. Available: https://github.com/apple/swift

[39] Mozilla, "The rust language website." [Online]. Available: http://www.rust-lang.org/

[40] M. Sharma, P. Yu, and A. F. Donaldson, "Rustsmith: Random differential compiler testing for rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1483–1486. [Online]. Available: https://doi.org/10.1145/3597926.3604919

[41] Q. Wang and R. Jung, "Rustlantis: Randomized differential testing of the rust compiler," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689780