FLAT: Formal Languages as Types

And Their Applications in Testing

FENGMIN ZHU, CISPA Helmholtz Center for Information Security, Germany
ANDREAS ZELLER, CISPA Helmholtz Center for Information Security, Germany

Programmers regularly use strings to encode many types of data, such as Unix file paths, URLs, and email addresses, that are conceptually different. However, existing mainstream programming languages use a unified string type to represent them all. As a result, their type systems will keep quiet when a function requiring an email address is instead fed an HTML text, which may cause unexceptional failures or vulnerabilities.

To let the type system distinguish such conceptually different string types, in this paper, we propose to regard *formal languages as types* (FLAT), thereby restricting the set of valid strings by context-free grammars and semantic constraints if needed. To this end, email addresses and HTML text are treated as different types. We realize this idea in Python as a testing framework FLAT-PY. It contains user annotations, all directly attached to the user's code, to (1) define such *language types*, (2) specify pre-/post-conditions serving as *semantic oracles* or contracts for functions, and (3) fuzz functions via random string inputs generated from a *language-based fuzzer*. From these annotations, FLAY-PY *automatically* checks type correctness at runtime via *code instrumentation*, and reports any detected type error as soon as possible, preventing bugs from flowing deeply into other parts of the code. Case studies on real Python code fragments show that FLAT-PY is able to catch logical bugs from random inputs, requiring a reasonable amount of user annotations.

CCS Concepts: • **Software and its engineering → Data types and structures**; **Software testing and debugging**; • **Theory of computation → Grammars and context-free languages**.

Additional Key Words and Phrases: Context-free grammars, Refinement types, Testing, Fuzzing, Python

ACM Reference Format:
Fengmin Zhu and Andreas Zeller. 2024. FLAT: Formal Languages as Types: And Their Applications in Testing. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2024), 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Processing structured data is a common task in developing various kinds of software. File system APIs usually takes file paths as inputs to locate and operate the corresponding resources. A network library must correctly validate and extract data fields such as a domain name from a URL or an email address, according to their RFC standards. A server backend typically encodes and decodes packets that are transmitted in standard data exchange formats like XML and JSON.

Ideally, values that represent file paths, URLs, email addresses, etc. should be given *different* types simply because they are conceptually different, for instance, by creating different class types in an object-oriented language or different algebraic data types in a functional language. However, in reality, programmers use a *unified* string type to rule them all, which eases the design of APIs and the exchange of data, particularly in Unix, where file paths, shell commands, and standard input/output streams are text-based.

Authors' addresses: Fengmin Zhu, fengmin.zhu@cispa.de, CISPA Helmholtz Center for Information Security, Saarbrücken, Saarland, Germany, 66123; Andreas Zeller, zeller@cispa.de, CISPA Helmholtz Center for Information Security, Saarbrücken, Saarland, Germany, 66123.

However, this golden practice ignores the latent structure of strings and can introduce potential bugs. Consider a Python function that sends a verification email to a user registered on a web service:

```
def send_verif_email(email: str) -> None
```

A static type checker (*e.g.,* mypy[1] and pyre[2]) happily accepts any string as the input argument `email`, including an XML string and an HTML string injected with malicious JavaScript code. Later, those undesired inputs can lead to unpredictable behaviors and even security issues that are hard to detect and repair.

One possible industrial solution is to perform additional input format validation through *regular expression* (regex) matching, supported in many mainstream programming languages like Python's `re` library. To ensure that the input `email` address is valid in `send_verif_email`, we may attach the following validation code at the beginning of this function:

```
1 import re
2 def send_verif_email(email: str) -> None:
3   pattern = re.compile(r'(?<=[^0-9a-zA-Z\!\#\$\%\&\'\*\+\-\/\=\?\^\_\`\{\|\}\~\-])' \
4     r'([a-zA-Z0-9_.-]+@[a-zA-Z0-9_.-]+(?:\.[a-zA-Z0-9]+)*\.[a-zA-Z0-9]{2,6})' \
5     r'(?=[^0-9a-zA-Z\!\#\$\%\&\'\*\+\-\/\=\?\^\_\`\{\|\}\~\-])') # regex pattern
6   assert pattern.fullmatch(email) is not None # ensure it matches the input email
7   ... # now can safely send email
```

The regex pattern shown in Lines 2-5 is taken from a Python library JioNLP[3]. It is not only *unreadable* but also *wrong*: it both over- and under-approximates the legal email addresses as defined in RFC 5322 [17] standard of Internet message format. For example, it will *reject* the legal email address `'name/surname@example.com'` (note that `'/'` is allowed) and *accept* the illegal email address `'a"b(c)d,e:f;g<h>i[j\\k]l@example.com'` (as `','` and brackets `'[]'` are disallowed)[4]. Regex patterns like this are unreadable, unusable, and may be too complex to be correct, thus are not reasonable options for expressing the exact intended structure of strings.

Such a regex-based input format validation approach has the following drawbacks: (1) unwieldy validation code (Lines 3-6 shown above) has to be inserted manually, which is labor-intensive and time-consuming; (2) regex patterns are less expressive and do not offer higher level formal language support as in *context-free grammars* (CFGs); and (3) no existing *language-based fuzzing* technique (*e.g.,* [9, 21, 25]) takes (Python's or JavaScript's) regex patterns as inputs.

To address these three issues, in this paper, we present a more general solution from a type system's angle based on the idea of regarding *formal languages as types* (FLAT). The formal languages we use are CFGs: they are well-defined, well-understood, and well-supported by language-based fuzzers. Even for a regular language, a well-structured CFG with reasonable names for nonterminal symbols is more readable than a compact yet mysterious regex pattern. Not even mention that some languages are indeed not regular, including the RFC-standard email addresses. A CFG gives rises to a *language type* whose inhabitants are restricted to the set of sentences accepted by that grammar, hence rejected malformed inputs. Attaching additional logical constraints to CFGs, users can express *context-sensitivity* such as restricting the lengths of strings in language types.

Applying FLAT, the above email address validation problem is solved in a simple way: all the users need is to annotate the input argument `email` with a language type for valid email addresses, say the FLAT built-in type `RFCEmail` that follows the RFC 5322 standard:

[1] https://mypy-lang.org
[2] https://pyre-check.org
[3] https://github.com/dongrixinyu/JioNLP/blob/e17dba0/jionlp/rule/rule_pattern.py#L43
[4] Both examples are taken from: https://en.wikipedia.org/wiki/Email_address#Examples.

```
def send_verif_email(email: RFCEmail) -> None
```

No email address regex pattern needs to be specified: it is replaced by the CFG defined in RFC 5322 standard. No regex-based validation code needs to be manually inserted: FLAT will *automatically* check if any input `email` matches the `RFCEmail` syntax via parsing *at runtime*, for each call to `send_verif_email`. If, for instance, feeding an XML string or a JavaScript-injected HTML string to this function, FLAT will report type errors at runtime. Note that this paper focuses on *runtime* type checking, as it is more general and easier to implement than static type checking (which we leave as a future direction).

Using a type-based approach, FLAT tackles several common tasks in software development in *one* framework and FLAT's automation *reduces* user efforts in testing:

API documentation. Conventionally, for string-typed APIs that require the inputs to have specific formats, developers describe such requirements as *comments* in the API documentation, which means they are invisible to type systems and can be neglected by careless developers. In FLAT, they are now parts of the *types*: they are *visible* to the type checker and are *enforced* to be fulfilled via runtime type checking.

Input validation. To make sure that the user-input strings are legal, the traditional method is to validate them through regex matching or parsing, which can be time-consuming and error-prone. FLAT saves API developer's time by automatically checking the input strings against the attached input types at runtime. Specifying a grammar is usually more intuitive and correct than specifying a potentially complex regex pattern, or it is easier than implementing a custom parser from scratch.

Malicious input detection. Another kind of invalid inputs are malicious inputs crafted by attackers. Mature industry solutions exist for *specific* scenarios, such as prepared statements for SQL injection and HTML sanitization for cross-site scripting (XSS). FLAT, on the other hand, offers a *general* solution to detect and reject malicious inputs in under-studied scenarios. For instance, to avoid dangerous Shell command injection like:

```
eval('__import__("os").system("rm -rf .")')
```

One may restrict the input to `eval()` within a secured fragment of Python expressions (*e.g.*, arithmetic only) via a language type representing the desired secured fragment.

Language-based fuzzing. For functions whose input strings are all annotated with language types, FLAT offers a convenient `fuzz` primitive to enable language-based fuzzing. For example, to test the function `send_verif_email` against 1,000 randomly-generated email addresses, all the developers need is a single line of code `fuzz(send_verif_email, 1000)`. FLAT takes over the process of translating the language types into the input grammars recognized by the fuzzer and setting up the entire testing pipeline.

Test oracles. Fuzzing is known to be effective in detecting program crashes. To find functional/logical bugs, additionally developers have to provide *test oracles*, which are usually hand-written programs that determine if an actual output is expected or not. Since the test oracles encode certain invariants or relationships between inputs and outputs, they can also be expressed as *pre-* and *post-conditions* of functions. In FLAT, developers do not have to provide test oracles manually but instead attach pre- and post-conditions as semantic *contracts* in a declarative manner. FLAT will detect contract violations at runtime for each test input generated by the fuzzer, thereby automatically detecting functional bugs.

Traditionally, users have to write *extra code* to accomplish each of the tasks above.

As with any other type system, the only user effort required by FLAT is to provide *type annotations* that specify the intended behaviors of their programs. FLAT already provides standard formats such as email addresses and JSON, and are subject to extension in the future. For custom formats,

their grammars are either already there (described in the documentation) or *should* be there, as a well-engineered API should contain a description of the required formats. In other case, the grammars are feasible to be specified and this process is worthwhile.

Contributions. Our main contribution is FLAT, a general framework for typing and type-checking string-manipulating programs, based on the idea of "formal languages as types". After beginning with an overview (§2) of the motivations and ideas behind FLAT by Python examples, we make the following detailed technical contributions:

- We define an abstract core language FLAT-CORE (§3, §4) to demonstrate FLAT's type system and runtime type checking. We rely on code instrumention to check if all input arguments, return values, and local variables match the annotated types.
- We realize the idea of FLAT in Python and present FLAT-PY (§5), a testing framework for Python with built-in language types for commonly used data formats (*e.g.*, email address, URL, JSON) and Python-style annotations.
- We conducted case studies (§6) on open-source Python code fragments. Requiring a reasonable amount of user annotations that are intuitive to specify, FLAT-PY identifies logical bugs from randomly generated test inputs.

After discussing related work in §7, we close the paper in §8 with conclusion and future directions. Our tool and data are publicly available: see §8 for the link.

2 A TOUR OF FLAT-PY

To demonstrate how one uses FLAT-PY to test and debug their code, let us consider an ad hoc parser:

```
1 def get_hostname(url: str) -> str:
2   """Extract the hostname part."""
3   start = url.find('://') + 3
4   end = url.find('/', start)
5   host = url[start:end]
6   return host
```

This function aims to extract a hostname from a given URL. It first computes the starting and ending positions of the hostname part, represented by the local variables `start` and `end` respectively. The `start` (Line 3) is the position right after `'://'`. This position is computed by adding the starting index of the pattern with its length three, where the starting index is obtained via a call of the `find` method. The `end` (Line 4) is the position of the first `'/'` seen afterward, which is obtained via a call of the `find` method within a range beginning with `start`. Finally, using these two positions, the hostname part is extracted as a substring of the input `url` (Line 5) and returned (Line 6).

Let us now imagine that we wish to save the extracted hostname in a database table named `hosts`. We achieve this by executing a SQL query that is instantiated from the following template, where `<host>` is the metavariable we will substitute:

```
INSERT INTO hosts VALUES ('<host>')
```

The function below summarizes our workflow:

```
1 def save_hostname(url: str, db_cursor: MysqlCursor):
2   sql_temp = "INSERT INTO hosts VALUES ('{host}')"
3   hostname = get_hostname(url)
4   sql_query = sql_temp.format(host=hostname)
5   db_cursor.execute(sql_query)
```

It uses the `format` method to instantiate the template `sql_temp` (Line 4), and the `execute` method from MySQL library to fire a query (Line 5). Is this implementation correct and safe?

2.1 Detecting SQL Injection

The previous implementation is indeed unsafe and suffers from SQL injection. To see this, let us consider the following malicious input, which is indeed not a valid URL (note that "--" starts a single line comment in SQL):

```
"https://localhost'); DROP TABLE users --/"
```

Feeding it into the `get_hostname` function, we obtain the following output:

```
"localhost'); DROP TABLE users --"
```

Instantiating the `sql_temp` variable in function `save_hostname` with it, we end up having a query involving an undesired **`DROP TABLE`** command:

```
INSERT INTO hosts VALUES ('localhost'); DROP TABLE users --')
```

Since all the above steps are well-typed, and that no runtime error has occurred so far, this malicious query will be executed, leading to the destruction of the `users` table in our database. Only then we will detect this security issue, but it is too late.

Although there are various industry practices to prevent SQL injection attacks, such as prepared statements, the problem of ill-formed inputs remains unsolved. What we really need is a fine-grained type system that is aware of the internal structures of strings. In our framework, to enable such a feature, the user needs to specify type annotations using our *language types*.

Solution 1: annotate `get_hostname`. We expect the input `url` to be a URL string, and the return value to be a hostname. Thus, we annotate this function using two built-in language types `URL` and `Host`[5]:

```
def get_hostname(url: URL) -> Host:
    ...
```

FLAT-PY now processes the above type annotations, and generates an *instrumented* version with two type assertions: one checks if the input `url` is a sentence of the URL grammar, and the other checks if the return value is a sentence of the hostname grammar, via parsing. We apply Earley parsing [4]: the worst time complexity is cubic, but linear for deterministic CFGs, which is the case for many practical grammars that are $LL(k)$ or $LR(k)$. If either check fails, a type error will be reported. Feeding the aforementioned malicious input to the instrumented code, we will encounter the following type error:

```
Type mismatch for argument 0 of method get_hostname
  expected type: URL
  actual value:  "https://localhost'); DROP TABLE users --/"
```

The error message says that the malicious input is ill-typed, as it does not match the URL grammar (`';'` is disallowed). This error aborts the execution and avoids the SQL injection attack.

Solution 2: annotate `sql_query`. An alternative approach is to annotate the local variable `sql_query` defined in `save_hostname` with a type for "safe" SQL queries:

```
sql_query: SafeSQL = sql_temp.format(host=hostname)
```

This customized language type `SafeSQL` can be introduced using a special type constructor `lang` provided by FLAT-PY:

```
SafeSQL = lang('SafeSQL', """
start: "INSERT INTO hosts VALUES " "(" Host ")";
""")
```

[5]As this function aims to handle simple URLs with ordinary hostnames (say, not IP addresses) and without queries and fragments, here we use a simplified URL syntax rather than the standard one defined in RFC 3986 [1]. For the latter, we provide another built-in type `RFC_URL`.

The second argument defines a grammar for this language type using an ANTLR-like [16] notation, and one is allowed to directly refer `Host` to the hostname grammar. As our intention here is to only insert hostnames into the database, the grammar only accepts this specific **`INSERT`** command, while rejecting the dangerous **`DROP TABLE`** command.

So far, we have finished all the required annotations. FLAT-PY now takes over to generate the instrumented code. Feeding the aforementioned malicious input to it, a type error will be triggered:

```
Type mismatch
  expect:    SafeSQL
  but found: "INSERT INTO hosts VALUES ('localhost'); DROP TABLE users --')"
```

Again, we avoid the SQL injection attack.

Remarks. To prevent SQL injection attacks, one may use *prepared statements*. In this approach, developers define the SQL code first and then pass in each parameter, possibly from outside inputs. The values of the parameters are regarded as data but not code, thus this approach guarantees that an attacker cannot change the intents of the queries. But this approach still neglects the formats of the parameters. We again need FLAT to ensure that the passed `host` parameter is indeed a valid hostname, otherwise the database would contain arbitrary strings in the `hosts` table.

2.2 Language-Based Fuzzing

Apart from the security issue, is our implementation functionally correct? That is to say, does it behave normally when feeding valid inputs? To answer this question, *language-based fuzzing* is a convenient approach: test inputs are automatically generated from the input grammars rather than manually specified. Traditionally, developers have to set up the entire fuzzing pipeline by themselves, including specifying the input grammars in the fuzzer's format.

In FLAT-PY, if the input types of a function are annotated, such a process is *automated* through a special `fuzz` annotation. To fuzz the `get_hostname` function, a single line of code is sufficient:

```
fuzz(get_hostname, 50)
```

The first argument of the `fuzz` function is the test target and the second is the number of random inputs needed. The input grammar for the `url` parameter is not needed here, but is automatically inferred from the type annotation of `get_hostname`. FLAT-PY will translate this type into an input grammar recognized by the backend fuzzer ISLa [21]. It will also set up the entire fuzzing process: invoke ISLa to generate 50 random test inputs, feed them to the target function `get_hostname()`, and record the testing results. We select ISLa as the backend fuzzer because it can also generate inputs satisfying additional logical constraints, which is useful to handle context-sensitivity (will see later).

From the 50 random test inputs, one empty-path URL `"http://W"` triggered a type error:

```
Type mismatch
  expect:    Host
  but found: ''
```

Note that this failing input `"http://W"` is indeed a valid URL. However, the output value—the empty string—is not a valid hostname. But for this particular input, the hostname should be `"W"` rather than empty, thus our implementation is incorrect.

Through further debugging, we realized this bug was caused by the code at Line 4: since the character `'/'` was not found in the rest part of the input `"W"`, the call to `find` returned `-1`, leading to an incorrect ending index of the hostname part. Consequently, the function outputs an empty string that violates the `Host` type requirement.

To fix this issue, one can, for example, use an if-statement for a case analysis on the emptiness of the path: if it is empty, then the hostname should be the remainder of the string; otherwise, the current approach applies. The fixed version is shown below (added lines are highlighted):

```
def get_hostname(url: URL) -> Host:
  """Extract the hostname part."""
  start = url.find('://') + 3
  end = url.find('/', start)
  if end == -1:      # fixed code
    end = len(url)   # fixed code
  host = url[start:end]
  return host
```

2.3 Test Oracles

There is one more property we wish to validate: the output must be the exact hostname extracted from the input `url`. Traditionally, to check this property, we have to develop a separate *test oracle* that first extracts the hostname from the input URL and then compares it against the actual output. This is indeed extra work for developers, in particular we need to build a URL parser to extract the hostname part.

This property describes a relationship between the input and the output string of the function `get_hostname`, thus in FLAT-PY, we express it as a *post-condition*, introduced by the `ensures` annotation attached to the function. To ease the process of substring extraction by syntax, FLAT-PY introduces *XPaths* that locate subtrees in the *derivation tree* of a string, and offers a `select(xpath, s)` function to extract the substring from the given string `s` that corresponds to the subtree located by the given `xpath`. The following shows the added annotation:

```
@ensures(lambda url, ret: ret == select(xpath(URL, "..host"), url))
def get_hostname(url: URL) -> Host:
  ...
```

In the `ensures` annotation, the post-condition is a predicate over all input arguments (`url`) and the return value (`ret`). Users may define this predicate as a lambda expression as shown above, it checks if the return value `ret` is equal to the hostname substring extracted from the input `url` (using the `select` function). The XPath is constructed via the `xpath(lang_type, path_lit)` function. The XPath literal `"..host"` refers to the unique node labeled with `host` (a nonterminal symbol of the URL grammar) in the derivation tree of `url`. This post-condition will be validated whenever the function returns: if violated, a type error will be reported. Using the `fuzz` annotation presented earlier in §2.2, no more type errors were detected.

3 FLAT-CORE: THE CORE LANGUAGE

We begin the technical presentation of our FLAT framework with an abstract core language called FLAT-CORE. It is a minimal language that integrates FLAT's language types into a tiny imperative programming language. It also offers annotations for users to attach pre- and post-conditions for methods. It can be extended to combine other advanced language features so that one can build FLAT instantiations for existing programming languages.

Program. A FLAT-CORE program p contains a set of definitions:

$$
\begin{array}{rcll}
p & ::= & \bar{d} & \text{(program)} \\
d & ::= & \textbf{def}\, f(x_1 : t_1, \ldots, x_k : t_k) : t = e & \text{(fun def)} \\
 & | & \textbf{method}\, m(x_1 : t_1, \ldots, x_k : t_k) : t\ \bar{\varphi}\ \{\bar{s}\} & \text{(method def)} \\
 & | & \textbf{lang}\, L = G & \text{(lang def)} \\
\varphi & ::= & \textbf{requires}\, e \mid \textbf{ensures}\, e & \text{(pre, post)}
\end{array}
$$

The meta notation $\bar{\cdot}$ refers to repeating $\cdot$ zero or multiple times. Like in Dafny [15] we distinguish between functions and methods: one uses functions to write specifications that should be pure, and methods to model the verification/testing targets that are usually impure. Thus, a function body is an expression e, while a method body is a block of statements $\{\bar{s}\}$. We require the user to annotate the types t_i for each parameter x_i and the return type t. A method can optionally have pre-/post-conditions specified using **requires** and **ensures** clauses. A pre-condition is a *predicate* (*i.e.,* a Boolean function) over the method's input arguments. A post-condition is a predicate over the method's input arguments and its return value. Both are typically lambda expressions. Furthermore, the special **lang**-definition specifies a *language type* L via a grammar G: any string value s has this type iff $s \in \mathcal{L}(G)$.

Grammars. We adopt an ANTLR-like [16] meta notation for expressing a grammar in the **lang**-definition. A grammar G consists of a set of production rules $\{r_1; r_2; \ldots; r_n\}$. Each production rule $A \longrightarrow \alpha$ consists of a nonterminal A and a clause α that is inductively defined as follows:

$$
\begin{array}{rcll}
\alpha & ::= & a \mid A \mid \alpha_1\alpha_2 \mid (\alpha_1 \mid \alpha_2) & \text{(standard)} \\
 & | & \alpha* \mid \alpha+ \mid \alpha? \mid \alpha\{k\} \mid \alpha\{k_1, k_2\} & \text{(repetition)} \\
 & | & [c_1\text{-}c_2] & \text{(char set)}
\end{array}
$$

Like ANTLR, we support terminals (a string literal a), nonterminals (A), concatenation (whitespace), alternatives ("|"), Kleene star ("$*$"), Kleene plus ("+") and optional ("?"). In addition, we support convenient notations inspired by Python's regex syntax:

- $\alpha\{k\}$ for repeating α exactly k ($k \geq 2$) times,
- $\alpha\{k_1, k_2\}$ for repeating α at least k_1 times and at most k_2 times ($k_1 < k_2$), and
- $[c_1\text{-}c_2]$ for the character set in between c_1 and c_2 (both inclusive).

Types. We adopt a two-layer type system. A *simple type* constitutes built-in types (for integers, Booleans, and strings) and function types:

$$
\begin{array}{rcll}
\tau & ::= & \textbf{Int} \mid \textbf{Bool} \mid \textbf{String} & \text{(built-in type)} \\
 & | & (\tau_1, \ldots, \tau_k) \longrightarrow \tau & \text{(function type)}
\end{array}
$$

The full type system supports language types introduced by **lang**-definitions and *refinement types*:

$$
\begin{array}{rcll}
t & ::= & \tau & \text{(simple type)} \\
 & | & L & \text{(language type)} \\
 & | & \{x : t \mid e\} & \text{(refinement type)}
\end{array}
$$

A refinement type $\{x : t \mid e\}$ consists of a *base type* t and a **Bool**-typed expression e as the *refinement*. We use the standard set-comprehension notation to indicate the inhabitants of a refinement type is a set of t-typed values that satisfy the condition e, where the value is bound to x in e. For example, a refinement type for a positive integer is written $\{n : \textbf{Int} \mid n > 0\}$. The bound variable x is ignored if not used in the refinement.

To fit the setting of testing, base types cannot contain function types: one usually needs first-order logic propositions (*i.e.,* with quantifiers $\forall$ and $\exists$) to express interesting properties over functions, but they cannot be evaluated to a Boolean value at runtime.

On the other hand, refinement types can be nested, say $\{\{n : \textbf{Int} \mid n > 0\} \mid n < 10\}$. One may put the two conditions "$n > 0$" and "$n > 10$" together to obtain an equivalent but compact form $\{n : \textbf{Int} \mid n > 0 \wedge n < 10\}$. We call such an operation *type normalization*, where the normalized type is a refinement type whose base type is always a simple type:

$$\mathit{norm}(\tau) = \{\tau \mid \textbf{true}\} \qquad \mathit{norm}(L) = \{s : \textbf{String} \mid s \in L\}$$

$$\frac{\mathit{norm}(t) = \{y : \tau \mid e_1\}}{\mathit{norm}(\{x : t \mid e\}) = \{x : \tau \mid e_1[x/y] \wedge e\}}$$

Expressions. We consider standard expressions with two special constructs:

$$\begin{array}{rcll} e & ::= & n \mid x & \text{(constants, variables)} \\ & \mid & e(e_1, \ldots, e_k) & \text{(function application)} \\ & \mid & (x_1, \ldots, x_k) \rightarrow e & \text{(lambda expressions)} \\ & \mid & \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 & \text{(if-then-else)} \\ & \mid & e \in L & \text{(language type test)} \\ & \mid & \mathit{select}(\pi, e) \mid \mathit{select\_all}(\pi, e) & \text{(XPath selection)} \end{array}$$

The special construct $e \in L$ tests (hence it has type **Bool**) if the **String**-typed expression e has the language type L. The two special functions $\mathit{select}(\pi, e)$ and $\mathit{select\_all}(\pi, e)$ are used for substring selection from the **String**-typed expression e by an *XPath* π (will be introduced soon). The former returns the *unique* substring located by π (if there are none or multiple, an error will be raised): it is the `select` function shown in §2.3. The latter returns *all* substrings located by π as a list.

Note that ordinary unary/binary expressions are internally desugared to function application, for example, $x > 0$ is internally just $gt(x, 0)$, where gt is a built-in function for greater than on integers. But for better readability, we prefer unary/binary expressions as syntactic sugars.

Library functions and XPaths. The core language comes with a library including: (1) standard unary/binary operations (for arithmetic, comparison, and Boolean computation) over integers and Booleans; (2) string operations defined in SMT-LIB's theory of Unicode strings[6]; and (3) XPath selection functions.

For ease of accessing substrings of a string $s \in L$, which are indeed the nodes in the derivation tree of s, we adopt an XPath-like syntax, using the nonterminals of the grammar as labels. An XPath is a sequence of *selectors*, each of which is one of the following:

- "$.A[k]$": to select the k-th ($k \geq 1$) direct child with label A;
- "$.A$": to select all direct children with label A;
- "$..A$": to select all descendants (both direct and indirect children) with label A.

For example, the XPath "$.A[1]..B.C$" refers to substrings of all the direct children with label C, that are of all descendants with label B, that are in the first child with label A of the root tree. To see this concretely, let us consider a string s with its derivation tree depicted in Fig. 1. The XPath selection $\mathit{select\_all}(.A[1]..B.C, s)$ returns a list of three substrings `'a'`, `'b'`, and `'c'`. Note that both `'b'` and `'c'` are selected because their common ancestor B_2 is indeed a descendant—though not a direct child—of A_1. The substring `'foo'` is not selected since its parent C_4 is not a direct child of B_3, as required by "$.C$". The substring `'bar'` is not selected since it is under A_2, the second (but not the first) child of the root S, as required by "$.A[1]$".

[6] https://smt-lib.org/theories-UnicodeStrings.shtml

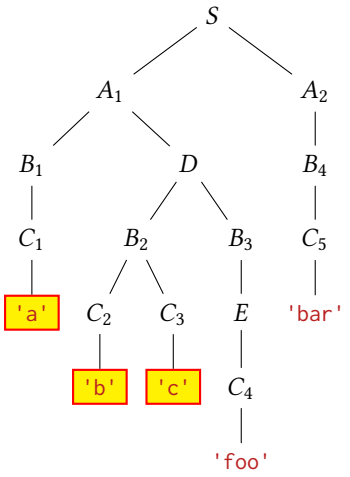

Fig. 1. The XPath "$.A[1]..B.C$" selects three (leaf) nodes from this derivation tree that are highlighted.

Statements. We consider commonly-used statements in a typical imperative language:

$$
\begin{array}{rcll}
s & ::= & \textbf{var}\, x : t; \mid x = e; & \text{(decl, assign)} \\
& \mid & \textbf{var}\, y = \textbf{call}\, m(e_1, \ldots, e_k); & \text{(method call)} \\
& \mid & \textbf{assert}\, e; & \text{(assertions)} \\
& \mid & \textbf{return}\, e; & \text{(return)} \\
& \mid & \textbf{if}\, e\ \{\bar{s}\}\ \textbf{else}\ \{\bar{s}\} \mid \textbf{while}\, e\ \{\bar{s}\} & \text{(control flow)}
\end{array}
$$

We allow a declare-and-assign statement "$\textbf{var}\, x = e;$" where the type of x is inferred from e. It is conceptually equivalent to "$\textbf{var}\, x : t; x = e;$" given that e has type t.

As the core language distinguishes methods from functions, here we introduce a method-call statement to invoke methods. Since statements can only appear in the method body, one can only call methods from methods, but not functions.

Example. Using the core language, we rewrite the Python function `get_hostname` defined in §2 as the method below:

```
method getname(url : URL) : Host
  ensures (url, ret) → ret == select(..host, url)
{
  var start : Int = indexof(url, "://", 0) + 3;
  var end : Int = indexof(url, "/", start);
  var host : String = substr(url, start, end − start);
  return host;
}
```

This method contains a post-condition that uses the select operator to extract the substring associated with the XPath "$..host$". The method body uses two string operations:

- $indexof(s, t, i)$ is the SMT-LIB function `str.indexof`: it returns the index of the first occurrence of the string t in string s, starting at the index i;
- $substr(s, i, n)$ is the SMT-LIB function `str.substr`: it evaluates to the longest substring of s of length at most n, starting at the index i.

4 RUNTIME TYPE CHECKING VIA INSTRUMENTATION

After showing the syntax and semantics of FLAT-CORE, this section presents how to "type-check" a FLAT-CORE program. Since this paper focuses on the dynamic approach, the "type checking" we refer to here is not the traditional static type checking done at compile time, but *instrumenting* necessary type assertions that will be executed *at runtime*, each checks if a concrete value has the specified type. This instrumentation process is done *automatically* via a preprocessing and analysis of the type definitions and the annotations in the original program. In the end, we execute the instrumented version: any violated assertion will immediately raise a type error at runtime.

4.1 Preparation

There are trivial type errors in FLAT-CORE programs that are related to simply types merely (not related to any logical constraint), such as assigning an integer variable with a string value, or feeding a Boolean to a function that expects a string. One can easily identify them via a standard static type checking algorithm (as done in Java), ignoring all the syntactic constraints required by language types (*i.e.,* regarding them as the **String** type) and all the semantic constraints occurred in refinement types and pre-/post-conditions. If the program is free of trivial errors, then we will perform program instrumentation to identify nontrivial errors related to the violation of any syntactic or semantic constraint.

4.2 Instrumentation

The overall workflow of program instrumentation is to traverse all methods in the original program and insert necessary assertions, including:

- a string having a language type must be a sentence of the corresponding grammar;
- a value having a refinement type must satisfy the refinement condition;
- when calling a method, its pre-conditions must hold for the input arguments;
- when a method returns, its return value must fulfill its post-conditions.

To formally present the instrumentation process, we build rules that transform an original program into an instrumented version. At the top level, the whole program transformation is done by replacing each original method

$$\textbf{method}\, m(x_1 : t_1, \ldots, x_k : t_k) : t\ \Phi\ \{S\},$$

with a new version

$$\textbf{method}\, m(x_1 : t_1, \ldots, x_k : t_k) : t\ \{S'\}.$$

The most important step is to transform the method body S into S' to contain all necessary assertions listed above (will be discussed soon). And this includes checks for pre- and post-conditions, hence the contract Φ is no longer needed in the new version.

We formally present the transformation of method bodies as a judgment of the form $\Gamma \mid m \vdash S \hookrightarrow S'$, stating that under the typing context Γ and the method m, a list of statements S (belonging to m's body) are transformed into S'. The typing context Γ maintains a list of bindings $x : t$, indicating that the local variable x has type t. Initially, the typing context records the types of m's input arguments:

$$\Gamma \triangleq \{x_i : t_i \mid 1 \leq i \leq k\}.$$

All the transformation rules process the statement list S sequentially. Most of them do not refer to the current method m so we elide it in the rule for simplicity (hence in form $\Gamma \vdash S \hookrightarrow S'$).

4.2.1 Declarations. No additional assertions are needed, but the typing context gets updated by appending (denoted by a comma) a new binding recording the local variable's type:

$$\text{Decl}\frac{\Gamma, x : t \vdash S \hookrightarrow S'}{\Gamma \vdash \textbf{var}\, x : t; S \hookrightarrow \textbf{var}\, x : t; S'}$$

4.2.2 Assignments. Additional assertions are needed to check if the assigned value e fulfills the constraint required by the type t of the target variable x. We extract this constraint via type normalization, say if $\mathit{norm}(t) = \{y : \tau \mid \varphi\}$, then any value of t must fulfill the predicate $\lambda y.\varphi$. We define a helper function $\mathit{extract}(t)$ to obtain this predicate. To avoid evaluating e twice, we decide, in the transformed body, to first perform the original assignment and then assert that x must fulfill that predicate:

$$\text{Assign}\frac{(x : t) \in \Gamma \quad \mathit{extract}(t) = p \quad \Gamma \vdash S \hookrightarrow S'}{\Gamma \vdash x = e; S \hookrightarrow x = e; \textbf{assert}\, p(x); S'}$$

The added assertion is indeed redundant if $p(x)$ is simply **true**, which happens if either t is a simple type or t is a refinement type with a trivial **true** condition. Our implementation did the optimization to drop such redundant assertions. But for the sake of generality, the transformation rules (including the subsequent) shown in the paper will keep them.

4.2.3 Method Calls. To handle method-related statements, we rely on several meta-properties for methods, which we precomputed for each method m defined in the program:

- $\mathit{sig}(m)$: the type signature of the method m, which consists of parameter names with types, and the return type;
- $\mathit{pre}(m)$: the pre-condition of the method m;
- $\mathit{post}(m)$: the post-condition of the method m.

In case multiple pre-/post-conditions are declared, $\mathit{pre}(m)$ and $\mathit{post}(m)$ give the conjunction. Note that $\mathit{pre}(m)$ is a predicate over m's input arguments, and $\mathit{post}(m)$ a predicate over m's input arguments and its return value.

At call sites, we need assertions: (1) to ensure each argument fulfills the constraint required by the corresponding parameter type of the callee, and (2) to ensure the pre-condition.

Our transformation rule is the following:

$$\text{Call}\frac{\begin{array}{c}\mathit{sig}(m) = (x_1 : t_1, \ldots, x_k : t_k) \to t \qquad \mathit{pre}(m) = p_m \\ \forall i : z_i \text{ fresh} \quad \forall i : \mathit{extract}(t_i) = p_i \quad \Gamma, y : t \vdash S \hookrightarrow S'\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{var}\, y = \textbf{call}\, m(e_1, \ldots, e_k); S \hookrightarrow \{\textbf{var}\, z_i = e_i; \textbf{assert}\, p_i(z_i);\}_{i=1}^{k} \\ \qquad\qquad \textbf{assert}\, p_m(z_1, \ldots, z_k); \textbf{var}\, y = \textbf{call}\, m(z_1, \ldots, z_k); S'\end{array}}$$

To avoid evaluating the actual arguments e_i more than once, we introduce fresh variables z_i to save their values. To achieve (1), we query the signature of the callee m, extract the required constraints by m's input types, and insert necessary assertions as done in rule Assign. To achieve (2), we query the pre-condition of m and insert an assertion to ensure the pre-condition holds for z_is. Finally, we call m with z_is and assign the return value to a newly declared variable y, appending the binding of $y : t$ to Γ, where t is the return type of m.

4.2.4 Return Statements. Dual to method calls, at returning points we need assertions: (1) to ensure the return value fulfills the constraint required by the return type, and (2) to ensure the post-condition.

Assuming the arguments are immutable, our transformation rule is the following:

$$\text{Return}\frac{\mathit{sig}(m) = (x_1 : t_1, \ldots, x_k : t_k) \to t \quad \mathit{post}(m) = p_m \quad z \text{ fresh} \quad \mathit{extract}(t) = p}{\Gamma \mid m \vdash \textbf{return}\, e; S \hookrightarrow \textbf{var}\, z = e; \textbf{assert}\, p(z); \textbf{assert}\, p_m(x_1, \ldots, x_k, z); \textbf{return}\, z;}$$

Again, to avoid evaluating the return value e more than once, we introduce a fresh variable z to hold its value. To achieve (1), we query the signature of the current method m, extract the required constraints by m's return type, and insert necessary assertions as done in rule Assign. To achieve (2), we query the post-condition of m and insert an assertion to ensure the post-condition holds for this method's input arguments x_is and the return value z. Because any statement after the return statement is unreachable, we do not process the rest statements S as other rules do.

If the arguments are mutable, we need to save their initial values into fresh variables and use them in the post-condition assertion, to match the usual interpretation of a post-condition that it relates the initial state (before executing the function body) with the return value.

4.2.5 Other Statements. No additional assertions are needed for other statements. We simply keep their structure and recursively process their nested bodies (if any):

$$\text{Assert}\ \frac{\Gamma \vdash S \hookrightarrow S'}{\Gamma \vdash \textbf{assert}\ e; S \hookrightarrow \textbf{assert}\ e; S'} \qquad \text{If}\ \frac{\Gamma \vdash S_1 \hookrightarrow S_1' \quad \Gamma \vdash S_2 \hookrightarrow S_2' \quad \Gamma \vdash S \hookrightarrow S'}{\Gamma \vdash \textbf{if}\ e\ \{S_1\}\ \textbf{else}\ \{S_2\}; S \hookrightarrow \textbf{if}\ e\ \{S_1'\}\ \textbf{else}\ \{S_2'\}; S'}$$

$$\text{While}\ \frac{\Gamma \vdash S_1 \hookrightarrow S_1' \quad \Gamma \vdash S \hookrightarrow S'}{\Gamma \vdash \textbf{while}\ e\ \{S_1\}; S \hookrightarrow \textbf{while}\ e\ \{S_1'\}; S'}$$

5 FLAT-PY: FLAT FOR PYTHON

In this section, we present FLAT-PY, a pratical testing framework for Python. It offers annotations (§5.1) for defining language types and refinement types, and specifying contracts for functions. It comes with built-in types for email address, URLs, JSON, etc. It provides a convenient `fuzz` function (§5.2) to enable random testing with the aid of language-based fuzzers. From these annotations, we adopt a similar approach (as in §4.2) to instrument necessary type assertions via Python AST transformation (§5.3). Finally, we discuss how our FLAT framework can be realized in other programming languages (**??**).

5.1 Annotations

We introduce two groups of annotations for Python:

- type constructors for defining new language types and refinement types, and
- method decorators for adding pre- and post-conditions.

Python's PEP 484[7] standard offers a flexible way to attach type annotations: type can be arbitrary Python expressions. To represent our language types and refinement types, which do not exist in Python's type system, we build two internal classes `LangType` and `RefinementType` for each of them and provide two type constructors as user APIs:

```
def lang(name: str, rules: str) -> LangType
def refine(base: type, predicate: Any) -> RefinementType
```

The language type constructor `lang` takes two parameters: a type `name` and a string representing the production rules, using the *extended Backus-Naur form* (EBNF) defined in FLAT-CORE (§3). For example, the following expression creates a language type for a simple integer arithmetic expression using only addition and subtraction:

```
lang('IntExp', """
start: (number op)* number;
number: [0-9]+;
op: "+" | "-";
""")
```

[7] https://peps.python.org/pep-0484

The refinement type constructor `refine` builds a refinement type $\{x : t \mid e\}$, taking two parameters:

- a Python `type` that represents the base type t, which is either a Python built-in type (`int`, `bool`, or `str`) mapping to the FLAT-CORE built-in types (**Int**, **Bool**, or **String**), or a language type constructed by `lang`;
- a Python function `predicate` that packs the bound variable x and its refinement e into a lambda expression `lambda x: e`.

For example,

```
refine(JSON, lambda s: len(s) < 10)
```

maps to the refinement type $\{s : \texttt{JSON} \mid \mathit{length}(s) < 10\}$.

One may use Python's type alias declaration to provide shorthands for language/refinement types and use them as the type annotations:

```
IntExp = lang('IntExp', """
start: (number op)* number;
number: [0-9]+;
op: "+" | "-";
""")
def calculate(exp: IntExp) -> int:
  ...
```

For pre- and post-conditions, we provide two method decorators that both take a predicate as inputs:

```
def requires(predicate: Any)  # for pre-condition
def ensures(predicate: Any)   # for post-condition
```

In `requires`, the `predicate` is over the input arguments of the method. In `ensures`, the `predicate` is over not only the input arguments but also the return value, so that one is able to express input-output relations as oracles. For example, the following contract reveals the functional correctness of converting a string with only digits into the corresponding integer value:

```
@requires(lambda s: s.isdigit())
@ensures(lambda s, n: int(s) == n)
def convert_digit(s: str) -> int:
  ...
```

To avoid repeating the input arguments in the lambda expressions, we also accept a string as the `predicate`, which is a textual representation of a valid Python expression in which the method input arguments are automatically bound and the return value is bound to "`return`". Using this style, the contract for `convert_digit` is rewritten as below:

```
@requires('s.isdigit()')
@ensures('int(s) == return')
def convert_digit(s: str) -> int:
  ...
```

Both styles will be used interchangeably in the remainder of the paper for better readability.

5.2 Type-Directed Test Generation

We offer a `fuzz` function that triggers the testing of a `target` function via `k` randomly-generated inputs:

```
def fuzz(target: Callable, k: int,
         using: Optional[dict[str, Generator]] = None)
```

Each input consists of one random value per parameter of the `target` function, and the producers can be specified in the optional `using` argument. We provide default producers for two kinds of parameters: (1) if a default value is specified, say v, then we synthesize a constant producer that always yields the value v; (2) if the parameter has a grammar-based refinement type $\{L \mid \varphi\}$, then we synthesize a producer that yields sentences of L that also fulfill φ. Otherwise, the user must provide one explicitly. If the `target` function contains preconditions, then only values that fulfill the conditions will be used as test cases.

Synthesizing a constant producer is trivial:

```
def constant_producer_for_v():
  while True:
    yield v
```

The interesting case is how to produce sentences of a CFG that also fulfill a semantic constraint. Recall that we use ISLa as our backend fuzzer. ISLa has its own specification language for expressing the semantic constraints, and *not all* Python Boolean expressions are convertible to formulae in this language. Our idea is to split the condition into two parts $\varphi = \varphi_1 \wedge \varphi_2$, where φ_1 is convertible to an ISLa formula φ_{ISLa}, and φ_2 is not. The producer works in a generate-and-filter fashion: it first calls the ISLa solver to obtain a string, which is in L and fulfills φ_1, but may not hold φ_2; this string is yielded only when φ_2 evaluates to `True`.

Technically, the condition split is realized by rewriting it into conjunctive normal form and converting as many of the conjuncts into ISLa formulae as possible, including:

- standard arithmetic, comparison, and Boolean operations;
- string operations that have equivalences in SMT string theory;
- substring selection via our library functions with XPaths, together with forall/exists check on the selected strings.

5.3 AST Transformation

Python offers a convenient `ast` module to parse, print, and manipulate AST nodes. We perform program transformation by subclassing `ast.NodeTransformer`. Except for the method call statement, all the statements of FLAT-CORE have equivalences in Python statements. For them, the transformation rules are conceptually the same as the rules presented in §4.2.

As Python does not distinguish methods from functions as FLAT-CORE does, the method call statement is simply a call expression in Python (of type `ast.Call`). The rule Call (in §4.2) does not apply. But realizing a simple fact that if the method m is called (from whatever method m'), the body of m will be entered, we can instead move all the necessary assertions from the caller (m') site to *callee* (m) site, by replacing the actual arguments (*i.e.,* the fresh variables z_is in rule Call) with the formal parameters of m. For example, we instrument the following assertions (highlighted) for `convert_digit`:

```
def convert_digit(s: str) -> int:
  assert s.isdigit()   # pre-condition
  ... # instrumented body
```

So far, if any instrumented assertion fails, the stack trace will report the locations in the instrumented code rather than the original code. This is inconvenient for the users to debug their code. To fix this issue, in our implementation, we instrument additional statements that inject the original locations and raise our customized errors rather than Python's `AssertionError`. Such customized errors will be caught, and a new stack trace will be built from the current stack trace, reporting the original locations extracted from the stack frames. For precondition violation errors, the new stack trace reports the error locations of the caller.

5.4 FLAT for Other Programming Languages

The concepts and ideas behind FLAT are *language-agnostic*. We choose Python as the first language to realize FLAT as a proof of concept in this paper because: (a) Python is flexible about type annotations thus easy to define and attach custom types, and (b) Python has a convenient `ast` module for code instrumention via AST transformation. It is possible to port FLAT to other host programming languages, but the required amount of engineering work depends on the features of that host language.

In a statically-typed language like Java, extending the type system of the host language with full-fledged refinement types is usually difficult: one has to modify the compiler to check those new types, and also to be careful about the interplay of the new types with the existing ones. A lightweight approach is to not regard the refinement types as types in that host language but as annotations (or other forms of syntactic metadata) attached to functions and variables. For example, in Java, one can leverage its annotation system to build special annotations for attaching type refinements, together with pre- and post-conditions. Once the annotations are ready, we parse them and perform program transformation, for example, through compiler plug-ins in Java.

Finally, library functions supported by FLAT-CORE, in particular the XPath-related ones, should be implemented in the host language as well. Apart from these, functions available from either the official or third-party libraries can be used to increase the expressiveness of the specifications.

6 CASE STUDIES

In this section, we apply FLAT-PY to real-world Python code fragments manipulating structured strings. We inspect their code to learn what the expected behaviors are, and attach necessary FLAT annotations to them. Based on the annotations, we execute our runtime checker, and if any type errors happens, we will study and understand the cause of failures. We conducted three case studies from §6.1 to §6.3, targeting the following research questions:

- RQ1: How much effort does the user put into specifying annotations?
- RQ2: How many type errors are detected using runtime checks?
- RQ3: What is the overhead of executing runtime checks?

Finally, we discuss the threats to validity in §6.4.

All experiments were conducted on a MacBook Pro with Apple(R) M1 Max chip, 32 GB memory, running Python version 3.11.9.

6.1 Format Validation

It is common in web/GUI applications to validate that user-provided data matches the required format before writing into databases. We studied three format validation functions taken from the PlatformIO Core project[8], a platform for embedded software development. The functions validate user names, passwords, and team names.

As an example, let us first consider the `validate_teamname` function:

```
1 def validate_teamname(value):
2   value = str(value).strip() if value else None
3   if not value or not re.match(
4     r"^[a-z\d](?:[a-z\d]|[\-_ ](?=[a-z\d])){0,19}$", value, flags=re.I
5   ):
6     raise BadParameter(
7       "Invalid team name format. "
```

[8] https://github.com/platformio/platformio-core/blob/591b377e4a4f7219b95531e447aec8d28fd41a79/platformio/account/validate.py

```
8       "Team name must only contain alphanumeric characters, "
9       "single hyphens, underscores, spaces. It can not "
10      "begin or end with a hyphen or an underscore and must"
11      " not be longer than 20 characters."
12    )
13  return value
```

The error message at Lines 7-11 describes the format requirements for a valid team name. We study if all strings that meet the required format can be accepted by this validation function, via random test generation. We will adopt and compare two approaches: one with FLAT-PY based on user annotations, and the other without FLAT-PY based on a grammar-based fuzzer like ISLa alone.

Testing with FLAT-PY. We start by specifying a language type for valid team names. From the natural language description in the error message, we first learn that a valid team name only contains alphanumeric characters, single hyphens (`'-'`), underscores (`'_'`), and whitespaces (`' '`), and is no longer than 20 characters. We define a language type `TeamNameFormat` to meet the above requirements:

```
TeamNameFormat = lang('TeamNameFormat', """
start: char{1,20};
char: [a-zA-Z0-9-_ ];
""")
```

We use this type to restrict the input values of the validator function:

```
def validate_teamname(value: TeamNameFormat):
  ...
```

We enable random testing of the validator function via the built-in `fuzz` annotation and we set 1,000 as the number of random test inputs:

```
fuzz(validate_teamname, 1000)
```

We have provided enough annotations for FLAT-PY to set up fuzz testing. FLAT-PY's internal program instrumentor takes care of synthesizing a proper input producer from the type signature of `validate_teamname`. But due to a missing constraint that is not considered by the language type `TeamNameFormat`, it is fine to see the validator crash with the `BadParameter` error.

To attach the missing constraint "it cannot begin or end with a hyphen or an underscore" into the `TeamNameFormat` type, we define a new refinement type `TeamName`:

```
TeamName = refine(TeamNameFormat,
  lambda s: not s.startswith('-') and not s.endswith('-') and \
            not s.startswith('_') and not s.endswith('_'))
```

This type now expresses the *exact* set of valid team names, so that once we update the type signature of the validator function, no more crashes are expected:

```
def validate_teamname(value: TeamName):
  ...
```

However, this time, FLAT-PY did find failing inputs such as `'R-_b'`. In this example, as the special characters (hyphen and underscore) are neither the first nor the last character, it indeed follows the desired format, but is rejected by the mysterious regex. We studied this regex carefully and realized that it encoded another constraint not mentioned in the error message: the character after the special character must be a normal alphanumeric character, as required by the lookahead assertion `'(?=[a-z\d])'` after `'[\-_ ]'`. Thus, `'R-_b'` is rejected by the regex because `'_'` cannot occur after `'-'`.

Following the same process, we studied the other two format validation functions in the PlatformIO Core project: `validate_username` and `validate_password`.

Table 1. Summary of testing format validation functions. LoA: lines of annotations. LoC: lines of code. T_P: time of producing random inputs (seconds). T_C: time of executing the code with instrumented checks (seconds). T_O: time of executing the original code without checks (seconds).

Function	LoA/LoC	Passed/Total	T_P (s)	T_C (s)	T_O (s)
`validate_username`	6/13	993/1000	32.857	64.514	0.001
`validate_password`	5/9	1000/1000	9.457	8.406	0.001
`validate_teamname`	8/13	993/1000	12.043	19.894	0.001

Results of FLAT-PY. Table 1 summarizes the results. For RQ1, one may measure the annotation burden by the ratio of LoA to LoC. On the three functions we studied, the ratio is less than 1.0. However, it may exceed 1.0 and vary depending on the code and what we wish to test. FLAT-PY tries to reduce user burden by offering convenient EBNF-style notations and accepting Python expressions directly as semantic constraints. And for this case, the required annotations are clear from the code comments. To this extent, we find attaching annotations is usually intuitive and pleasant.

For RQ2, FLAT-PY identified 14 failing inputs on two validation functions, from a total of 3,000 random inputs. These random inputs were produced in 54.357 s, with an average speed of 55.2 inputs per second. FLAT-PY is capable of catching logical bugs in real code, with the aid of language-based test generation.

For RQ3, from the last two columns (T_C and T_O), we see that the instrumented checks bring significant overhead ($T_C - T_O$), increasing the time from 0.01 s to a few seconds and even a minute. This is because the time cost of parsing is more than that of executing the original code. For functions that takes outside inputs, to prevent potential injection attacks, this overhead is *unavoidable*: either developers have to build validators (*e.g.,* via regex matching), or FLAT-PY checks inputs against the specified types (format validation for free). If developers choose to validate themselves (maybe because it is more efficient than CFG parsing), FLAT-PY offers a convenient way of testing: once the validators are well-tested, they can be used in production without FLAT-PY's checks. These extra CFG parsing or regex matching potentially make the code vunerable to Denial of Service (DoS) attacks. Developers should secure their systems using best industry practices such as monitoring and rejecting malicious inflows before they reach the parsing stage.

Testing with ISLa. One may realize the same functionality without our FLAT-PY tool, for example, to write testing code that first produces random inputs with the aid of a fuzzer like ISLa, and then sends them to the target function.

To use the ISLa solver for test generation, we need to provide a grammar with optional ISLa constraints. For better efficiency, we define this grammar in a different manner:

```
import string
ALPHA_NUM = reduce(lambda c1, c2: f'"{c1}" | "{c2}"',
                   string.ascii_letters + string.digits)
grammar = f"""
  <start> ::= <normal> | <normal><chars><normal>
  <chars> ::= "" | <char><chars>
  <normal> ::= {ALPHA_NUM} | " "
  <char> ::= <normal> | "-" | "_"
  """
constraint = 'forall <chars> s in start: str.len(s) <= 18'
```

As required by ISLa, the `grammar` is written in *Backus-Naur Form* (BNF), where terminals are double-quoted (*e.g.,* `"-"`) and nonterminals are angle-bracketed (*e.g.,* `<normal>`). The constraint "a term name cannot begin or end with a hyphen or an underscore" is reflected in the grammar: in the `<start>` rule, the first and the last character of the sentence are both `<normal>` characters—alphanumeric (defined as `ALPHA_NUM`) or `' '`. For any other `<char>` in the middle, special characters `'-'` and `'_'` are allowed. To express there are at most 18 such middle `<char>`s (so that the entire length is no longer than 20), we define a recursive rule `<chars>` and attach an additional ISLa `constraint` for bounding the length. From this `grammar` and `constraint` we instantiate an ISLa solver:

```
from isla.solver import ISLaSolver
solver = ISLaSolver(grammar, formula=constraint)
```

Using this `solver`, we write testing code that feeds 1,000 random inputs (obtained by the `solve` method) to the validator function, collecting all `failing_inputs`:

```
failing_inputs = []
for _ in range(1000):
  inp = str(solver.solve())
  try:
    validate_teamname(inp)
  except BadParameter:
    failing_inputs.append(inp)
```

Comparison. FLAT-PY saves developer's time and effort in the following aspects:

- EBNF-style grammars make it easier and more convenient to specify syntaxes, especially with repetitions and loops;
- FLAT-PY directly accepts Python expressions as constraints, thus no need to learn a new specification language such as the ISLa constraint language;
- No need to write code for setting up fuzzers and collecting testing results, as FLAT-PY will synthesize them automatically.

Key takeaways. From the above comparison, we see that the `fuzz` annotation in FLAT-PY reduces user burden in language-based test generation. Although one has to specify the language types, the process is usually intuitive as the required formats should be known, either formally or informally. Informal natural language descriptions can be converted into formal language types in a straightforward manner. To test format validators implemented by potentially complex regex patterns, FLAT-PY helps to uncover any inconsistency between the regex patterns and the natural language descriptions.

6.2 File Path Safety

Operating systems provide system APIs to process files, and file paths are usually represented as strings. Safety checking is needed to prohibit security issues, for instance, unexpectedly deleting a file located in a system folder.

The following function[9] rejects any access to an "unsafe" file that is outside the trusted `code/` folder, hence to avoid the situation where a system file is accidentally deleted:

```
1 def safepath(path: str) -> str:
2   base = os.path.abspath("code")
3   file = os.path.abspath(os.path.join(base, path))
4   if os.path.commonpath([base, file]) != base:
5     print(f"ERROR: Tried to access file '{file}' outside of code/ folder!")
```

[9] https://github.com/macazaga/gpt-autopilot/blob/b782a1c7005eadb3f93f4dbd1f56cdf499ed265b/helpers.py#L15

```
    sys.exit(1)
  return path
```

The expected behavior of `safepath` is that: if the relative `path` (to the base path `code/`) is indeed unsafe, then the function exits (and raises a `SystemExit` exception), otherwise it returns the `path` itself. FLAT-PY offers a special decorator `raise_if(exc_type, predicate)` to check that if the `predicate` holds for the input arguments, then the function must raise an exception of type `exc_type`. We use `raise_if` and `ensures` to attach post-conditions:

```
@raise_if(SystemExit, lambda path: not is_safe(path))
@ensures(lambda path, ret: ret == path)
def safepath(path: RelPath) -> RelPath:
  ...
```

And we wish to fuzz it as we did in §6.1:

```
fuzz(safepath, 1000)
```

The remaining problems are how to define a grammar for relative paths and a predicate that decides if a path is safe or not. Starting at the base folder `code/`, a relative path can be constructed via a sequence of one of the folder changing operations:

- entering a new folder: as the choice of the folder name is very unlikely to affect the behavior of the testing function, we simply pick a representative name `"foo"` here;
- going back to the parent folder via `".."`: this is where one can move outside the `code/` folder; or
- staying at the current folder via `"."`.

Thus, we introduce a new language type `RelPath` for relative paths as follows:

```
RelPath = lang('RelPath', """
start: (part "/")*;
part: "foo" | ".." | ".";
""")
```

We observe that once we move outside of the base path, we will not be able to return again, and thus the path is unsafe. To keep track of where we are, we use a counter `level` with initial value `0`, and:

- increase it by one if entering a new folder;
- decrease it by one if going back to the parent folder; or
- do nothing if staying at the current folder.

If `level < 0`, we immediately know the path is unsafe. We implement the procedure above as a safety decision function in Python:

```
def is_safe(path: RelPath) -> bool:
  level = 0
  for part in select_all(xpath(RelPath, '..part'), path):
    match part:
      case 'foo':
        level += 1
      case '..':
        level -= 1
        if level < 0:
          return False

  return True
```

Note that we use our built-in function `select_all` and an XPath `'..part'` to extract the parts from parse trees.

Results. For RQ1, we added (or edited) 19 lines of the annotations to the original 7 lines of code, where the ratio of LoA to LoC (= 2.7) exceeds 1.0. This is mostly due to a nontrivial semantic predicate `is_safe`, but itself is not hard to specify in Python once we have the observation. Sometimes, such user burden is inevitable if we need to thoroughly test the logical behavior of the program. For RQ2, FLAT-PY did not find any failing inputs. For RQ3, we measured the three time metrics as defined in Table 1: $T_P = 0.383$ s, $T_C = 0.340$ s, and $T_O = 0.006$ s. The overhead was significant but can be turned off in production after the code is well-tested.

Key takeaways. This case study shows the *high expressiveness* of FLAT-PY in refining types: the refinement can be any Python predicate, thus allowing complex logical properties to be expressed, which makes it possible to test nontrivial logical behaviors. In the future, we will use these annotations to eventually verify the functional correctness at compile time. Such complex properties are totally *optional* in FLAT-PY: developers may choose to what extent they wish to specify the types and properties, on partial arguments (and return values) for only a few functions, depending on their own goals. The annotations can be gradually refined and augmented over time.

6.3 File Path Sanitization

Another situation where unsafe file paths cause security issues is the existence of certain nonstandard, dangerous characters, which may cause damage to the file system or lead to shell command injection. We study a function[10] that sanitizes a potentially unsafe path into a safe version, free of certain dangerous characters (see docstring for details):

```
def safepath(path: str) -> str:
  """
  Returns safe version of path.
  Safe means, path is normalised with func:`normpath`, and all parts are
  sanitised like this:
  - cannot start with dash ``-``
  - cannot start with dot ``.``
  - cannot have control characters: 0x01..0x1F (0..31) and 0x7F (127)
  - cannot contain null byte 0x00
  - cannot start or end with whitespace
  - cannot contain '/', '\', ':' (slash, backslash, colon)
  - consecutive whitespaces are folded to one blank
  """
  ...  # 11 lines of code elided
```

Our goal is to test if `safepath` can always returns a sanitised path for any potentially dangerous path, so we need to define new language types `Path` and `SanitizedPath` for each of them, and refine the type signature of `safepath`:

```
def safepath(path: Path) -> SanitizedPath:
  ...
```

Again, we will apply the fuzzing technique for detecting bugs:

```
fuzz(safepath, 1000)
```

Input grammar. To define a grammar for `Path`, let us extend the grammar for `RelPath` defined in §6.2 with absolute paths and more general file names with ASCII characters:

[10] https://github.com/joeken/Parenchym/blob/63864cdaff76b9aa1b8dbe795eb537b5be5add3a/pym/lib.py#L374

```
Path = lang('Path', """
start: "/" | "/"? part ("/" part)*;
part: (%x0-2E | %x30-7F)*;
""")
```

As the parent folder `'..'` and the current folder `'.'` are both just ASCII strings, each part is simply an ASCII character sequence without the Unix separator `'/'` (of code point `0x2F`). Here we encode this using a special clause supported by FLAT-PY of the form "%xk_1-k_2", inspired by RFC 5234 (Augmented Backus–Naur form), to mean any character in between the hexadecimal code points k_1 and k_2 (both inclusive).

Output grammar. A `SanitizedPath` is still a `Path` but without certain characters (at certain places) as detailed in the docstring of `safepath`. We define `SanitizedPath` as a refinement type based on `Path`:

```
SanitizedPath = refine(Path, is_sanitized)
```

where the refinement predicate is defined as follows:

```
def is_sanitized(path: Path) -> bool:
  parts = select_all(xpath(Path, '..part'), path)
  return not any(
    [part.startswith('-') or part.startswith('.') or part.startswith(' ') or
     part.endswith(' ') or
     any([is_not_allowed(ch) for ch in part]) or
     (' ' * 2) in part
     for part in parts])
```

It checks if no part violates the constraints listed in the docstring:

- starts with dash (`'-'`), dot (`'.'`), or whitespace (`' '`);
- ends with whitespace;
- contains any control characters, null byte, slash (`'/'`), backslash (`'\\'`), and colon (`':'`), as defined in an auxiliary predicate `is_not_allowed` (see below);
- contains consecutive whitespaces, *i.e.,* at least two whitespaces (`' ' * 2`).

The auxiliary predicate `is_not_allowed` is defined as follows:

```
def is_not_allowed(char: str) -> bool:
  return ord(char) in range(0, 31 + 1) or ord(char) == 127 or
         char in {'/', '\\', ':'}
```

With the input and output grammars, we are ready to fuzz `safepath`. The fuzzer did not produce any failing input. However, we found this testing inadequate due to the potentially low coverage of "unusual" inputs containing the not-allowed characters, as the fuzzer would instead choose a normal character (like alphanumeric characters) with a higher probability.

Input grammar, revised. To increase the frequency of such unusual inputs, which are more likely to uncover bugs, we *refine* `Path` as a subtype `UnusualPath`:

```
UnusualPath = lang('UnusualPath', """
start: "/" | "/"? part ("/" part)*;
part: char*;
char: [abAB0] | [-.\\: ] | %x0-4;
""")
```

In contrast to `Path`, this `UnusualPath` type only considers a subset of `chars`, which are divided into three groups, each containing a small but the same number of (five) characters as representatives, and hence is chosen by the fuzzer with an equal probability:

Table 2. Failure test cases of `safepath`.

#	Input `path`	Length	Output	Constraint violated
1	`'/\x03:..a'`	6	`'..a'`	start with `'.'`
2	`'/\x00\x040//:-/\x00\\\x01/'`	13	`'0/-'`	start with `'-'`
3	`':-/\\B0/./\x03B\x02\x00\x04/'`	15	`'-/B0/B'`	start with `'-'`
4	`'//:.\x03\x01\x030//b'`	11	`'.0/b'`	start with `'.'`
5	`'/:\\-'`	4	`'-'`	start with `'-'`
6	`'/-.:-\\\\'`	7	`'-'`	start with `'-'`
7	`':-\x03:'`	4	`'-'`	start with `'-'`
8	`'\\-:'`	3	`'-'`	start with `'-'`

- normal alphanumeric characters {`'a'`, `'b'`, `'A'`, `'B'`, `'0'`},
- control characters from code point 0x0 to 0x5 (in Python syntax, they are represented as `'\x01'` to `'\x04'`), and
- special characters {`'-'`, `'.'`, `'\\'`, `':'`, `' '`} that are not allowed anywhere.

Without changing the type signature of `safepath`, *i.e.,* a function from `Path` to `SanitizedPath`, we tell the `fuzz` function to generate random paths using the above `UnusualPath` by specifying the optional argument `using` (where `lang_generator` convert a language type to a `Generator`):

```
fuzz(safepath, 1000, using={'path': lang_generator(UnusualPath)})
```

This time, we identified failing inputs (will be shown soon).

Results. For RQ1, we added (or edited) 23 lines of the annotations to the original 31 lines of code (*i.e.,* LoA/LoC = 0.75). Like in §6.1, the required annotations are clear from code comments so we performed a straightforward translation.

For RQ2, FLAT-PY found 8 failing inputs, with lengths 3-15. The information of these failure test cases is summarized in Table 2. For each test case, the table lists the input argument `path` (we fix the argument `sep` to be the Unix path separator `'/'`) with its length, the output, and the violated condition that explains why the output is not sanitized. The common failure reason is that the outputs start with a disallowed character (here, `'-'` or `'.'`).

For RQ3, we measured three time metrics as defined in Table 1: $T_P = 1.071$ s, $T_C = 4.755$ s, and $T_O = 0.001$ s. The overhead was significant but can be turned off in production after the code is well-tested.

Key takeaways. This case study shows how to refine language types for the purpose of testing in the `fuzz` annotation, without changing the type signature of the target function, which serves as a contract of the function. The effort we put in defining the `UnusualPath` type paid off as we increased the chance of hitting edge cases. Such effort is still necessary if we use a traditional language-based test generation tool. FLAT-PY makes it easier by offering built-in functions to construct customized input generators, which can be passed as options to the `fuzz` annotation.

6.4 Threats to Validity

External validity is the major threat in our case studies: there are only three cases and we are unsure if the same results can be generalized to other codebases, in particular the annotation burden may vary. When the tested code becomes more complex, one may put more effort on tuning the annotations (as in §6.3) to uncover more interesting bugs. However, one does not have to do so at the very beginning but refine the annotations *iteratively*: start with attaching grammars merely,

check for potential format violations, fix them, and refine the annotations to gradually consider more and more semantic correctness. In this way, the annotation effort will be under control.

7 RELATED WORK

Safe strings. It is a widespread problem that many strings have latent structure but type checkers of mainstream programming languages cannot reach it. The TypeScript community introduces *regex-validated strings* to guarantee that a string value must match a given regex. However, the memory of that structure is discarded after validation and thus cannot be further reused. To address this issue, Kelly et al. [13] present *SafeString*, a programming model where one specifies the latent structure as a grammar, and once a string is validated using a parser, its underlying structure is stored as an object in the memory so that rechecking is saved when updates come.

Both SafeString and our FLAT aim to improve the type safety of strings by exposing their latent structures. The major difference lies in the level of automation. SafeString defines an interface for safe strings. But it is the programmer's responsibility to realize a particular kind of safe string: to specify its structure, to build its parser, and to define a group of type-safe operations if needed. Therefore, it fits better when developing new software and refactoring existing codebases. On the other hand, FLAT is an extension to the type system of an existing programming language, offering users annotations to attach types and specifications. Apart from annotations, no additional code is mandatory in FLAT: parsers are automatically derived from grammars, and standard string operations apply to language types. Type checking is automated. To this end, FLAT applies to legacy codebases as well.

June [2] is a follow-up work of SafeString. It exposes the latent structure of strings to test generation tools via annotation-driven code transformation and equips them with built-in safe string annotations for delimited strings, file paths, emails, dates, etc, each of which is implemented as a safe string definition by subclassing the core `SafeString` class. Compared with SafeString, June is burdenless if using the built-in safe strings merely. When it comes to other kinds of safe strings, it inherits the inconvenience of SafeString too— the user has to subclass `SafeString` to realize the custom definition. In contrast, FLAT provides a principled type system where one defines new types in a declarative and simple way.

Refinement types. The concept of *refinement types* was first invented in ML's type systems [5]. The refinement type system allows more errors to be exposed at compile time, by refining user-defined algebraic data types in standard ML. For instance, a singleton list as a subtype of list.

A more modern style of refinement types is *Logically Qualified Data Types* [18], abbreviated to Liquid Types. Liquid types allow programmers to enjoy many of the benefits of dependent types with minor annotation effort, by encoding refinements as first-order logic predicates, with standard operations on common data types such as integers, bit vectors, and arrays. This idea is then introduced in many other programming languages, like Haskell [22], TypeScript [23], and Java [6]. However, none of them have focused on how to refine the string type to inject syntactic and semantic requirements of strings, as FLAT does.

Primitive obsession. There are conceptually different types like file paths and UUIDs but are represented into a single built-in type "string". Such a code smell is called "primitive obsession" [10, 20] and is considered a bad practice to use one primitive type for values with conceptually different types.

One possible solution is to define new types that are distinguishable by the type system. For instance, in an object-oriented language, one may use two separate class types `Path` and `UUID`, instead of the primitive string type, to distinguish file paths from UUIDs. This solution brings abstraction overhead (both for developers and runtime): one needs to wrap up a string into a

`Path` object and when processing, unwrap the string value. Scala 3 introduces *opaque types* [3] for defining aliases as conceptually different types from a primitive type. This distinction is made only at the type-system level for type safety but not at runtime for efficiency.

The above idea of distinguishing types using names is more like a *nominal* approach, while FLAT, which adopts a refinement type system, is a *structural* approach—the refinement type itself conveys structural information about the valid value set.

Language-based testing. Test generation has been studied for a long to improve test coverage and reduce human labor by automatically producing test inputs. To fit programs accepting structural text, such as compilers and interpreters, syntax-aware, grammar-based input generation has been studied. Godefroid et al. [7] present *grammar-based white-box fuzzing*, which increases the test coverage on Internet Explorer 7 JavaScript interpreter. CSmith [24] generates random C programs from a subset of the C grammar with hand-crafted generators, which detected hundreds of correctness bugs in mainstream C compilers like GCC and LLVM. LangFuzz [11] enables *black-box fuzzing* based on a CFG, and discovers security issues on the Mozilla JavaScript engine.

The above grammar-based test generators can produce syntactically valid inputs, but sometimes semantics validity is also important. Input algebras [9] allows one to express semantic requirements as a Boolean combination of patterns, serving as a specialization of the original CFG. A grammar transformer computes the specialized grammar, from which any standard grammar-based fuzzer can produce inputs conforming to the requirements. ISLa [21], on the other hand, natively supports semantic constraints expressed in its specification language, which is essentially a first-order logic over derivation trees of strings. These language-based, semantic-aware fuzzers enable type-directed testing in FLAT: types are translated into a CFG with logical constraints that depict the valid input space, from which random inputs are produced for testing.

Language mining. FLAT relies on user annotations to check correctness at runtime. The out-of-the-box FLAT built-in types and the rich EBNF notations ease the process of adding user annotations, but if they could be automatically inferred, our approach would become even more attractive. The key problem to realize this is language mining—automatically learning grammars from programs or examples.

Schröder and Cito [19] propose a static and automated grammar inference system for ad hoc parsers. Their intuition is that ad hoc parsers are usually language recognizers and internally maintain state machines for parsing. Through a collection of the constraints that the input must fulfill from the parser code, they construct a regular expression r as the inferred language accepted by the parser. Annotating this parser function with input type r, FLAT can test both the parser function itself and other functions referring to it.

There are also dynamic approaches that mine input grammars from a set of positive examples. Autogram [12] uses *dynamic tainting* to track the data flow from the input string to its fragments found during execution. A grammar that describes the hierarchical structure of the fragments is generalized from the structure of the call tree. However, if any fragment of the input is not stored in some variable, there will be no data flow to learn from. Mimid [8] tackles this problem by tracking all accesses of individual characters in the input, regardless of their usage. Arvada [14] targets a black-box setting where the program code is unavailable. Instead, it relies on an oracle to tell if an input is valid or not. It attempts to create the smallest CFG possible consistent with all examples, via tree bubbling and merging.

8 CONCLUSION AND FUTURE DIRECTIONS

String is a universal type used to encode all kinds of data that indeed have different latent structures. To distinguish them, we propose FLAT, regarding formal languages as types. Such language types

expose the underlying syntactic structure of strings. With semantic constraints as refinements, context-sensitivity is also taken into account. Such an expressive type system allows us to catch bugs as early as possible, and meanwhile, to automatically generate random inputs that conform to the user's assumptions, higher the chances of hitting more interesting and deeper bugs.

We implement FLAT in Python and present a practical testing framework FLAT-PY. We conducted case studies on real Python code fragments and detected logical bugs via random testing based on a reasonable amount of user annotations. The source code and experimental data are publicly available:

https://github.com/flatypes/flat-py.

Our future research directions of the FLAT framework will focus on extensions, empirical studies, and applications:

Static checking. While FLAT-PY's dynamic checking helps to catch bugs, the other side of the coin, static checking, is also important to guarantee the correctness of programs at compile time. We will explore the static type checking of formal language types in near future, leveraging techniques from program analysis and formal verification.

Empirical studies of annotation burden. We have seen that a reasonable amount of annotations required by FLAT gain benefits in testing, but it is still unclear how much annotation burden is needed in the wild. For this, we plan to conduct empirical studies to investigate several interesting questions: how often developers use standard/custom string formats, how complex their grammars are, and to which extent Large Language Models (LLMs) help in inferring the annotations?

XML schemas. There are DSLs that can express the intended latent structure of specific types of strings, such as XML schemas. In theory, every XML schema defines a sub-language of the XML language, hence can be expressed as a language type in FLAT. We plan to study how to build a converter from XML schemas (and other similar DSLs) to language types.

Porting to other languages. Apart from Python, there are many other programming languages like Java and JavaScript where string manipulation is common and thus FLAT can be useful. We are interested in realizing FLAT in these languages and understanding the technical and engineering challenges and gaps.

REFERENCES

[1] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. 2005. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986. https://doi.org/10.17487/RFC3986

[2] Dan Bruce, David Kelly, Héctor D. Menéndez, Earl T. Barr, and David Clark. 2023. June: A Type Testability Transformation for Improved ATG Performance. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 274–284. https://doi.org/10.1145/3597926.3598055

[3] Scala contributors. [n. d.]. Opaque Types. https://docs.scala-lang.org/scala3/book/types-opaque-types.html. Accessed: 2024-07-29.

[4] Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (1970), 94–102. https://doi.org/10.1145/362007.362035

[5] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

[6] Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1520–1532. https://doi.org/10.1109/ICSE48619.2023.00132

[7] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 206–215. https://doi.org/10.1145/1375581.1375607

[8] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 172–183. https://doi.org/10.1145/3368089.3409679

[9] Rahul Gopinath, Hamed Nemati, and Andreas Zeller. 2021. Input Algebras. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 699–710. https://doi.org/10.1109/ICSE43902.2021.00070

[10] Refactoring Guru. [n. d.]. Primitive Obsession. https://refactoring.guru/smells/primitive-obsession. Accessed: 2024-07-29.

[11] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[12] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 720–725. https://doi.org/10.1145/2970276.2970321

[13] David Kelly, Mark Marron, David Clark, and Earl T. Barr. 2019. SafeStrings: Representing Strings as Structured Data. *CoRR* abs/1904.11254 (2019). arXiv:1904.11254 http://arxiv.org/abs/1904.11254

[14] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2022. Learning Highly Recursive Input Grammars. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 456–467. https://doi.org/10.1109/ASE51524.2021.9678879

[15] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370.

[16] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[17] Pete Resnick. 2008. Internet Message Format. RFC 5322. https://doi.org/10.17487/RFC5322

[18] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

[19] Michael Schröder and Jürgen Cito. 2022. Grammars for Free: Toward Grammar Inference for Ad Hoc Parsers. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Pittsburgh, Pennsylvania) *(ICSE-NIER '22)*. Association for Computing Machinery, New York, NY, USA, 41–45. https://doi.org/10.1145/3510455.3512787

[20] Mark Seemann. [n. d.]. Primitive Obsession. https://blog.ploeh.dk/2011/05/25/DesignSmellPrimitiveObsession. Accessed: 2024-07-29.

[21] Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3540250.3549139

[22] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

[23] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 310–325. https://doi.org/10.1145/2908080.2908110

[24] 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, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[25] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book.