Grammar-Based String Refinement Types

Fengmin Zhu
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
fengmin.zhu@cispa.de

***Abstract*—Programmers use strings to represent variates of data that contain internal structure or syntax. However, existing mainstream programming languages do not provide users with means to further narrow down the set of valid values for a string. An invalid string input may cause runtime errors or even severe security vulnerabilities. To address that, this paper presents a Ph.D. research proposal on the type checking of grammar-based string refinement types, a kind of fine-grained types for specifying the set of valid string values via grammar. The string refinement type system uses subtyping to capture the inclusion relation between the languages of grammars. Based on that, we follow a well-known bidirectional type checking framework to combine the checking and inference of string refinement types into one. Evaluations on real-world codebases will be conducted to measure the practicality of this approach.**

***Index Terms*—Refinement types, context-free grammars, type checking, subtyping, constraint solving**

I. Introduction

Strings are everywhere in programming: they are used to encode various types of data: email addresses, URIs, telephone numbers, SQL queries, JavaScript code, etc. Feeding a function that parses an email address with a URI may cause undesired exceptions. Even worse, inputting a string that encodes malicious code can trigger security vulnerability if the program indeed lacks input validation (*e.g.,* SQL injection, JavaScript injection). One explanation for why strings cause errors and failures is that many mainstream programming languages do not offer developers the means to distinguish string values representing distinct data types and to specify which values are valid for that data type.

To provide more fine-grained control over the string type, inspired by the concept of *refinement types* [1], we propose to introduce grammar-based *string refinement types*. The big idea of refinement type is to provide additional constraints on coarse types (here the string type), *e.g.,* $\{s : \texttt{str} \mid len(s) = 5\}$ encodes the set of strings with length 5. Realizing that the data types mentioned above have structure and they follow certain syntax, it is natural to express the set of valid values using a *context-free grammar* (CFG): our string refinement type $\{s : \texttt{str} \mid s \in \mathcal{L}(G)\}$ encodes the set of string values that are in $\mathcal{L}(G)$—the *language* of G.

Making use of the syntax information conveyed in string refinement types, we will gain "more type-safety" from *static* type checking on string-manipulating programs, which helps to avoid potential bugs and security vulnerabilities at compile time. For example, we can check the type signature of the following Python function:

```
def extract_domain(email: Email) -> Domain:
  parts = email.split('@')
  return parts[1]
```

where `Email` and `Domain` are string refinement types encoding resp. an email address and a domain. Since RFC 5322 [2] defines that a valid email address must comprise a local part and a domain concatenated by an '`@`' symbol, accessing `parts[1]` will never raise **`IndexError`**. Type inference is also an important topic for string refinement types. Suppose

```
"mongodb://localhost:{}/{}".format(port, dbname)
```

follows the grammar

$$\texttt{<url>} \rightarrow \texttt{mongodb://localhost:<int>/<ident>},$$

then we could infer that `port` must follow the grammar of `<int>` (integers) and `dbname` the grammar of `<ident>` (identifiers). If `dbname` indeed contains injected queries that are likely to be malicious, the program does not type check, thus security vulnerability is avoided.

II. Problems, Challenges & Hypotheses

The central technical problems of this proposal are type *checking* and type *inference*. These two procedures, studied by the type system community for long, can be collapsed into one big procedure—a.k.a. *bidirectional type checking* [3]. A bidirectional type checker switches between *checking* and *inference mode*: when the expected type of an expression is known—*e.g.,* from user annotations, function types (if checking arguments of a function call), or previously inferred types—the checking mode is activated; otherwise, the inference mode is enabled to synthesize a type based on its sub-expressions' types.

Challenge 1: Subtyping: To check if an expression e has an expected type t, the trivial case is when e is a string literal s: the problem is reduced to tell if s can be parsed by the grammar of t. Otherwise, suppose the most precise (inferred) type for e is $\{s : \texttt{str} \mid s \in \mathcal{L}(G_1)\}$, and t is $\{s : \texttt{str} \mid s \in \mathcal{L}(G_2)\}$, we must check if the former is a *subtype* of the latter, *i.e.,* if G_1 is a subgrammar of G_2 (*i.e.,* $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$). Therefore, subtyping relation should be brought to string refinement types. This not only fits the object-oriented languages (like Python) better but also uncovers the fact that a string of G_1 is also a string of a larger grammar G_2.

It is well-known that in CFGs, equality checking (*i.e.,* whether the two languages accept the same set of sentences) is undecidable [4]. This further indicates there is no universal

way to test the subtyping relation—if a CFG is a subgrammar of another, which makes it a technical challenge. However, we could instead propose ad hoc algorithms that lead to sound but incomplete type checking, and preferably the algorithms should cover plenty of real scenarios.

Challenge 2: Grammar Inference: For type inference, the key challenge is to compute the resulting types for various string manipulations including concatenation, substring extraction and replacement. Concatenation is a trivial one: the resulting grammar is simply the concatenation of the two input grammars. But for substring extraction `s[i:j]`, the inference becomes nontrivial as the substring in the range `[i:j]` may not correspond to a fixed nonterminal symbol in the grammar of `s`. In short, type inference requires computations on grammars and the resulting grammar should be as precise as possible.

Challenge 3: Logical Constraints Solving: Sometimes, it is not enough to use a CFG alone; adding logical constraints makes it more expressive and powerful. For instance, the refinement type $\{s : \texttt{str} \mid s \in \mathcal{L}(G) \wedge len(s) \leq 10\}$ requires not only the string be in $\mathcal{L}(G)$, but also its length be no longer than 10. Taking such logical constraints into account, the subtype checker should be able to prove/disprove if one constraint logically implies another. To discharge such proof automatically, constraint solving is available: the technical problem is how to encode constraints as the solver's inputs.

Applications: String refinement types bring opportunities for dynamic approaches too. With type annotations or inferred types, we can check at runtime if a concrete string follows the required syntax; so that we can discover incompatibility issues before they cause major damage at a system level. The type annotations can be further leveraged for data-flow analysis, system testing, grammar-based fuzzing, and so on.

Research Hypotheses & Limitations: To fit the bidirectional type checking framework better, we assume the program code is attached with sufficient refinement type annotations, particularly, the types of function arguments must be known. For recursive functions, the return type must also be given (it is inferrable on non-recursive functions). To leverage off-the-shelf SMT solvers (*e.g.,* Z3 [5]), the additional constraints presented in refinement types are assumed to be in decidable first-order logic. Since subtyping relation testing is in general undecidable for CFGs, the type checker cannot guarantee completeness. But soundness is possible to achieve by rejecting a subtyping relation if we don't know how to prove/disprove it.

III. Related Work

The most related work to my topic is *grammar inference* of ad hoc parsers [6]. They perform a static analysis on the source code of parsers and infer a grammar (typically a regular expression) expressing the parser-accepted language. Their work does not bring grammars into type checking, but their algorithm will help to infer string refinement types for programs that do ad hoc parsing. Another approach for learning grammars is *grammar synthesis*. Parsify [7] synthesizes CFGs hierarchically and interactively from human-specified inputs. Arvada [8] is an oracle-based algorithm learning highly recursive CFGs from a set of positive examples.

Grammar-based fuzzing [9] is a testing technique where valid inputs are randomly generated from a grammar. It is also possible to attach additional constraints to grammars: from Boolean combinations of patterns [10] to first-order logic [11]. This inspires us to also take logical constraints on strings into account in our refinement type system.

The concept of *refinement types* was first invented in ML type systems [1, 12] and later introduced in Haskell [13] and TypeScript [14]. In those works, refinements apply language-wide to almost every type. Particularly refining the string type with grammars is yet still a novel and unsolved problem.

IV. Expected Contributions

The expected contributions of this research are as follows:

- an annotation system for developers to attach string refinement types into source code in mainstream languages such as Python;
- a sound and practical subtype solver for string refinement types (challenge 1);
- a type checker for string refinement types (challenge 2) integrated into an existing language's compiler such as Python's static typing framework `mypy`;
- support for first-order logical constraints in string refinement types (challenge 3);
- evaluations on real-world codebases for static analysis and bug detection.

In addition to the technical difficulties mentioned in section II, how to integrate string refinement types into existing compilers, and meanwhile, to maintain all the existing typing features, is unobvious. For example, `mypy` relies on Python's type hints (PEP 484). To fit this framework better, our string refinement types should be expressed in type hints as well. Since `mypy` has no refinement types, we must extend its type system to include this new string refinement type, recognize it from type hints, perform type checking on it, and consider its subtyping relations with other Python types including the built-in `str` type and string literal types.

V. Plan of Evaluation

The evaluation process will consist of two phases. In the first phase, the functional correctness of the implemented type checker will be examined using: (1) unit tests for validating the behaviors of individual components (*e.g.,* subtyping solver, type synthesizer); and (2) end-to-end tests including both (hand-crafted) well-typed and ill-typed programs for validating the behaviors of the entire type checker—it should report type errors for ill-typed programs and accept well-typed programs.

In the second phase, large programs extracted from real-world codebases will be used to measure how the proposed approach helps to detect potential bugs and vulnerabilities. Several metrics will be studied to assess the overall performance, such as the effort of annotating string refinement types, the cost of performing static type checking, the number of issues found and the proportion of false positives.

REFERENCES

[1] T. Freeman and F. Pfenning, "Refinement types for ml," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, (New York, NY, USA), p. 268–277, Association for Computing Machinery, 1991.

[2] P. Resnick, "Internet Message Format." RFC 5322, Oct. 2008.

[3] J. Dunfield and N. Krishnaswami, "Bidirectional typing," *ACM Comput. Surv.*, vol. 54, may 2021.

[4] H. J. Hoogeboom, "Undecidable problems for context-free grammars," *Preprint https://liacs. leidenuniv. nl/~ hoogeboomhj/second/codingcomputations. pdf*, 2015.

[5] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.

[6] M. Schröder and J. Cito, "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*, ICSE-NIER '22, (New York, NY, USA), p. 41–45, Association for Computing Machinery, 2022.

[7] A. Leung, J. Sarracino, and S. Lerner, "Interactive parser synthesis by example," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, (New York, NY, USA), pp. 565–574, ACM, 2015.

[8] N. Kulkarni, C. Lemieux, and K. Sen, "Learning highly recursive input grammars," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, p. 456–467, IEEE Press, 2022.

[9] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," 2019.

[10] R. Gopinath, H. Nemati, and A. Zeller, "Input algebras," in *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, p. 699–710, IEEE Press, 2021.

[11] D. Steinhöfel and A. Zeller, "Input invariants," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, (New York, NY, USA), p. 583–594, Association for Computing Machinery, 2022.

[12] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), p. 159–169, Association for Computing Machinery, 2008.

[13] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for haskell," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, (New York, NY, USA), p. 269–282, Association for Computing Machinery, 2014.

[14] P. Vekris, B. Cosman, and R. Jhala, "Refinement types for typescript," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), p. 310–325, Association for Computing Machinery, 2016.