Fuzzy-Testing Subtype Relation

Extended Abstract

Artem Pelenitsyn
Northeastern University
pelenitsyn.a@northeastern.edu

Abstract

We propose a hybrid random-testing approach involving well-developed QuickCheck-style generators and the idea of fuzzing. It allowed for efficient bug detection in an implementation of a particular subtype relation found in the Julia programming language. We believe the approach is applicable to other binary relations on a set of terms described by a typical context-free grammar with binders.

Keywords quickcheck, testing, fuzzing, subtyping, enumeration, Julia programming language

1 Introduction

Fuzzy testing is a branch of randomized testing suggesting generation of large volume of nonsensical and even ill-formed inputs to the tested software. Fuzzy testing is gaining popularity among various programming research communities, especially in the area of compiler construction [18].

Unlike fuzzying, the renown QuickCheck library and its derivatives generate random data using well-defined specifications given by algebraic data types and predicates on them. QuickCheck-style techniques have been utilized in a wide range of applications from theorem proving [7, 14] to compiler testing [4, 13].

We seek the middle ground between fuzzing and generative, QuickCheck-style techniques. We find a compelling case study for this purpose in the Julia programming language [2]. In particular, its subtype relation, despite being based on the clean intuition of semantic subtyping, is hard to capture formally [17]. Although the relation becomes handier to formalize and QuickCheck in a proof assistant when being restricted to a well-behaved subset [1, 3], the whole relation does not enjoy the same opportunity.

We develop a combined approach to randomized testing of the Julia subtype relation. In particular, to generate pairs of Julia types that are likely in the subtype relation we split the grammar of types, following 20/80 rule, into two parts: the larger nearly trivial part amenable to a QuickCheck style generation, and a smaller more involved part processed by a fuzzing-inspired mutation procedure that accounts for well-formedness requirements and the intended semantics of subtyping. Generated pairs are fed to an implementation of the relation under test and the Julia VM to find discrepancies.

venue, date, place
2019.

2 Background: Subtype Relation of Julia

We approximate Julia's type grammar as follows.

$$\tau ::= \top \mid \bot \mid \mathit{tname} \mid \mathtt{X} \mid \exists \mathtt{X}.\tau \mid \mathit{tname}\{\tau\} \mid \tau_1 \times \tau_2 \mid \tau_1 \cup \tau_2$$

Here, *tname*s arise from the *class table* — a set of user-declared types (like `Int`, `Array`, etc.), which form a nominal subtyping hierarchy.

The full subtype relation in Julia adds to the nominal hierarchy a set of intuitive rules inspired by semantic subtyping [8]. In the semantic approach, types are treated as sets of values, and subtyping reduces to set inclusion; e.g. a union of two types is greater than either of components, pairs are covariant, type applications are invariant, etc.

Our definition of subtyping is declarative. In fact, we can only call it a specification of subtyping, and not a definition, because of certain complications in the rules for existentials. Decidable algorithmic subtyping for a Julia-like type system remains an open problem.

3 Fuzzing

To generate pairs of Julia types that are likely in the subtype relation, we have two mathematical objects: the type grammar and the (declarative specification of the) subtype relation. The former already suits the established techniques to generate terms of a grammar. For the latter, however, there is a severe restriction to apply the same: generation of data subject to a certain predicate is only feasible if we have a "good" definition of the predicate (e.g. inductive, strictly-positive). This is not the case for Julia subtyping: the definition from Section 2 is too abstract to be fed to a constrained data generation engine.

As we do not have a suitable formulation of the subtype relation, we concentrate on generation of type terms. We handle the likeness of subtyping between generated terms by adapting known generation techniques. In particular, our main insight is that the type grammar can be factored into two parts following the 20/80 rules: 80% of the grammar are straightforward, and the remaining 20% require special care for two reasons. First, efficiency of the generation process: while context-free sentences are simple, context-sensitive fragments (e.g. binders) slow down generation significantly. Second, those 20% are exactly the part we use for fuzzing to get several related types.

Julia's type grammar looks similar to what is often found in programming languages theory: it has recursive constructors (pairs, unions and type applications), constants (type names, top and bottom) and other leaf nodes (type variables); finally, it has binders (existentials). We propose to use simple enumerative generation to obtain a skeleton of a type, called *quasitype*, using only recursive constructors. After that, we augment the skeleton with information about leaf nodes and binder positions such that the binding structures are well-scoped and all names come from a simplified class table. Additionally, we order the output of the second phase in a way suggesting that less general (w.r.t. to subtyping) instances come earlier in the list than more general ones.

The implementation proceeds as follows. First, we define the grammar of quasitypes:

$$\textbf{data}\ QT = App\ QT \mid Pair\ QT\ QT \mid Union\ QT\ QT \mid Hole$$

Note the absence of the existential constructor and *tname* in the application constructor `App`; all the leaf nodes are factored out into a single node `Hole`. Then, values of `QT` are generated by the FEAT library [6] finishing the first stage of the process. On the second, mutation stage, we need to turn a quasitype `QT` into a list of well-formed types `T` given by the grammar from Section 2. For this, all the holes of the quasitype get replaced with either a nominal type or a type variable and binders are added. The most challenging question here is what information we need to keep track of to achieve enumeration of all non-isomorphic hole fillings with type variables. The answer is encoded in the type of the mutator algorithm:

$$mutator :: QT \rightarrow StateT\ Nat\ [\,]\ T$$

The `Nat` part keeps track of the number of used type variables. Once the mutator type is coined, the implementation turns into straightforward structural recursion over `QT`.

4 Evaluation

To perform experiments with the fuzzer [15], its output is piped into a Julia script that searches for discrepancies between two implementations of the subtyping algorithm from [17] and the Julia VM. Most of the cases found were due to the bugs in [17] (while it was under development). But we also filed three bug reports to the Julia's tracker: two of them were fixed (#26180, #26654), and one remains open (#27101).

At least one more project independently employed the fuzzer as a regression testing tool while formalizing a part of Julia subtyping algorithm in Coq [3]. The authors reported successful re-targeting of the tool to their needs.

5 Threats To Validity

The presented technique suffers from several problems usual to randomized testing. First, deduplication of found bugs is hard: sometimes hundreds of failing test cases point to the same bug. Second, shrinking (minimization) of a bug-exposing test case is not implemented; given that a failed test case consists of a pair of terms instead of one term, it is not even clear how to approach shrinking algorithmically. Third, the distribution of the generated test cases is not studied: previous work [12] suggests this task may turn into a research problem on its own. Finally, some aspects of our implementation use certain algebraic properties of subtyping in Julia that might not hold in other system — this somewhat restricts the applicability of the algorithm in a different setting. Nevertheless, the general idea should still be valid.

6 Related Work and Alternative Designs

The seminal paper on QuickCheck [5] inspired numerous works on generative testing. The most relevant to us is probably the work on SmallCheck [16]: it conjectures that exhaustive enumeration of small inputs is sometimes more effective that sampling large random input. For our use case, the Julia subtype relation, this is even inevitable because Julia VM often times crashes on large inputs.

Generation of constrained data has been studied since, at least, QuickCheck. One of its limiting properties is performance. Certain speed-ups can be achieved by lazy evaluation of predicates [16]. However, in our experiments, an attempt to generate well-formed terms using this kind of techniques did not perform well. The idea of to-be-augmented quasitypes worked more efficiently.

Works on testing *inductive relations* suggest that a generation engine can leverage the full access to the predicate definition instead of viewing it as a black box. Such access is provided through either a DSL for predicate definitions [10] or deep reflective abilities of the language, e.g. Coq [11]. Our relation is too complicated for either of these approaches.

An approach to testing a binary relation was thought of in the very QuickCheck paper for testing a unification algorithm [5, 5.1.4]. Our technique is inspired by it. Moreover, a recent work proposed uniting QuickChecking and fuzzing too but applied the idea to test implementations of binary file formats [9]. This task is closer to traditional use cases for fuzzing, that is why it was handled by off-the-shelf fuzzers.

7 Conclusion and Future Work

We have built a fuzzy-testing technique for a complex not-obviously-inductive binary relation of subtyping as found in the Julia language. We believe our ideas can be applied in similar settings; e.g. in functional languages, complicated subtype relations arise from higher-rank or impredicative features of the type system. Also, gradually typed languages with relations of consistency (usually, a simpler, mostly syntactic) and consistent subtyping (more involved, closer to what Julia has in its subtype relation) seem like a good target for the presented technique.

References

[1] Julia Belyakova. 2019. Decidable Tag-based Semantic Subtyping for Nominal Types, Pairs, and Unions. In *21st Workshop on Formal Techniques for Java-like Programs (FTfJP 2019)*. (to appear).

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017). https://doi.org/10.1137/141000671

[3] Benjamin Chung, Francesco Zappa Nardelli, and Jan Vitek. 2019. On Julia's efficient algorithm for subtyping union types and covariant tuples. In *33nd European Conference on Object-Oriented Programming (ECOOP 2019)*. (to appear).

[4] Koen Claessen, Jonas Duregård, and Michał Pałka. 2015. Generating constrained random data with uniform distribution. *Journal of Functional Programming* 25 (2015), e8. https://doi.org/10.1017/S0956796815000143

[5] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. 268–279. https://doi.org/10.1145/351240.351266

[6] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2364506.2364515

[7] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. 2003. Combining Testing and Proving in Dependent Type Theory. In *Theorem Proving in Higher Order Logics*, David Basin and Burkhart Wolff (Eds.). Springer Berlin Heidelberg, 188–203.

[8] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* 55, 4 (2008). https://doi.org/10.1145/1391289.1391293

[9] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: An Automatic Random Fuzzer for Common File Formats. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 13–20. https://doi.org/10.1145/2976002.2976017

[10] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 114–129. https://doi.org/10.1145/3009837.3009868

[11] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating Good Generators for Inductive Relations. *Proc. ACM Program. Lang.* 2, POPL, Article 45 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158133

[12] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching Processes for QuickCheck Generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3242744.3242747

[13] Michał Pałka. 2012. *Testing an Optimising Compiler by Generating Random Lambda Terms*. Licentiate Thesis. Chalmers University of Technology, Gothenburg, Sweden.

[14] Z. Paraskevopoulou, C. Hriţçu, M. Dénès, L. Lampropoulos, and B.C. Pierce. 2015. Foundational property-based testing. *Lecture Notes in Computer Science* 9236 (2015), 325–343. https://doi.org/10.1007/978-3-319-22102-1_22

[15] Artem Pelenitsyn. 2018. subtype-fuzzer. https://github.com/prl-prg/subtype-fuzzer. (2018).

[16] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1411286.1411292

[17] Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 113:1–113:27. https://doi.org/10.1145/3276483

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