Fuzzy-Testing A Subtyping Relation

Artem Pelenitsyn
Northeastern University
pelenitsyn.a@northeastern.edu

Abstract

Modern object-oriented languages make use of complex subtype relations. Consider the Julia programming language: it has user-defined invariant parametric types, existential types, union and covariant product types, as well as some ad hoc rules facilitating the use of the core mechanism of the language — multiple dispatch. Julia's subtype relation, which has recieved no rigorous description so far, is implemented in 2KLOC of highly-optimized C code touched by only a handful of core developers, and perceived by the community as a black box; examples sending the algorithm into a loop or crushing the VM keep popping up on the issue tracker. To tame the complexity, we have built a formal definition of the relation, accompanied by an implementation. But how can we be sure that our implementation indeed corresponds to the cryptic C one? To answer that, we propose a fuzzy-testing approach. It allows for efficient bug detection in an implementation of a subtype relation in the presence of an oracle. We believe that the approach can be used to fuzzy-test an implementation of any subtype relation and, more generally, any binary (even n-ary) relation.

To implement our approach, we model the type grammar of interest in Haskell and make use of some well-known functional enumeration techniques. We present a design of a special-purpose monad stack aimed to carefully guide the enumeration engine and avoid state-space blowup.

Keywords quickcheck, testing, fuzzing, subtyping, enumeration

1 Introduction

Fuzzy testing is not new. It grows in popularity among various communities. Maybe, not so much in the programming languages community. A particular area of PL especially successful in fuzzing is compiler construction (see [19] and accompanying references). The task of enumerating valid compiler inputs is well-studied and hard, because most mainstream programming languages lack formal description (the purely syntactic criterion is usually too rough).

On the other side of the PL spectrum is the zoo of ever sophisticating type theories. Nowadays they usually possess a rich metatheory, often a mechanized one. It seems that there is not much room (and sense) for fuzzing there. Perhaps surprisingly, this starts to change because of a number of academic efforts achieving popularity in industry, especially, GHC and Coq compilers:

- The Glasgow Haskell Compiler being... well, a compiler, requires comprehensive testing techniques. A successful fuzzing attempt was made to test GHC's strictness analyzer: first, by writing a custom generator manually [13] and then, when realized the complexity of the task, by building a systematic approach [5].
- The call for QuickCheck-like functionality in the field of theorem proving was raised and explained only recently [14]. Much like in Haskell community, it moved towards the development of methods for generating the constrained random data [10, 11].

Generally, both lines of work considers random (or exhaustive) generation of terms from a given algebra. At the same time, the terms should satisfy certain (inductive) predicates. The examples are: complete binary trees, well-typed lambda-terms of simply-typed lambda calculus, etc.

What if, to test a certain relation, we have to generate a pair of terms, not a single term? Also, the relation might not be inductive or, on the extreme, not even well-founded due to, e.g., its dark origins lying in the industrial suburbs of the compiler construction area. We hit one such relation in the wild and found it surprisingly painful to apply any of the existing approaches directly.

Julia is a programming language for technical computing meant to be both high-level and performant, providing object-oriented paradigm with multiple dispatch as the main code structuring tools [2]. While being a dynamic language with no static type checking whatsoever, Julia has surprisingly rich grammar of types utilized for the annotations of method parameters. Multiple dispatch mechanism decides dynamically which method from the overloading set to call, matching the run-time type-tags of the arguments with the method signatures. The matching process falls back to the Julia subtype relation.

The implementation of subtyping in Julia is perceived as one of the most cryptic parts of the compiler. Also, the relation possess no formal definition to date; leave alone formal semantics of the language or, at least, multiple dispatch. Any PL-theorist who self-inflicts him/herself with the task of building such a formal definition has to suffer from the absence of matters to prove any kind of soundness theorem. Therefore, (s)he has to resort to testing.

Contributions We propose an approach for testing a relatively complex subtype relation in the presence of an oracle, taking the Julia subtype relation as a particular instance (we

venue, date, place
2018.

τ	::=		Types
	$\mid$	$\top$	Top
	$\mid$	$\bot$	Bottom
	$\mid$	*tname*	Type Name
	$\mid$	X	Type Variable
	$\mid$	$\exists \mathsf{X}^{\tau_u}_{\tau_l}.\tau$	Bounded Existential[1]
	$\mid$	$tname\{\tau\}$	Type Application
	$\mid$	$\tau_1 \times \tau_2$	Pair
	$\mid$	$\tau_1 \cup \tau_2$	Union

Figure 1. Julia's type grammar (simplified)

describe the relation in Section 2). The approach is based on the generation of pairs of *similar* random type-terms. The similarity is handled by the idea of *quasitypes* (Section 3.1) — a simplified grammar of types, which can be handled by conventional methods of enumeration of algebraic types and then mutated into various similar types — candidates for subtyping pairs (Section 3.2). We evaluate our approach on the custom implementation of the Julia subtype relation using Julia VM as an oracle with the outcome of finding bugs in both (Section 4).

2 The Subtype Relation

We start with the description of a particular subtype relation inspired by the Julia programming language. This lets us reason about the fuzzing in detail later on.

2.1 Types

Figure 1 shows our approximation of the Julia's type grammar. In fact, Julia supplies *n*-ary versions of the three latter type constructors. Also, it allows for values to appear in the types under certain circumstances (e.g. `Array{Int,2}` stands for the type of integer matrices). Left-hand side of the type applications is also somewhat more subtle but can be sorted to exactly *tname*. We omit these features as irrelevant.

The definition is open with respect to the *class table* — a set of user-declared types (like `Int`, `Array`, etc.). This acts as a source of *tname*s mentioned in Figure 1. There are two relevant features of a *tname*: the optional presence of a type parameter and a superclass defaulted to $\top$. Both are used in the definition of the subtype relation. We denote the lookup operation in the class table as follows: $tname\{\mathsf{X}\} <:: \tau$. This means *tname* have been declared as a parametric type with

1. Pairs distribute over unions, e.g.
$$(\tau_1 \cup \tau_2) \times \tau_3 \;\simeq\; (\tau_1 \times \tau_3) \cup (\tau_2 \times \tau_3).$$
2. Existentials commute, e.g.
$$\exists \mathsf{X}.\exists \mathsf{Y}.\tau \;\simeq\; \exists \mathsf{Y}.\exists \mathsf{X}.\tau$$
3. Existentials distribute over unions:
$$\exists \mathsf{X}.(\tau_1 \cup \tau_2) \;\simeq\; (\exists \mathsf{X}.\tau_1) \cup (\exists \mathsf{X}.\tau_2).$$
4. Existential can re-associate with a single component of a neighbor pair given that the other component has no free occurrences of the bounded variable, e.g.
$$\exists \mathsf{X}.(\tau_1 \times \tau_2) \;\simeq\; \tau_1 \times \exists \mathsf{X}.\tau_2, \quad \text{if } \mathsf{X} \notin \mathrm{fv}(\tau_1).$$
5. η-rule for existentials:
$$tname \;\simeq\; \exists \mathsf{X}.tname\{\mathsf{X}\} \quad \text{if } tname\{Y\} <:: \tau$$

Figure 2. Type constructors interactions w. r. t. subtyping ($\tau_1 \;\simeq\; \tau_2$ means $\tau_1 \;\leqslant\; \tau_2$ and $\tau_2 \;\leqslant\; \tau_1$)

the parameter X and a superclass τ. Note, that X may have free occurrences in τ.

We do not provide a formal definition of well-formedness for the types: this is mostly boring. In particular, all type variables should be $\exists$-bound, all *tname*s should refer to a types contained in the class table, and only those *tname*s can participate in type application that are declared as parametric there.

2.2 Subtyping

The subtype relation in Julia stems from the set of naïve rules for subtyping between individual type constructors (Figure 3). The $\exists$-rules have similar form to the $\cup$-ones, that is why existenial types in Julia are often called the *unionall-types*. Both type constructors have either *forall* or *exists* semantics depending on the side of subtype relation they occur, as is seen in the premises.

The rules should be augmented with the set of equivalences governing the constructors' interaction (Figure 2). The subtype relation under consideration is reflexive (modulo α-equivalence) and transitive closure of the rules from Figure 3 enriched with the equivalences from Figure 2.

Our definition is declarative. In fact, we can only call it a specification of subtyping and not a definition, because the rules in Figure 3 do not define well-founded relation due to the rule $\exists$-L ($\leqslant$ occurs in the negative position). The set of equivalences from Figure 2 calls for some kind of a normal form, but our attempts at one failed: in order to derive certain subtyping derivations algorithmically, we had to anti-normalize the types occasionally.

[1]What we call bounded existentials here corresponds to `t where X` in Julia's syntax. The appearance of existentials in Julia is quite distinct from what we are used to call as such in typed lambda calculi (i.e. there is no dedicated syntactic forms like `pack` / `open`). Instead, it is closely related to the notion from the works on Java wildcards [4, 16] and Java generalized interfaces [17].

$$\frac{}{\tau \leqslant \top}\ \textsc{Top} \qquad \frac{}{\bot \leqslant \tau}\ \textsc{Bot} \qquad \frac{\forall i \in \{1,2\}.\ \tau_i \leqslant \tau}{\tau_1 \cup \tau_2 \leqslant \tau}\ \cup\text{-L} \qquad \frac{\exists i \in \{1,2\}.\ \tau \leqslant \tau_i}{\tau \leqslant \tau_1 \cup \tau_2}\ \cup\text{-R}$$

$$\frac{\forall \tau''.\ \tau_l \leqslant \tau'' \leqslant \tau_u \Rightarrow [\tau''/X]\tau \leqslant \tau'}{\exists X_{\tau_l}^{\tau_u}.\tau \leqslant \tau'}\ \exists\text{-L} \qquad \frac{\exists \tau''.\ \tau_l \leqslant \tau'' \leqslant \tau_u \wedge \tau' \leqslant [\tau''/X]\tau}{\tau' \leqslant \exists X_{\tau_l}^{\tau_u}.\tau}\ \exists\text{-R}$$

$$\frac{\forall i \in \{1,2\}.\ \tau_i \leqslant \tau_i'}{\tau_1 \times \tau_2 \leqslant \tau_1' \times \tau_2'}\ \textsc{Pairs} \qquad \frac{\tau \leqslant \tau' \wedge \tau' \leqslant \tau}{tname\{\tau\} \leqslant tname\{\tau'\}}\ \textsc{App-Inv} \qquad \frac{tname(\{X\}) <:: \tau'' \qquad ([\tau/X])\tau'' \leqslant \tau'}{tname(\{\tau\}) \leqslant \tau'}\ \textsc{Super}$$

Figure 3. Subtyping rules in Julia (simplified)

Decidable algorithmic subtyping for Julia-like type system remains an open problem, both for implementors and academicians. The former struggle just with the decidability part, as witnessed by the number of stack overflow subtype issues on the Julia's bug tracker. The latter have more problems, i.e.

- to find the ways to encode declarative specification handy,
- to come up with the algorithmic definition,
- to prove soundness of the latter with respect to the former.

Our feeling is that along this way it is important to have a tool to fuzzy-test various implementations against each other. We present the corresponding approach in the next section.

3 Fuzzing

The goal is to design an efficient engine to generate pairs of types that are likely to be in a subtype relation. The key idea to solve it is to split the task into two stages. First, generate a simplified skeleton type, a *quasitype*. We should be able to do this fast, avoiding type well-formedness concerns mentioned in Section 2.1.

A quasitype serves as a starting point of a subsequent stage, *mutation*. Given a quasitype, this step produces a list of similar types that have good chances to be in a subtype relation.

3.1 Quasitypes

The purpose of quasitypes is to handle most of the grammar of types avoiding the most peculiar kinds of constructors: binding ones (bring in scoping) and leaves (depend on the class table and also scoping — type variable constructor). Thus, we turn the grammar of types given on Figure 1 into the grammar of quasitypes by:

- removing the $\exists$-constructor,
- removing the *tname*-part of the type application constructor,
- factoring all non-recursive leaf constructors into a single one called *hole* and denoted $\Box$.

Resulting grammar is easily coded in Haskell:

```
data QTau = H              -- Hole
          | A QTau         -- Type Application
          | P QTau QTau    -- Tuple
          | U QTau QTau    -- Union
          deriving (Typeable)
```

An instance of `Typeable` class allows us to use an off-the-shelf solution for the enumeration of the values of `QTau` — the FEAT library[2] [7]. It generates quasitypes up to a given `size` (the number of constructors):

```
deriveEnumerable '' QTau
qts :: [QTau]
qts = take size ∘ concat ∘ map snd $ vs where
  vs = values :: [(Integer, [QTau])]
```

3.2 Mutation: Turning Quasitypes Into Types

Every quasitype can be turned in a type in various ways. Therefore, the type of mutation function is:

```
mutate :: QTau → [Tau]
```

Here, `Tau` is the Haskell datatype mirroring the grammar from Figure 1.

On the way, we might need to use various effects, e.g. non-determinism, state, etc. We call the set of effects we need to use during the mutation `Mut` and go in two steps:

```
mutator :: QTau → Mut Tau
runMut  :: Mut Tau → [Tau]
mutate   = runMut ∘ mutator
```

So far, the simplest possible approach is to put `Mut a = [a]` and `runMut = id`. We argue that a reasonable enumeration strategy requires more than that, and the source of complications is, unsurprisingly, the type variables.

[2]https://hackage.haskell.org/package/testing-feat

```
mutHole :: Mut Tau
mutHole = StateT $ λn →
      zip   (zip knownTypes
                 $ repeat Set.empty)
            [0, 0 . .]
   ++ map (addVarInfo n)
            (varNamesUpTo n)
   ++ [addVarInfo (S n) n]
  where
  addVarInfo n v =
     ((var v, S.singleton $ Var v), n)
```

(a) Mutate a hole

```
mutBin  ::  BinTyCon → QTau → QTau
           → Mut Tau
mutBin con q1 q2 = do
  (s1, vs1) ← mutator q1
  (s2, vs2) ← mutator q2
  return (con s1 s2, vs1 `S.union` vs2)
mutApp    ::  QTau → Mut Tau
mutApp q  =  do
  (s, vs)   ← mutator q
  let ss    = augmenter s vs
  lift $ [(tc s, vs) | tc ← knownTyCons
                     , (s, vs) ← ss]
```

(b) Mutate recursive constructors

```
augmenter :: Tau → Vars → [(Tau, Vars)]
augmenter t vs = map bindVars ds
  where
  ds = deconcatenations $ S.toList vs
  bindVars (bvs, fvs) =
     (trivialAugmenter t bvs, S.fromList fvs)
trivialAugmenter :: Tau → [Var] → Tau
trivialAugmenter t vs = foldr f t vs
  where
  f v t = TExist v Bot Top t
```

(c) Bind free variables

Figure 4. Mutation of a quasitype

3.2.1 Filling In Holes

Every hole turns either a concrete type name or a type variable (we leave out the top and bottom types for now). In the latter case we need to bind corresponding type variable at some point.

For the type names, we feed the algorithm with a set of known types. In particular, we take two type names which are in subtype relation:

```
knownTypes :: [Tau]
knownTypes = map name ["Int", "Number"]
```

For the type variables, what are the possible ways to fill in, say, two holes in a pair $\square \times \square$ with variables? Naïve reasoning would proceed as follows: take the number of fresh variables equal to the number of holes, e.g. `X` and `Y`, and try all combinations of those for the poking. Unfortunately, this produces some redundant results. No matter how we bind free variables, we won't be able to differentiate between, e.g. `X` × `X` and `Y` × `Y`.

For a quasitype with m holes, we should produce, in particular, a type with all the holes replaced with different type variables. Therefore, we definitely need m fresh type variables to mutate this quasitype. What remains to determine is how to enumerate only non-equivalent fillings with those.

First, we assume that the set of fresh variable names is well-ordered; internally they are just the Peano's `Nat`. Then, we propose the following *strategy for filling the holes*:

> *Traversing a type-term in-order, we keep a number of already used type variables, $n \geqslant 0$. When encounter a hole, return all possible fillings of that hole with (1) the* `knownTypes`*, as well as (2) all the used variables (i.e. $0 \ldots n$), and also (3) a filling with the next new variable, increasing the number of used variables in the latter case.*

Therefore, the second candidate for the `Mut` synonym is the function type `Nat -> [(Tau,Nat)]`, or

StateT Nat [ ] *Tau*

on the `transformers`speak.

3.2.2 Binding Free Type Variables: Augmentation

In order to try out various places for bindings of type variables, we need to get an information about the free variables in the current sub-term after mutation. Thus, the final type of the mutation monad is:

```
type Vars   = S.Set Var
type Mut a = StateT Nat [ ] (a, Vars)
```

Having this information at hand, we can add the ∃-nodes on the fly during the mutation — we call this *augmentation*.

Trying to augment every sub-term of a type-term would blow up the state-space. Instead, we use equivalences of Figure 2 to devise the two tactics:

- we augment a free variable only once between each two consequent (not necessarily tight) type application nodes in a subtree, and do this right below the "upper" such node; this way, we need one extra augmentation step at the very end of the mutation process;
- we do not care about the order of binding constructors, because they commute anyway.

3.2.3 Bringing It All Together

The essential bits of the implementation for the above sketch of the algorithm are given on the Figure 4. The top-level `mutator` function mentioned at the beginning of the Section 3.2 simply dispatches between the `mut`-functions of subfigures (4a) and (4b).

The hole mutation function, `mutHole` from (4a), closely follows the strategy laid up in Section 3.2.1: parts (1)–(3) of the strategy are exactly three parts of the definition (`zip`, `map` and singleton list). Parts (2) and (3) use an auxiliary function `addVarInfo` which injects a variable named `v` into the corresponding state given the number of already used variables, `n`.

Note, that in (1)–(3) we try to order the types in the resulting list from less general (w.r.t. $\leqslant$) to more so. We prefer to try less then $\binom{m}{2}$ subtypings during the actual testing. In fact, when we test $\tau_i \leqslant \tau_j$, it is always the case that τ_i comes first in the list.

Mutating binary constructors (`mutBin`, 4b) is boring, even `Applicative` (and, therefore, the subject to `ApplicativeDo` extension in our build script). It can be done the same way for both, pairs and unions. On the other hand, type applications are delicate: after mutation of the τ-part from $tname\{\tau\}$, we need, first, to augment the result, and then, to fill in the $tname$-part. For the latter, we use a `knownTypes`-analog for the type operators: `knownTyCons`. In our experiments they were `Ref` and `Val` of Julia.

We omit fuzzing the bounds of a type variable. For the sufficiently large type-terms, the subtype algorithm will have to update bounds on the way and, therefore, non-trivial bounds are tried out anyway. On the other hand, even simple pathological bounds might yield interesting results, and we leave this for the future work.

The `augmenter` (4c) has to do combinatorial enumeration of all partitions of a given set of free variables into two subsets: the variables that should stay free, and those we are going to bind at this point. This is done by converting the set to a list and computing all `deconcatenations`[3] of the list. The fact that bindings enumeration can be done in this relatively cheap manner (rather then, e.g., doing `permutations` in addition) roots in $\exists$-commutativity (Figure 2).

As noted in Section 3.2.2, we have to add one augmentation step after the whole recursive mutation process is done. Hence, we update the definition of `mutate` from the beginning of 3.2:

$$\begin{aligned} mutate = {} & runMut \\ & \circ (\ggg lift \circ uncurry\ augmenter) \\ & \circ mutator \end{aligned}$$

4 Evaluation

We performed a number of experiments on fuzzy-testing of our implementation of the subtyping algorithm [18] against the Julia VM's one. We pipe the fuzzer's[4] output to a Julia script which searches for discrepancies between the two algorithms. Most of the cases found were due to the bugs in our implementation. But we also filed three bug reports to the Julia's tracker: two of them were fixed (#26180, #26654), and one remains open (#27101).

[3] http://hackage.haskell.org/package/HaskellForMaths
[4] https://github.com/prl-prg/subtype-fuzzer

5 Related Work and Alternative Designs

Random Sampling vs. Exhaustive Enumeration The seminal paper on QuickCheck [6] inspired numerous works on generative techniques for testing. The most immediate and the most relevant to us is, probably, the work on SmallCheck and Lazy SmallCheck [15]: 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 is not especially stable on the large inputs — often times it crashes for those.

Also, QuickCheck and SmallCheck aim at testing of properties. In our case, there is only one property of interest (consistency with Julia VM) which is "external" to Haskell, i.e. implemented in a different language. In fact, we tried to do the actual testing in Haskell, shell-calling the VM, but this performed poorly. Hence, our goal moved from testing the properties to generation of terms subject of certain properties: well-formedness and likeliness to be in subtype relation. The rest is done outside Haskell.

Generation of Constrained Data Generation of constrained data has been studied since, at least, QuickCheck. The Lazy SmallCheck work mentions that lazy evaluation of the predicates can speed-up the generation process. This resembles the technique called *narrowing* in logic-functional programming. The `lazy-search` library[5] improves on this approach and adds reflective abilities of the search algorithm with respect to a given predicate, making narrowing even more relevant.

In our experiments, the attempt at generation of well-formed terms did not perform well. The idea of to-be-augmented quasiterms turns out to be more efficient.

Narrowing and binary (n-ary) relations The technique of narrowing for testing *inductive relations* suggests that the generation engine can leverage the full access to the predicate definition instead of viewing it as a black box. This was done either by supplying a DSL for the predicate definitions [10] or by using deep reflective abilities of the language, e.g. Coq [11]. Again, for well-formedness, it doesn't perform as well as quasiterm-generation. For subtype relation, it is not even clear how to extend the framework for binary (n-ary) relations. Also, our relation at hand is both, far from being inductive (which is essential for Coq) and complicated enough to bar from using a simple DSL for encoding.

The approach for testing a binary relation was thought of in the very QuickCheck paper for testing of a unification algorithm [6, 5.1.4]. Essentially, we build on it.

Why fuzzing Fuzzing often means taking a valid input and mutating it into several non-necessarily valid ones, see e.g. QuickFuzz [8]. Our approach goes the other way round:

[5] https://hackage.haskell.org/package/lazy-search

it takes an undercooked type, a quasitype, and mutates it into several well-done ones with the bonus property of being likely in subtype relation. We could start from a well-formed type and try to mutate it likewise, but preserving well-formedness during the mutation seems not easier then bringing it in.

Subptyping is often too hard for a formall study of its algorithmic formulations: e.g. for Java it is both undecidable [9] and jeopardizes the soundness of the language[1]. In this field, the fuzzing seems to be an almost indispensable tool.

6 Conclusion and Future Work

We have presented a fuzzy-testing technique for the complex not-obviously-inductive binary relation of subtyping. The main ideas are: first, factoring the grammar of terms involved (types, in our case) into two parts, following 20/80-rule — 80% of grammar accounts for 20% of complexity — and second, mutating the terms yielded from the standard enumeration algorithm applied to the simplified grammar. We believe these ideas can be applied to the task of fuzzy-testing in similar settings. We mention here a set of possible future enhancements that could make the ideas more accessible.

The task of factoring a grammar might be automated through the metaprogramming or generic programming techniques. In fact, there are several relatively simple rules for the factoring: merge all non-recursive nodes, remove all naming-related sub-nodes, remove all binding nodes (i.e., those holding var-related sub-nodes). These are well in scope of the code manipulation techniques with one possible complication, namely: when use popular generic representations, e.g. `GHC.Generics` or `generics-sop`, one have to apply certain effort to handle datatype-recursion [3].

On the other hand, the grammar datatype can be designed with that transformation in mind from the very beginning. In that case, the growable trees technique [12] could be applied.

In contrast, the mutation phase seems to heavily depend on the properties of the relation at hand: we used most of the equivalences from Figure 2 in our case study. The only piece of advice to suggest here is: the more algebraic laws of the relation are known the more chances to keep search-state tractable.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 695412).

References

[1] Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 838–848. https://doi.org/10.1145/2983990.2984004

[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] Anna Bolotina and Artem Pelenitsyn. 2018. Handling Recursion in Generic Programming Using Closed Type Families. (2018). Accepted talk in Trends in Functional Programming.

[4] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. 2008. A Model for Java with Wildcards. In *European Conference on Object-Oriented Programming (ECOOP 2008)*. https://doi.org/10.1007/978-3-540-70592-5_2

[5] 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

[6] 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

[7] 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

[8] 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

[9] Radu Grigore. 2017. Java Generics Are Turing Complete. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3009837.3009871

[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] Shayan Najd and Simon Peyton Jones. 2017. Trees that Grow. *Journal of Universal Computer Science* 23, 1 (jan 2017), 42–62.

[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. Hriţcu, 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] 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

[16] Ross Tate, Alan Leung, and Sorin Lerner. 2011. Taming Wildcards in Java's Type System. In *Conference on Programming Language Design and Implementation (PLDI 2011)*. https://doi.org/10.1145/1993498.1993570

[17] Stefan Wehr and Peter Thiemann. 2009. On the Decidability of Subtyping with Bounded Existential Types. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS '09)*. Springer-Verlag, Berlin, Heidelberg, 111–127. https://doi.org/10.1007/978-3-642-10672-9_10

[18] Francesco Zappa Nardelli, Julia Belyakova, Ben Chung, Artem Pelenitsyn, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA (Nov. 2018), to appear.

[19] Qirun Zhang, Chengnian Sun, and Zhendong Su. [n. d.]. 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