Generating Well-Typed Strata Programs, Correctly (Experience Report)

ERNEST NG*, Cornell University, USA
JOSHUA M. COHEN, Amazon Web Services, USA
MICHAEL HICKS, Amazon Web Services, USA and University of Pennsylvania, USA

Developing generators of random well-typed programs for property-based testing is notoriously difficult and requires significant developer effort. Generators can either be silently *unsound* (producing ill-typed terms) or *incomplete* (incapable of generating certain well-typed terms). We discuss a methodology for using LLMs to create provably sound and complete generators in a proof assistant, demonstrating its practicality by applying our methods to Strata Core, an intermediate verification language under development at AWS. Strata Core includes many features that make developing and reasoning about such generators difficult, such as polymorphic n-ary function application. We demonstrate the utility of our generators by using them to test Strata Core transformations and infrastructure, revealing dozens of previously unknown bugs.

1 Introduction

When designing and implementing a new programming language, property-based testing (PBT) can be a valuable tool in one's arsenal. One key challenge in PBT research is the *constrained random generation problem* [10]: to effectively test properties with preconditions, we need ways to constrain random sampling to the set of all precondition-satisfying values only. For example, to test a compiler, a well-studied approach is to generate random well-typed programs that uncover bugs in the compiler's typechecking and optimization logic [17, 19]. However, user studies reveal that many software developers consider the task of writing constrained PBT generators that only produce values satisfying complex preconditions to be "tedious" and "high-effort" [8]. Indeed, designing generators that only produce well-typed programs is challenging and is an active area of research [6, 7, 18]. At Amazon Web Services (AWS), the process of developing a generator for well-typed expressions in the Cedar authorization policy language [3] was reported to take six developer-months, with the generator itself spanning more than 5,000 lines of code [4].

One solution is *constrained generator synthesis*: given a predicate P, synthesize a generator g such that $\forall x,\ x \in [\![g]\!] \iff P(x)$, where $[\![g]\!]$ is the set of values (the *support*) that can be produced by g. That is, x can be generated by g if and only if $P(x)$ holds. Such a generator is deemed *sound* and *complete*. Existing constrained generator synthesis tools in the Rocq and Lean proof assistants allow users to automatically derive generators and their correctness proofs [10, 15, 20]. Such tools can even be used to "test" theorem statements by generating precondition-satisfying inputs [4].

We sought to perform PBT in Lean for exactly this purpose: to "test" theorems that are not yet proven. At AWS, we are currently developing Strata, an intermediate verification language implemented in Lean that is already under active use. PBT is a critical tool for gaining confidence about parts of Strata's implementation whose proofs are still in development, as well as extending assurance to components which cannot be proven. For example, while we cannot prove a round-trip property regarding Strata's parser and pretty-printer (since these components perform I/O), we can test this property by generating random well-typed programs.

However, we found constrained generator synthesis tools for Lean [5, 10] to be too restrictive, as they only support certain classes of predicates (e.g., inductive relations) and were unable to

*Work done during an internship at Amazon Web Services.

Authors' Contact Information: Ernest Ng, Cornell University, Ithaca, NY, USA, ernest@cs.cornell.edu; Joshua M. Cohen, Amazon Web Services, Boston, MA, USA, cohenjo@amazon.com; Michael Hicks, Amazon Web Services, Arlington, VA, USA and University of Pennsylvania, Philadelphia, PA, USA, mwh@cis.upenn.edu.

produce generators suitable for Strata. So, we wondered: Could we instead use LLMs to synthesize generators of well-typed Strata programs, asking the LLM to prove them sound and complete with respect to Strata's typing specification? This work reports on our experience.

Concretely, we require LLMs to use Basalt [9], a recent Lean framework for both implementing and reasoning about generators, which enables direct mechanized proofs (including LLM-synthesized ones) about generators.[1]

We found that when we prompted an LLM to synthesize a constrained generator in Basalt and supplied it with proof obligations for soundness and completeness, the LLM would sometimes "cheat" by adding extra hypotheses to the theorem statements. Even after removing these spurious side-conditions by manually inspecting these proofs, we found that the LLM-synthesized generators were not necessarily of good quality. There were structural issues, such as an inability to effectively generate multi-argument function applications, which we solved by devising custom generation rules using results from the PBT literature [19]. There were also distributional issues, such as trivial values (e.g., integer literals) being generated too frequently, which required us to tune generators by modifying generator weights and visualizing their distributions [11]. After putting these solutions in place, our generators were effective in practice, and we were able to find 37 bugs when testing more than 80 properties (§5).[2]

The remainder of this extended abstract discusses these problems in detail and the solutions we developed. This experience suggests open problems that, if solved, could make LLM-based generator synthesis more reliable.

2 Background: Strata

Strata [21] is a new foundation for sound code reasoning tools under development at AWS. It provides a platform to enable reasoning about multiple source languages and backend analysis engines (e.g., SMT solvers, proof assistants, model checkers). Broadly, Strata consists of a series of *dialects* allowing programs to be represented at different levels of abstraction and transformations between these dialects. Strata Core, the central dialect of Strata, takes inspiration from existing intermediate verification languages such as Why3 [2] and Boogie [1]. Strata is implemented in Lean, with dialects and transformations backed by formal semantics and mechanized correctness proofs.

Strata Core is a sufficiently expressive language to make designing sound and complete PBT generators quite non-trivial. It is stratified into several layers: the expression language is a λ-calculus with Hindley–Milner-style prenex polymorphism, uninterpreted calls to *ambient functions*, equality, and existential and universal quantification. The imperative fragment includes mutable variables (including non-deterministic assignment) and structured control flow (including loop invariants). Top level declarations include abstract types, pure functions, stateful procedures with Dafny-style pre and post-conditions, user-defined algebraic datatypes, and type aliases.

3 LLM-Based Synthesis of Verified Generators: What Worked

3.1 Generating Well-Typed Terms: The Basic Approach

We found that an LLM (Claude Opus 5.0) was capable of synthesizing generators for well-typed terms based on a classic approach from the PBT literature. Pałka et al. [19] showed that to generate well-typed terms in the simply typed λ-calculus (STLC), one can view typing rules as *generation rules*. Given a typing context Γ and a target type τ, one can generate a well-typed term e of type τ by

[1] Basalt allows us to prove generators sound and complete against any Lean proposition (e.g., functions returning `Prop`), not just the limited classes of predicates supported in prior work.

[2] Our generators are open-source and available at https://github.com/strata-org/strata-generators.

randomly selecting a typing rule whose conclusion states that e has type τ in context Γ ($\Gamma \vdash e : \tau$), and constructing e by generating subterms that satisfy the rule's premises. For example, to generate a term with the product type $\tau_1 \times \tau_2$, we need to generate a pair of expressions (e_1, e_2). The premises of the typing rule for pairs (Figure 1) indicate that e_1 and e_2 should be recursively generated to have types τ_1 and τ_2 respectively.

Critically, the STLC typing rules are *syntax-directed*: the structure of the typing derivation is directly determined by the term's syntactic structure. This feature means that a generator for each typing rule can be obtained directly by reading each rule "bottom to top", a fairly mechanical process. Indeed, when supplied with typing rules for the expression layer of Strata Core (which resemble the STLC extended with prenex polymorphism), an LLM was able to automatically apply Pałka et al.'s approach to generate well-typed terms in this language.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

Fig. 1. STLC typing rule for pairs.

3.2 Proving Soundness and Completeness

To ensure that the LLM-synthesized term generator is correct, we instruct the LLM to prove the generator sound and complete using Basalt [9], a Lean framework for implementing and reasoning about generators. Basalt allows one to reason about the *support* of a generator g, written $[\![g]\!]$, the set of values produced by g with non-zero probability. Concretely, the correctness theorems for the term generator `genExpr` reason about the set of all terms of size at most n that can be generated, written $[\![\texttt{genExpr}\ \Gamma\ \tau\ n]\!]$. (In practice, to control the size of generated terms, we follow the common approach of adding a `size` parameter to a generator and making the generator structurally recursive on this parameter [15, 20].) The theorem statements of soundness and completeness are:

Theorem (Soundness). $\forall\, \Gamma\ e\ \tau\ n,\ e \in [\![\texttt{genExpr}\ \Gamma\ \tau\ n]\!] \implies \Gamma \vdash e : \tau$

Theorem (Completeness). $\forall\, \Gamma\ e\ \tau\ n,\ \Gamma \vdash e : \tau \land \texttt{size}\ e \leq n \implies e \in [\![\texttt{genExpr}\ \Gamma\ \tau\ n]\!]$

Following Rocq's QuickChick PBT framework [15, 20], the proofs for both theorems are performed by induction on the size parameter n and case analysis on the target type τ. For syntax-directed type systems such as the STLC, the derivation of the generator and its correctness proofs are relatively mechanical: see Paraskevopoulou et al. [20] for further details. In our experience, an LLM was able to synthesize generators and its correctness proofs for most of Strata Core.

3.3 Composing Generators

Since Strata Core consists of multiple language layers (§2), our LLM-synthesized generators likewise have a layered structure. Both the generators and their correctness proofs compose to produce results for full Strata Core programs. For example, the imperative layer includes a variable initialization command `init x : Int := e` where e is an expression; our command generator invokes the expression generator, and similarly for the proofs.

For the expression layer of Strata Core, we generate well-typed terms in the type-directed manner described above. For the imperative fragment of Strata Core, we generate commands and statements sequentially in the style of Lampropoulos and Pierce [16], extending the typing context when new variables are defined in the randomly generated program. The typing context is similarly extended whenever we generate top-level declarations such as type definitions, functions, and procedures, so that later portions of the randomly generated program can refer to earlier definitions.

4 Where LLM-Based Generator Synthesis Fell Short

4.1 Tuning Generators

One aspect of LLM-based generator synthesis which requires human judgment is generator *tuning*, the process of ensuring that the generator produces a "good" distribution of input values (one where "interesting" inputs appear often) [23]. A uniform distribution may not necessarily always be best for testing: for example, to test a loop unrolling optimization, one may want to generate programs containing loops more frequently. Reasoning about *how* to tune a generator to achieve an idealized distribution can also be challenging: in a PBT user study, multiple developers cited manual tuning as a source of "mental strain" [8], as they may not know *a priori* what precise distribution they are after [23].

Furthermore, an LLM-synthesized generator, even when proven correct, can have a poor distribution for testing. For instance, when we began developing generators for Strata Core, Basalt only provided one combinator `pick` for fair random choice between two generators. As a result, the LLM opted to nest `pick` calls to implement cases when it needed to choose between more than two sub-generators. For example, to generate terms of type `Bool`, the LLM would produce a chain of nested `pick` calls, where the top-most branch generates a Boolean literal (Figure 2). This structure caused Boolean literals to be generated nearly half the time, with each additional level of nesting halving the probability of reaching the sub-generators below it. Although the generator was proven sound and complete, it was not practically useful: Boolean literals are unlikely to trigger useful logic within a compiler.

```
pick
  <Generate Boolean literal>   -- 50% chance
  pick
    <Generate if-expressions>  -- 25% chance
    pick
      <Generate equality checks> -- 12.5%
      ...
```

Fig. 2. Nested `pick` calls in a naïve LLM-synthesized generator.

To resolve this issue, we manually developed Basalt combinators for fair and weighted n-ary choice, along with lemmas for reasoning about their support. We then prompted the LLM to refactor the generator and its correctness proofs using these new combinators, then used the Tyche PBT visualization tool [11] to visualize the frequency of different expression forms (e.g., applications). We found that the Tyche visualizations were effective at allowing the LLM to iteratively tune the generator based on empirical measurements.

4.2 Generating Ambient Function Applications

In Strata Core, the function that is invoked during a function application need not be a λ-abstraction: any function present in the ambient typing context (e.g., user-defined and standard library functions) can be invoked. We call such functions *ambient functions*.

The default approach to generating well-typed terms is ineffective at generating ambient function applications. Consider the STLC typing rule for function application (Figure 3). Observe that the type τ' for the function argument only appears in the rule's premise but not its conclusion. Since τ' is unconstrained, the corresponding generator can just generate any random type for τ'.

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

```
do
  -- Generate a random argument type τ'
  let τ' ← genType
  let e₁ ← genExpr Γ (Arrow τ' τ)
  let e₂ ← genExpr Γ τ'
  return (App e₁ e₂)
```

Fig. 3. Top: the STLC application typing rule. Bottom: the corresponding generator.

However, this approach is impractical when one wants to generate calls to multi-argument functions. For example, consider the function `String.take : String → Int → String`, which extracts the first n characters from a string. To generate a well-typed call to `String.take` using

the classic approach, we must apply the STLC application rule twice and hope that we generate `String` and `Int` as the two argument types, which is unlikely since argument types are randomly generated! (Indeed, a naïve LLM-synthesized generator rarely produced multi-argument function applications.)

To solve this problem, Pałka et al. [19] proposed an alternative generation rule that is logically unnecessary but useful for generation. To generate a call to the ambient function f, one can inspect f's type to determine its argument types. Then, one can generate argument terms with these argument types and directly apply f to these arguments, resulting in a well-typed term that is guaranteed by construction to invoke f. This generation rule obviates the need to randomly guess argument types: the types for all the arguments are pre-determined by f's type. After we prompted the LLM to refactor the term generator using this rule, the frequency of ambient function applications vastly increased.

Handling polymorphic ambient functions. Naive generation of polymorphic ambient functions results in a related issue: a random type instantiation is very unlikely to result in a well-typed term at the target type. We follow a similar approach, and our generator unifies type variables with concrete types, freely generating types for type variables that are unconstrained. The LLM was able to prove this mechanism correct despite the non-trivial proof: the completeness proof relied on proving completeness of the Strata Core unification algorithm.

Discussion. The need for a custom generation rule suggests that proving a generator sound and complete is insufficient for it to be practically useful. The default STLC application rule does not directly lead to a practical generation algorithm, and others have observed similar phenomena for well-typed term generators in other settings [7, 13, 18].

Function applications were not the only case in Strata where syntax-directed approaches were insufficient. For instance, we also needed to design a custom generation rule to produce mutually recursive algebraic data types (ADTs) subject to complex well-formedness constraints such as type inhabitation. At a high level, designing such a generator was complex, as the declarative specification for type inhabitation does not immediately suggest a straightforward generation algorithm. Our eventual generator, found after multiple attempts, carefully produces mutually recursive ADTs that are well-formed by construction, randomly permuting types and constructors to achieve completeness.

5 Properties Tested and Bugs Found

We now report on our testing experience using these generators and discuss bugs found via PBT.[3] We tested more than 80 properties spanning the Strata codebase. For example, we test that the parser and pretty-printer round-trip (`parse(print(p)) = p` for all randomly generated programs p), and that common sub-expression elimination does not result in unbound variables.

Table 1 provides an overview of the number of bugs we found via PBT. We now briefly discuss some of these bugs. When testing the round-trip property for the parser and pretty-printer, we found that the polymorphic function `foo<s>(x: s) { ... }` (where `s` is a type variable) failed to parse, as `<s` was also Strata's concrete syntax for the signed less-than operator over bit-vectors. This syntactic ambiguity meant that users were unable to define polymorphic functions with type variables beginning with `s`. Another bug we found involves Strata's symbolic execution engine, which collects proof obligations expressed in the form of assertions in the source program. We found that in programs containing two consecutive non-deterministic branches, proof obligations

[3] See our generators' documentation [22] for the full list of properties tested and bugs found.

in one of the conditionals would be silently dropped. These bugs are emblematic of the power of PBT: they involve inputs that humans would be very unlikely to write in unit tests or real programs.

Besides implementation bugs, we also discovered bugs in some of the specifications used throughout the Strata Lean codebase. For example, when attempting to prove the ADT generator complete against the spec for well-formed ADTs, we found that the spec allowed for ill-kinded applications of type constructors, such as `Seq<a, a>`, where the `Seq` type constructor is applied to two type arguments instead of one. The initial version of the LLM-synthesized completeness theorem required extra side-conditions to rule out such ill-kinded type constructor applications, and when manually inspecting the proof, we discovered this bug in the specification. This anecdote demonstrates how proving constrained PBT generators sound and complete against a specification can also help uncover bugs in the specification itself!

Table 1. Bugs found, by category.

Category	Bugs Found
Parser	2
Pretty-printer	10
Type checking & type inference	3
Program transformations	6
Symbolic execution engine	1
SMT encoding	4
Auto-derived ADT functions	2
Type system specification	9
Total	37

6 Conclusion

We began this project by asking: to what extent are LLMs capable of synthesizing generators of well-typed Strata programs and proving them sound and complete against the typing specification? We now address this question. We found that LLMs are able to mechanically derive generators and their correctness proofs when typing rules are syntax-directed. However, even when proven correct, an LLM-synthesized generator can suffer from structural issues, as well as distributional issues in which trivial values are generated too frequently. We resolve structural issues by devising custom generation rules, and we address distributional issues by iteratively tuning generators, guided by visualizations of generated values' frequencies.

Our experience raises some open questions: what are properties about a generator's distribution that ought to be proved to ensure that a generator is practically useful for testing? As ongoing work, we are working on proving other probabilistic properties about our generators of Strata programs, using other features of Basalt (e.g., the expected number of loops in a randomly generated Strata program). In particular, we hope to understand how a generator's `size` parameter (§3.2) affects the frequency at which different language features appear in generated programs. We recognize that in general, distributions should be tailored to different testing scenarios (there is no "one size fits all" distribution), but there are questions that are worth investigating regardless of the particular testing scenario. We observed in §4.2 how PBT practitioners often design custom generation rules to make generators produce programs that exercise non-trivial language features more frequently. Can we devise techniques to help generator designers (and LLMs) come up with these bespoke generation rules more easily?

More generally, can we prove that two versions of the same generator, e.g., a slow one and an optimized one, share similar distributions? This result can provide assurances that if we optimize a generator, the resultant generator's distribution will not be unexpectedly skewed. Additionally, for coverage-guided PBT generators [12, 14], how should coverage information influence the generator's distribution? In general, how can we quantitatively measure changes to a generator's distribution?

As LLMs become increasingly capable of synthesizing verified generators, the key challenge of generator synthesis shifts from proving them correct to proving their distributions useful. We hope that our experience can motivate the PBT community to develop tools and specifications to address this latter question.

References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects (FMCO)*.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Workshop on Intermediate Verification Languages (Boogie)*.

[3] Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, John Kastner, Anwar Mamat, Matt McCutchen, Neha Rungta, Bhakti Shah, Emina Torlak, and Andrew Wells. 2024. How We Built Cedar: A Verification-Guided Approach. In *Foundations of Software Engineering (FSE Companion)*.

[4] Segev Elazar Mittelman, Harrison Goldstein, and Leonidas Lampropoulos. 2026. Testing Theorems, Fully Automatically. *Proc. ACM Program. Lang.* 10, OOPSLA2 (2026). doi:10.1145/3839453

[5] Segev Elazar Mittelman, Ernest Ng, and Michael Hicks. 2026. Specimen: Automatically Derived Generators, Enumerators, and Checkers for Inductive Relations in Lean. https://github.com/strata-org/specimen. Accessed October 2026.

[6] Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *European Symposium on Programming (ESOP)*.

[7] Justin Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2024. Generating Well-Typed Terms That Are Not "Useless". *Proc. ACM Program. Lang.* 8, POPL (2024). doi:10.1145/3632919

[8] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *International Conference on Software Engineering (ICSE)*.

[9] Harrison Goldstein, Ernest Ng, and Michael Hicks. 2026. Basalt. https://github.com/hgoldstein95/basalt

[10] Harrison Goldstein, Hila Peleg, Cassia Torczon, Daniel Sainati, Leonidas Lampropoulos, and Benjamin C. Pierce. 2026. The Search for Constrained Random Generators. *Proc. ACM Program. Lang.* 10, PLDI (2026). doi:10.1145/3808329

[11] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. 2024. Tyche: Making Sense of PBT Effectiveness. In *User Interface Software and Technology (UIST)*.

[12] Zac Hatfield-Dodds. 2026. HypoFuzz. https://github.com/Zac-HD/hypofuzz

[13] Casey Klein, Matthew Flatt, and Robert Bruce Findler. 2010. Random testing for higher-order, stateful programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[14] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. *Proc. ACM Program. Lang.* 3, OOPSLA (2019). doi:10.1145/3360607

[15] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating good generators for inductive relations. *Proc. ACM Program. Lang.* 2, POPL (2017). doi:10.1145/3158133

[16] Leonidas Lampropoulos and Benjamin C. Pierce. 2026. *QuickChick: Property-Based Testing in Rocq*. Software Foundations, Vol. 4. Electronic textbook, Chapter TImp: Case Study: a Typed Imperative Language. https://softwarefoundations.cis.upenn.edu/qc-current/TImp.html Version 2.1.

[17] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). doi:10.1145/3428264

[18] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of compilers. *Proc. ACM Program. Lang.* 1, ICFP (2017). doi:10.1145/3110259

[19] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Automation of Software Test (AST)*.

[20] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing correctly with inductive relations. In *Programming Language Design and Implementation (PLDI)*.

[21] Strata Team. 2026. The Strata Core Language Syntax. https://strata-org.github.io/Strata/langdef/html-single/. Accessed October 2026.

[22] Strata Team. 2026. Strata-Generators: Properties Tested and Bugs Found. https://github.com/strata-org/Strata-Generators/blob/main/properties_bugs_found.md. Accessed October 2026.

[23] Ryan Tjoa, Poorva Garg, Harrison Goldstein, Todd Millstein, Benjamin C. Pierce, and Guy Van den Broeck. 2025. Tuning Random Generators: Property-Based Testing as Probabilistic Programming. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025). doi:10.1145/3763082