Testing Theorems, Fully Automatically

SEGEV ELAZAR MITTELMAN✉, University of Maryland, USA
HARRISON GOLDSTEIN, University at Buffalo, SUNY, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA

While the ultimate goal of interactive theorem proving is to *prove* theorems, it can really help to *test* them first. Testing theorems, specifically using *property-based testing*, helps users identify incorrect definitions and theorem statements before they waste time on a proof that could never succeed. Unfortunately, the testing infrastructure provided by modern theorem provers has yet to reach its full potential. Even QuickChick, the state-of-the-art property-based testing framework for Rocq, which offers random generation for data satisfying inductively defined relations, often requires substantial effort and expertise to be used effectively. This is in part because this effectiveness is heavily sensitive to both the order that hypotheses appear within a theorem, and to the order that inductive constraints appear within the inductive relations involved.

In this paper, we present a novel strategy for testing theorems that is highly effective, fully automatic, and robust to equivalent formulations of theorem and definition statements. To do so, we characterize the exponentially large space of possible QuickChick-style properties and generators as solutions to a constrained scheduling problem. To find the best property or generator in this space, we estimate effectiveness by introducing a notion of "density" for inductive relations, which approximates the tendency for a generator to succeed given arbitrary inputs. We implement our algorithm on top of the QuickChick framework for Rocq and evaluate it in a number of case studies from the literature, demonstrating that our push-button automation is on par with and in some cases even more effective at finding bugs than expertly handcrafted tests.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: property-based testing, QuickChick, Rocq, inductive relations

ACM Reference Format:
Segev Elazar Mittelman, Harrison Goldstein, and Leonidas Lampropoulos. 2026. Testing Theorems, Fully Automatically. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 321 (October 2026), 29 pages. https://doi.org/10.1145/3839453

1 Introduction

Formal verification in a proof assistant such as Rocq [40], Lean [28], or Agda[29] primarily involves specifying theorems using inductively defined types and relations before proving them via tactics or dependently typed programs. Attempting such proofs can be an exercise in frustration, as it is difficult to know if a theorem is incorrectly specified without trying (and failing) to prove it. Users can spend hours iterating on such proofs before realizing they are actually trying to prove `False`.

Testing offers a natural solution to this problem by uncovering such discrepancies early on in the proving cycle. As a result, Rocq's QuickChick [22, 25, 32], Isabelle's QuickCheck [2, 4], Lean's Plausible [8], and Agda's QuickCheck [14, 15] all offer ways to test theorems before trying to prove them, capitalizing on decades of research into property-based testing. QuickChick, in particular, represents the state-of-the-art in property-based testing for proof assistants providing users with a

Authors' Contact Information: Segev Elazar Mittelman, University of Maryland, College Park, USA, segevem@umd.edu; Harrison Goldstein, University at Buffalo, SUNY, Buffalo, USA, hgoldste@buffalo.edu; Leonidas Lampropoulos, University of Maryland, College Park, USA, leonidas@umd.edu.

ACM 2475-1421/2026/10-ART321
https://doi.org/10.1145/3839453

comprehensive toolbox for facilitating effective testing of theorems: domain specific languages for expressing properties and generators, support for deriving generators for random inputs satisfying an inductive relation [24, 34], as well as ways of checking whether such a relation holds [31]. However, despite all these tools being available, QuickChick users still face a daunting amount of effort before being able to test their conjectures.

To dig a bit deeper into this issue, let's turn to a familiar theorem, preservation for a typed lambda calculus: if an expression is well typed and it reduces, then the resulting expression should also have the same type.

```
Theorem preservation: forall Γ e e' τ, typing Γ e τ -> e ⇓ e' -> typing Γ e' τ.
```

How should we test such a theorem? The primary task is to instantiate the four universally quantified variables: an environment Γ, expressions e and e', and a type τ. A naive approach would be to start by generating all universally quantified variables according to the grammar of their Rocq types; then, we would need to check whether the generated values satisfy the hypotheses (that is, whether `e` has type τ in an environment Γ, as well as whether `e` steps to `e'`); finally, after filtering out the generated values that don't satisfy these premises, we could check whether the conclusion holds (i.e. if `e'` also has type τ in Γ). Figure 1 depicts QuickChick's notation for such an approach, using the `arbitrary` typeclass method to invoke a generator for an unconstrained element of a type, the `forAll` combinator to sample from such a generator and bind the result to a variable, the `check` combinator to decide whether an inductive relation holds [31], and implication (`==>`) to combine pre- and post-condition checks together. Unfortunately, this approach is ineffective for testing as both the typing and evaluation relations are too *sparse*: arbitrary contexts, terms and types are incredibly unlikely to satisfy either `typing` or ⇓.

```
Definition preservationProp :=
    forAll arbitrary (fun Γ =>
    forAll arbitrary (fun τ =>
    forAll arbitrary (fun e =>
    forAll arbitrary (fun e' =>
    check (typing Γ e τ) ==>
    check (e ⇓ e') ==>
    check (typing Γ e' τ))))).
```

Fig. 1. Property for testing preservation using QuickChick's type-based generators.

A better answer, drawing inspiration from work on testing compilers and typecheckers for functional languages [18, 27, 30], is to first generate an arbitrary environment Γ and a type τ, then a term `e` that is well-typed at τ in Γ, and then reduce it to produce an `e'`. In Figure 2, we use the `arbitrarySuchThat` typeclass method to directly generate inputs satisfying an inductive relation, which can leverage QuickChick's derivation support to automatically derive the many necessary typeclass instances [24].

```
Definition preservationProp :=
    forAll arbitrary (fun Γ =>
    forAll arbitrary (fun τ =>
    forAllMaybe (arbitrarySuchThat (fun e => typing Γ e τ)) (fun e =>
    forAllMaybe (arbitrarySuchThat (fun e' => e ⇓ e')) (fun e' =>
    check (typing Γ e' τ))))).
```

Fig. 2. Property for testing preservation using specification-based generators.

Testing preservation with this property using the proper generators derived by QuickChick can be extremely efficient, as a recent experience report found [36]. Unfortunately, achieving this efficiency requires significant user expertise: what *order* should we try to satisfy the theorem's constraints in? Concretely, in the case of preservation, why is it preferable to first generate an environment and a

type arbitrarily, and then generate a well-typed term rather than any other way around? Intuitively, the reason is that most types are inhabited by many terms, while most terms are not well-typed. In other words, given a random type, we can often successfully generate a term of that type, but not vice-versa. QuickChick, however, has no way of making such determinations for arbitrary theorems involving arbitrary inductive predicates; as a result, it relies on users and their expertise to write efficient properties, as well as to specify which generators to derive. Unfortunately, figuring out which generators should be derived to maximize the effectiveness of testing also requires significant expertise itself.

As we will discuss in detail later on in the paper, three distinct choices are left up to the users which *significantly* impact generation behavior. First, users need to select which indexes of an inductive relation should be considered inputs, and which should be outputs, a notion reminiscent of a *mode* in logic programming [11]. When testing preservation, for example, it is crucial to ensure that terms, rather than types, are the output of the generator. Second, any relations that are referred to within a constructor of an inductive, need to have their generator derived in advance. Just like with top-level generators, QuickChick doesn't have a way of determining how an inner relation should best be satisfied, and relies on user input for that as well. For instance, if the typing rule for variables looks something like the following:

```
| TyVar :
    forall Γ x τ, bind Γ x τ -> typing Γ (Var x) τ,
```

then a generator for `bind` that inputs an environment Γ and a type τ and randomly generates a variable that has type τ in Γ is required to obtain an effective generator for well-typed terms.

Last, but definitely not least, the *order in which hypotheses are specified within an inductive relation* can greatly impact generation. Consider the typing rule for application:

```
| TyApp : forall Γ e1 e2 τ1 τ2, typing Γ e1 (TFun τ1 τ2) -> typing Γ e2 τ1 ->
       typing Γ (App e1 e2) τ2.
```

How does QuickChick make the crucial choice of which hypothesis to satisfy first? If it could make such a determination, then QuickChick would also be able to solve the toplevel problem of testing theorems: in Rocq, the type of a constructor and the type of the theorem share a similar shape. Once again, however, QuickChick can't decide what order to satisfy premises in and relies on the order that users specified.

Worse, inductively defined specifications are often deeply nested. As a result, testing an arbitrary theorem requires that a user make such decisions about not only the inductive relations that appear in a theorem, but about all of their dependencies as well. All of this user effort and expertise is required because of two core inter-dependent issues: there exists no way of estimating the effectiveness of a particular derived generator for an inductive relation; and there exists no way of estimating the effectiveness of a particular sequence of such generators.

That is precisely the core contribution of this paper:

(1) We present a novel algorithm for ranking all possible ways of generating precondition-satisfying data to test a theorem specified using inductively defined relations. We frame our algorithm as an optimization problem over a notion of "density" that approximates the efficacy of the generator that an ordering would give rise to. As theorem statements and constructor types are similar, this algorithm also improves upon the generation of data satisfying the inductive relations themselves.
(2) We implement our approach on top of the QuickChick framework for Rocq, providing an effective and fully automatic `quickchick` tactic to test the current goal, which can derive generators for all necessary dependencies in an optimal way, greatly lowering the cost of testing for aspiring proof engineers.

(3) We thoroughly evaluate our approach:
- We show how QuickChick can now be used in a real world case study, developing an inductive encoding of the type system from Cedar, the authorization policy language used at AWS, and differentially testing it against the actual Cedar CLI, adjusting the inductive encoding to increase conformance.
- We demonstrate the efficiency of our approach using ETNA[36], a property-based testing evaluation platform, to compare our approach against previous automatic and user-tuned approaches from the literature.
- Using ETNA, we also showcase how sensitive QuickChick-derived generators are to the order of hypotheses in inductives, and how our approach is fully robust to such variations.
- We demonstrate the broader applicability of our implementation, selecting several theorems from the second volume of the Software Foundations series of online textbooks [33], and showing that our `quickchick` tactic can be used to effectively test them.

2 Background, by Example

We begin by diving deeper into the way QuickChick generates test inputs that are constrained by inductive relations. We showcase what QuickChick does well and what it falls short on, and we highlight how its shortcomings can be improved. We use the binary search tree development of the latest QuickChick paper on merging inductive relations [34] as our running example. The following code specifies what it means to be a valid BST using a `bst` inductive relation parameterized by a lower and an upper bound (`lo` and `hi`) on the keys of the tree:

```
Inductive tree :=
| Leaf : tree
| Node  : nat -> tree -> tree -> tree.

Inductive bst : nat -> nat -> tree -> Prop :=
| bst_Leaf lo hi : bst lo hi Leaf
| bst_Node lo x hi l r :
    between lo x hi -> bst lo x l -> bst x hi r ->
    bst lo hi (Node x l r).
```

A `Leaf` is always a valid binary search tree; a `Node x l r` is valid if its key `x` is `between` the `lo` and `hi` bounds, while its left and right subtrees are also valid with appropriate bounds.

In order to generate valid binary search trees, QuickChick requires users to specify a *mode* that determines which indexes of the relation should be inputs and which should be outputs. For example, to specify that the bounds are an input to the generation and the trees an output, users would issue the following command, where the arguments explicitly bound by a lambda are outputs, and everything else is assumed to be an input:

```
Derive Generator for (fun t => bst n m t).
```

For the rest of the paper, we will use a lighter-weight syntax to specify such modes, marking each argument with an input i or output o superscript as follows: `bst` $\texttt{n}^i$ $\texttt{m}^i$ $\texttt{t}^o$

When invoked, QuickChick creates a function with $\texttt{n}^i$ and $\texttt{m}^i$ as inputs, and whose output is a generator for `trees`. The generator is allowed to fail—i.e. it returns an option, since, in the general case, an inductive relation might not be inhabited. If the generator is recursive (as it would be for `bst`), it uses a fuel parameter to ensure termination. The skeleton of the function produced for the specified mode is shown in Figure 3. All QuickChick derived generators follow the same structure: after some fuel handling at the top level, the main body consists of a list of "handlers",

```
Fixpoint genBst ..fuel/size.. n^i m^i : G (option tree) :=
... (* fuel/size handling *) ...
    backtrack [ ... (* each constructor gives rise to a similar handler: *)
                match n^i, m^i with
                | lo, hi => ret (Some Leaf)
                | _ , _  => ret None
                end, ... ] ...
```

Fig. 3. Skeleton of a binary search tree generator, showing the handler for `Leaf`s.

one for each constructor of the inductive relation. Each handler attempts to produce a `tree` that satisfies its corresponding constructor, and a `backtrack` combinator samples handlers (without replacement) until one succeeds in producing such a tree. The shape of a constructor's conclusion determines how some of the constructor's variables are instantiated via pattern matching: the inputs to the generator are matched against the expressions found at corresponding positions of the constructor's conclusion (such as n^i against `lo` and m^i against `hi` in the `bst` example). Similarly, the output positions of the relation determine the shape of the generated data. For instance, the output position of `bst` is its third argument, and we see that the handler returns the third argument of the `bst_Leaf` constructor's conclusion, `Leaf` (wrapped in an `optional` type to allow for failure).

With the constructor's conclusion providing the first and last steps of the handler, the hypotheses form its core. The `bst_Leaf` constructor is not constrained by any hypotheses, so let's consider the more complicated handler for the more complicated `bst_Node`, which is shown on the right side of Figure 4. The hypotheses of `bst_Node` are satisfied in order, starting with `between lo x hi`. Each hypothesis is either used to generate the variables it constrains such that it is satisfied, or if all of its variables have been instantiated already, it is checked by a partial decision procedure. In the case of `bst_Node`, no checks are needed, as each hypothesis can instantiate one variable.

Crucially, this lack of checks is precisely what makes generation of binary search trees efficient. Unfortunately, this ideal generation is very easy to miss, one need only order `bst_Node`'s hypotheses slightly differently, placing `between lo x hi` after one or both of the `bst` constraints instead of being first, to introduce disastrous checks.

```
lo ⟵ n^i                          match n^i, m^i with
hi ⟵ m^i                          | lo, hi =>
x ⟵ between lo x^o hi               x <- arbitrarySizedST (fun x => between lo x hi);
l ⟵ bst lo x l^o                    l <- genBst lo x;
r ⟵ bst x hi r^o                    r <- genBst x hi;
return (Node x l r)                 ret (Node x l r)
                                  | _ , _  => ret None
                                  end
```

Fig. 4. QuickChick handler for the `bst_Node` constructor (right) and corresponding schedule notation (left).

2.1 Schedules

To discuss the effect of these checks without depicting a handler using concrete generator syntax every time, we introduce the notion of *schedules*: sequences of generators and checks that are in one-to-one correspondence with a handler. An example schedule for the `bst_Node` handler appears on the left hand side of Figure 4, which closely mirrors the actual generator code to its right but abstracts away from concrete pattern matching syntax or particular generator names.

Armed with this cleaner syntax for schedules, we can compare what handlers QuickChick will derive for different orderings of the `bst_Node` constructor. Consider the following ordering where the hypothesis for the left subtree appears before the constraint on the key—we will refer to this order as LBR (for Left-Between-Right).

```
bst_Node lo x hi l r :
    bst lo x l -> between lo x hi -> bst x hi r ->
    bst lo hi (Node x l r).
```

The schedule QuickChick will produce when presented with this variant of the `bst_Node` constructor will processes the left subtree's hypothesis first, and therefore has to instantiate both of its uninstantiated variables, `x` and `l`. QuickChick's implementation is tuned to favor recursive calls (as that prioritizes user preferences), which would require both `lo` and `x` to be instantiated before such a recursive call can be made; as such, `x` is instantiated by generating an arbitrary `nat` to use as the higher bound, as shown in Figure 5 on the left. Such a choice is disastrous to the effectiveness of generation in two ways: firstly, since `x` is generated arbitrarily, there is little chance that it will fall between `lo` and `hi`, and therefore that check will likely fail; secondly, and more subtly, when the left subtree is being generated using `lo` and `x` as its bounds, since `x` is not unlikely to be lower than `lo`, the only possible valid tree that can be generated is a `Leaf`, which skews the distribution of generated trees towards extremely shallow ones.

```
lo ← n^i                  lo ← n^i                  lo ← n^i
hi ← m^i                  hi ← m^i                  hi ← m^i
x ← nat                   x ← nat                   x ← nat
l ← bst lo x l^o          r ← bst x hi r^o          l ← bst lo x l^o
check (between lo x hi)   check (between lo x hi)   r ← bst x hi r^o
r ← bst x hi r^o          l ← bst lo x l^o          check (between lo x hi)
return (Node x l r)       return (Node x l r)       return (Node x l r)
```

Fig. 5. QuickChick schedules for bad orderings LBR, RBL, and LRB.

Worse, as Figure 5 shows, similar problems manifest in all but two of all possible `bst_Node` hypotheses orderings. In fact, as we will see later in the paper (Section 5), this lack of *robustness* to the order of hypotheses can be at times very pronounced. And, as discussed in the introduction, it stems from pragmatic design decisions behind QuickChick's derivation algorithm: What should the schedule for a given handler be? Lacking a way to distinguish between the exponentially many options, and to quickly estimate whether a particular mode of generation for a hypotheses would be effective, QuickChick falls back to user expertise. In this paper, we solve this problem: we use partial order reduction to avoid checking equivalent schedules; we perform a greedy preliminary branch-and-bound search to prune the search space to those schedules which minimize checks; and we holistically analyze these remaining schedules, introducing a notion of density that approximates how effective a schedule is to guide our search for the very best schedule.

3 Overview, by Example

In this section we will sketch the insights behind our approach, using the same examples that we've discussed so far. In the next section, we will describe our approach precisely and in detail.

3.1 Taming the Exponential Search Space

Recognizing that there are as many schedules for handling a relation's constructor as there are orderings of its hypotheses, it would be extremely expensive to exhaustively explore the search space in an effort to find the best schedule.

3.1.1 Observation 1: Avoid Arbitrary Generation. Let's start by considering again how to best handle a constraint of the form `bst lo x l`, when `x` and `l` are unconstrained, as is required for the `bst_Node` constructor discussed in the previous section. QuickChick can either arbitrarily generate `x` and recursively call the `bst` generator to create the left subtree `l`, it can arbitrarily generate the tree `l` and use a generator for the `between` relation—if such a generator is available, or it can generate both `x` and `l` at the same time—again, only if such a generator is already available. QuickChick heuristically prioritizes recursive calls, and so opts for the former.

However, in this and many other cases, such a choice is suboptimal: the generation of `x` will not be taking into account any of the constraints that are imposed by the `bst` relation. An arbitrarily generated `x`, for instance, might be less than the lower bound `lo`, therefore limiting `l` to a trivial `Leaf`. For other relations, an arbitrarily generated variable may have a value that is totally incompatible with a relation it is used as an input to, causing generation to fail altogether. Instead, we can and will opt to generate both `x` and `l` at the same time. In fact, that's a general guiding principle behind our approach: *it is always best to generate as many variables from a hypothesis as possible.* In other words, when considering a particular ordering of hypotheses, for each hypothesis, we will opt for the mode that generates as many currently uninstantiated variables as possible.

Implementation wise, such a choice comes at an (engineering) price: it means that we might need to create multiple mutually recursive generators and checkers to account for inter-dependent modes. It also means that an ordering of hypotheses uniquely determines a schedule, as depicted in Algorithm 1 in Appendix A . Thus, the size of the search space is reduced down to a (still-egregious) factorial in the number of hypotheses.

3.1.2 Observation 2: Avoid Equivalent Schedules. To further restrict the search space of possible schedules, we can borrow a trick from the concurrency literature: *partial order reduction* [19]. Consider the schedules that arise from the BLR and BRL orders in `bst_Node`, that is, when the `between` constraint is handled first, but the order in which we process the constraints on the left and right subtrees varies. These schedules are shown in Figure 6. Intuitively, after selecting the value of a `Node` in a tree, whether we generate the left or right subtree first makes little difference—they are independent. In general, whenever two hypotheses in a schedule constrain disjoint sets of variables (ignoring variables instantiated earlier in the schedule), they can be freely commuted without affecting generator performance. Therefore, we need only consider one representative from each class of commutation-equivalent schedules, radically reducing the number of schedules that need to be considered in practice.

```
lo ← n^i                          lo ← n^i
hi ← m^i                          hi ← m^i
x ← between lo x^o hi             x ← between lo x^o hi
l ← bst lo x l^o                  r ← bst x hi r^o
r ← bst x hi r^o                  l ← bst lo x l^o
return (Node x l r)               return (Node x l r)
```

Fig. 6. Reordered Schedules

3.1.3 Observation 3: Avoid Checks. After applying partial order reduction to all possible `bst_Node` schedules, we are left with three schedules to consider without loss of generality: `LBR` (equivalent to `LRB`), `BLR` (equivalent to `BRL`), and `RLB` (equivalent to `RBL`). These schedules are depicted in Figure 7.

How should we distinguish between these three schedules? There is a coarse, but very effective filter: two of the schedules contain restrictive checks. In general, *the fewer checks that are performed,*

```
lo ← n^i                  lo ← n^i                  lo ← n^i
hi ← m^i                  hi ← m^i                  hi ← m^i
(x,l) ← bst lo x^o l^o    x ← between lo x^o hi     (x,r) ← bst x^o hi r^o
check (between lo x hi)   l ← bst lo x l^o          check (between lo x hi)
r ← bst x hi r^o          r ← bst x hi r^o          l ← bst lo x l^o
return (Node x l r)       return (Node x l r)       return (Node x l r)
```

Fig. 7. Schedule candidates after partial order reduction

the lower the potential for a generator to fail or devolve to rejection sampling. In the `bst` example, that's actually enough to select the best schedule. However, for arbitrary relations, there is a crucial part of scheduling we haven't yet considered: the quality of the generators it depends on.

3.2 Estimating Effectiveness

To provide intuition on estimating the quality of generators involved in a schedule, we return to the more complex example of the introduction, preservation for a simply typed lambda calculus: If a well-typed expression takes a step, then the resulting expression shares its type. In Rocq notation, using two inductive relations `typing` and ⇓.

```
Theorem preservation: forall Γ e e' τ, typing Γ e τ -> e ⇓ e' -> typing Γ e' τ.
```

Given the observations of the previous section, to instantiate the variables of this theorem we only need to consider two options: generating an environment Γ, an expression e, and a type τ for which `typing` holds, and then generating an expression e' such that $e \Downarrow e'$; or generating two expressions e and e' where the former steps to the latter, and then an environment Γ and a type τ for which `typing` holds. The corresponding schedules appear in Figure 8.

```
(Γ, e, τ) ← typing Γ^o e^o τ^o      (e,e') ← e^o ⇓ e'^o
e' ← e ⇓ e'^o                       (Γ, τ) ← typing Γ^o e τ^o
check (typing Γ e' τ)               check (typing Γ e' τ)
```

Fig. 8. Schedule candidates for preservation

Using just the observations of the previous section, the schedules appear identical: neither performs unnecessary checks (the only check is the conclusion of the theorem). How, then, can we select which schedule is optimal?

The Complexity. To answer that, we need to figure out how effective generators for the inductive relations involved are. Each such generator can select a handler for *any* of its corresponding inductive's constructors to satisfy. Therefore, we need a way of *aggregating* the effectiveness of all such handlers. To figure how effective each handler is, we need to recursively solve the top-level scheduling problem for each constructor's type, selecting the best possible schedule for that constructor and estimating it's effectiveness. Moreover, selecting the best such schedule might recursively depend on yet another inductive relation (or even a different mode for one we've encountered) whose effectiveness we'll need to analyze, repeating this process as many times as the nesting of the relations involved dictates.

Introducing Density. The first step to taming this complexity is to pin down "effectiveness" itself, which dictates the granularity of the rest of the analysis. To that end, we will introduce a coarse (but not too coarse) classification of each step's expected performance, which we will call *density*:

```
type density = Checking | Backtracking | Partial | Total
```

Each density level roughly corresponds to different failure modes of generators:

(1) A check or a generator that internally performs checks is classified as Checking, the worst possible choice: independently generated values are very unlikely to satisfy a relation.
(2) A generator that does not perform checks but requires pattern matching on the output of to make sure its result has a certain shape is classified as Backtracking: not as bad as checking, but less than ideal.
(3) A generator which is partially defined over its input space but will generally succeed in generating an output is classified as Partial. Most good relation-satisfying generators fall in this category.
(4) A generator which does not perform checks, and always succeeds in generating an output (such as an `arbitrary` type-based instantiation of a variable) is classified as Total, and is ideal (so long using it doesn't cause more checks later in the schedule).

Schedule steps that involve a relation inside a schedule are as good as the corresponding generator. We will be approximating the density recursive relation-based generator calls as `Partial`; for non-recursive relation-based generators, we will need to estimate their overall density.

Densities, Schedules, and Handlers. After we have annotated each step of a schedule with a density, we need to estimate the effectiveness of the schedule as a whole. So far, we have been using the "least number of checks" as our guiding principle—avoiding checking constraints that were not taken into account during generation. Generalizing that principle to densities, the overall effectiveness of a schedule is simply the worst density of any of its steps, where the ordering is the order in which they were introduced earlier. For concreteness, let's take a closer look at a particular constructor of typing (application) for the two modes that we would need to analyze in our preservation example: when all three arguments are outputs (as on the left of Figure 8), and when the expression is an input (as on the right).

```
TyApp : forall Γ e1 e2 τ1 τ2, typing Γ e2 τ1 -> typing Γ e1 (TFun τ1 τ2) ->
          typing Γ (App e1 e2) τ2
```

typing Γ^o e^o τ^o

```
P: (Γ, e2, τ1) ← typing Γ^o e2^o τ1^o
T: τ2 ← arbitrary
?: e1 ← typing Γ e1^o (TFun τ1 τ2)
return (Γ, App e1 e2, τ2)
```

```
B: (Γ, e2, TFun τ1 τ2) ← typing Γ^o e2^o τ^o
?: e2 ← typing Γ e2^o τ1
return (Γ, App e1 e2, τ2)
```

typing Γ^o e^i τ^o

```
App e1 e2 ← e^i
P: (Γ, τ1) ← typing Γ^o e2 τ1^o
T: τ2 ← arbitrary
C: check (typing Γ e1 (TFun τ1 τ2))
return (Γ, τ2)
```

```
App e1 e2 ← e^i
B: (Γ, TFun τ1 τ2) ← typing Γ^o e2 τ^o
C: check (typing Γ e2 τ1)
return (Γ, App e1 e2, τ2)
```

Fig. 9. Schedules for application typing handler, two for each top-level mode

The schedules for the $\texttt{typing}^{oio}$ mode are both Checking: their worst schedule step is Checking. The schedules for the $\texttt{typing}^{ooo}$ mode both depend on a different mode of typing, where only the expression is an output ($\texttt{typing}^{ioi}$). It turns out that mode will be Partial (after applying this analysis recursively), and therefore the left schedule is Partial; similarly, the right schedule is Backtracking, as the first type generation needs to be an arrow. As such, the left one is the best schedule.

If both schedules had the same density, we would break ties in favor of *least variable dependencies:* since variables from separate steps are generated independently, using them as inputs to another generator might lead to problems. For example, generating the type in advance is a thorn in STLC generation literature [30].

Aggregating Handlers. After we have selected the best schedule for each constructor handler, we need to estimate the effectiveness of the top-level generator that picks amongst them. Since each constructor can impose different constraints on its input variables with pattern matching, it's not as simple as picking "the best handler density". Looking at the bottom row of Figure 9, each of the schedules begins by pattern matching the input expression against an App pattern. In general, that will be the case for most handlers: the shape of the conclusion will induce a pattern match that restricts the shape of the inputs for which the handler is applicable. Moreover, handlers for inputs that cover different patterns are entirely independent—they don't apply to the same cases. The final piece of intuition is that we need to combine schedule densities on a *per pattern* basis. The best efficiency for each pattern then, is the best among the handlers that cover it.

4 Scheduling, Formally

In this section, we formally describe the approach that we sketched so far via examples. We begin with precisely introducing the problem addressed, followed by the definition of pattern tries—the data structure we are using to group handlers for individual constructors. We provide an abstraction that allows for expressing diverse schedule analyses as ordered monoids. We then express both density and variable dependency as instances of this abstraction, with the overall algorithm then arising naturally as the product of the two.

Problem Definition. Similar to prior work on QuickChick, we target inductive relations, potentially parameterized over a number of type parameters A_i, that range over a number of simply-typed, first-order indices of type T_i. Every relation P is defined via a number of constructors C_i, each of which begins by universally quantifying variables, followed by any number of inductively defined preconditions, and finally a conclusion of P:

$$\texttt{Inductive}\ P\ (A_1 \ldots : \texttt{Type})\ :\ T_1 \to \ldots \to T_n \to \texttt{Prop}\ :=$$
$$\mid\ C_1 : \forall x_1 \ldots, (Q_1\ e_{11} \ldots) \to \ldots \to (P\ e_1 \ldots)$$
$$\mid\ \ldots$$

This encoding of relations follows standard Rocq practice where dependently typed relations impose constraints over simply-typed data. Syntactically different but equivalent forms (such as reordering quantifications) can be easily handled via lightweight rewriting as described in the literature [31].

Each generator for data satisfying such an inductive relation has a *mode*: a classification of the various indices T_j as either inputs, i, or outputs, o. The resulting mapping, $m : \{j \in \{1..n\}\} \to \{i, o\}$ is a valid generation *mode* as long as at least one of the indices is an output.

We target theorems of similar form to the types of inductive constructors, requiring them to be written in prenex normal form. Unlike a constructor, the conclusion of a theorem is an arbitrary relation to be checked.

$$\texttt{Theorem}\ T\ :\ \forall x_1 \ldots, (Q_1\ e_{11} \ldots) \to \ldots \to (P\ e_1 \ldots)$$

Given such a theorem T, our goal is to *schedule* the generation of its variables, taking into account its preconditions Q_i, with the following constraints:

- Every variable needs to be generated.
- Each variable can only be generated using at most one Q_i.
- Each Q_i used for generation will instantiate as many uninstantiated variables as possible.
- Each precondition not used for generation has to be checked.

A schedule, then, is a sequence of schedule steps each of which assigns the result of a generator to a variable, a pattern match of a variable against a pattern, or a check. Compared to the syntax in the previous section, we decouple the pattern matching from generation to simplify definitions later. Generators $\hat{\tau}$ can either be type-based, according to some type τ, or inductive relation based, according to some inductive P some of whose arguments $e^{\square}$ are specified and some of which are to be generated; we will denote the latter using a $\square$. Patterns are standard: either variables, wildcards, or constructors recursively applied to patterns. The full grammar is shown in Figure 10.

$$
\begin{aligned}
s &:= x \leftarrow \hat{\tau} \mid p \leftarrow x \mid (P\ \overline{e})? \\
\hat{\tau} &:= \tau \mid P\ \overline{e^{\square}} \\
e^{\square} &:= e \mid \square \\
p &:= x \mid \_ \mid C\ \overline{p}
\end{aligned}
$$

Fig. 10. Schedule Step Grammar

Armed with these definitions, the problem we are trying to solve in this paper is *theorem scheduling*: Given a theorem T, pick the valid schedule that leads to the most efficient generation.

Partial Order Reduction

To drastically reduce the schedules that we need to consider, the first step is to carry out a partial order reduction leveraging the commutativity of independent hypotheses, as discussed in the previous section. Let $\text{sched}'(C, m)$ be the set of valid schedules for a constructor C under mode m, where a schedule is valid if it satisfies the four constraints above, and consider two adjacent hypotheses h and j in a permutation π, where h appears earlier than j, and which constrain variable sets v_h and v_j. Let v_{π_h} be the set of variables constrained by any of the hypotheses in the prefix of π before h. Then we can define a commutativity predicate on hypotheses $\text{canSwap}(v_{\pi_h}, h, j)$ and a swapping function as such:

$$
\begin{aligned}
&\text{canSwap}(\pi, h, j) := v_h \setminus v_{\pi_h} \ \cap v_j \setminus v_{\pi_h} = \emptyset \\
&\text{swap}(\pi, h, j) := \pi[h \leftrightarrow j]
\end{aligned}
$$

We can then define a $\text{commutesStep}(\pi, \pi')$ relation relating any two permutations that are one swap away:

$$
(\pi, \pi') \in \text{commutesStep} \iff \exists h, j \in \pi, \text{canSwap}(\pi, h, j) \ \wedge \text{swap}(\pi, h, j) = \pi'
$$

We define our equivalence relation of commuting schedules which commute to each other in any number of steps by taking the reflexive transitive closure of commutesStep and composing it with the bijection between orders and schedules:

$$
\begin{aligned}
&\text{commutesOrder} := \text{commutesStep}^* \\
&\text{commutes} := \{(s, s') \in \text{sched}'(C, m)^2 \mid \text{commutesOrder}(\text{orderToSched}(s), \text{orderToSched}(s'))\}
\end{aligned}
$$

Finally, we define our reduced search space of schedules as the quotient,

$$
\text{sched}(C, m) := \text{sched}'(C, m) \,/\text{commutes}
$$

Implementation Note: Greedy Branch and Bound Search. Although commutativity substantially reduces the schedule space, exhaustively enumerating it can still be prohibitively expensive. We therefore represent the space approximately as a lazily evaluated tree whose nodes are incomplete schedules. The children of a node are obtained by inserting the next hypothesis into each position

that is distinct according to the commutativity relation. The leaves thus form a reduced schedule space, although some equivalent schedules may remain because equivalence between leaves in separate branches cannot be enforced while the tree is consumed lazily.

Performing the recursive density analysis at every leaf would itself be expensive. Instead, we assign each incomplete schedule a cheaper score: the number of explicit checks already required by its partial ordering. This score is monotone along a branch, since extending an incomplete schedule cannot eliminate a check that has already been forced. Consequently, once the search has found a complete schedule with check count c, it can prune any node whose current check count is worse than c. Our depth-first search greedily orders the children of each node by their check counts and explores the most promising child first. This ordering tends to find a strong complete schedule early, establishing a bound that enables more aggressive pruning of the remaining branches.

Pattern Tries

The second component of our approach is pattern tries. As discussed in the previous section, our analysis needs to be done over the set of patterns that appear in the conclusions of the constructors of a relation. In the STLC example, we only saw a simple pattern (`App _ _`), but in practice, the patterns that will appear in conclusions of constructors can be nested and overlapping.

Given an inductive P and a mode m, we define $pt(P, m)$ to be the mapping from patterns which match P's inputs to the constructors of P that cover it. Implementation-wise, as we process each constructor of an inductive relation, its conclusion might refer to a previously unseen pattern, or it might further refine one already in our mapping. This naturally gives rise to an efficient trie-based representation of this mapping, with patterns replacing the usual strings as keys of the trie. Such generalizations of trie keys to compositions of constructors or functions have been explored in different settings in the literature [9, 21].

Efficiency Score

To evaluate the efficiency of a schedule for a given theorem T, we will define two efficiency score components: both density and variable dependency fit into the same abstraction. The two primary operations that need to be supported by efficiency scores are *comparisons* (to select the best/worst amongst a group of options) and *aggregations* (to combine the score of individual steps in a schedule). That is, a score is simply an ordered monoid M, together with a function $\text{score}_{sc} : s \to m \to M$ that for a given mode m assigns to each schedule step s an element of the monoid. For our purposes, the smaller the score, the better.

Schedule Score

Let $\sigma = [s_1, \ldots, s_k]$ be a schedule and m the mode we're trying to score. Then the score of σ is defined as the sum of the scores over all the steps of the schedule using the monoid's binary operation.

$$\text{score}_{\sigma}(\sigma) := \sum_{i=1}^{k} \text{score}_{sc}(s_i, m)$$

As such we can model both of our guiding principles of the previous section: the density of a schedule is that of its worst schedule step (i.e. binary operation is maximum), and breaking ties using the schedule with the least variable dependencies (binary operation is summation).

Constructor Score

We can use schedule scores to define scores of constructors—the score of a constructor for a given mode m is the best, and therefore, minimal, score amongst all of its schedules:

$$\text{score}_c(C, m) := \min_{\sigma \in \text{sched}(C,m)} \text{score}_\sigma(\sigma)$$

This additionally defines the score of a theorem, where the mode assigns all of the theorem's variables to output.

Inductive Relation Score

The score of an inductive relation is calculated per covered input pattern. For each covered pattern, we consider the best constructor score among its set of covering constructors (as we can use the efficient constructors to generate outputs given that input pattern); we then assign to the inductive the worst (ergo maximum) score of any of its covered input patterns.

$$\text{score}_{ind}(P, m) := \max_{p \in pt(P,m)} \left(\min_{c \in pt(P,m)(p)} \text{score}_c(c, m) \right)$$

Intuitively, a poorly scoring input pattern implies that any inputs that match that pattern will struggle to generate corresponding relation-satisfying outputs. This issue does not disappear even if another input pattern scores exceptionally well, as the generator would still perform poorly on the low scoring input pattern.

Density

We define density as the totally ordered monoid over the four-element set containing:

$$\texttt{Total} \leq \texttt{Partial} \leq \texttt{Backtracking} \leq \texttt{Checking}$$

It's binary operation is `max`.

The density of a schedule step s for a mode m is given on the left of Figure 11: unconstrained generation is Total, a checking step is Checking, pattern matching against the result of a generation is Backtracking, recursive calls to the same mode are approximated as Partial (discussed below in *Approximation*), and invoking a generator for a different relation has the same density score as the density of that relation (written as $|_d$ to project the density component from the score).

Variable Dependency

The monoid of variable dependencies is the standard monoid of natural numbers with addition. The definition of the variable dependency of a schedule step is parameterized by the set I of variables that are fixed by the mode. Checks and generators calls induce a number of dependencies equal to the number of distinct free variables involved, excluding the ones bound at the toplevel as inputs, as shown on the right of Figure 11.

$$\begin{aligned}
&\text{density } m\ (x \leftarrow \tau) = \text{Total} && \text{dep}_I\ m\ (x \leftarrow \tau) = 0 \\
&\text{density } m\ (P\ \overline{e})? \ = \text{Checking} && \text{dep}_I\ m\ (P\ \overline{e})? \ = \#(\text{vars}(\overline{e}) \setminus I) \\
&\text{density } m\ (p \leftarrow x) = \text{Backtracking} && \text{dep}_I\ m\ (p \leftarrow x) = 0 \\
&\text{density } m\ (x \leftarrow P\ \overline{e^{\square}}) \mid \text{rec?}\ (m, P\ \overline{e^{\square}}) = \text{Partial} && \text{dep}_I\ m\ (x \leftarrow P\ \overline{e^{\square}}) \ = \#(\text{vars}(\overline{e}) \setminus I) \\
&\qquad\qquad \mid \text{otherwise} = \text{score}_{ind}(P, m)|_d &&
\end{aligned}$$

Fig. 11. Definition of density (left) and variable dependency (right).

Approximation: Recursive Calls as Partial. The exact density of recursive relation-based generator calls can only be computed after all schedules have been selected for the generator and the generators it depends on. But determining the schedules for each constructor of the relation requires assigning a density to recursive calls. Instead of attempting to tie this deeply recursive knot, we label recursive calls optimistically as Partial, the best density the vast majority of relations can achieve (unless they are truly Total, defined on every possible input and not performing backtracking or checks).

This approximation is based on the fact that if a generator's density is truly Checking, that density characterization is not attributed to its recursive calls, but to the explicit Checks or Checking density generator calls that occur as steps in the generator's schedules. The assumption underlying our approximation then is that even if we locally overestimate the density of a recursive call, the Checking density of other parts of the generator will most of the time emerge globally when the densities for each input shape are aggregated.

This assumption has worked well for all the case studies we have considered, but it's still a heuristic, and we can adversarially construct counterexamples that directly target the assumption above. For instance, consider the following inductive that describes primes in a roundabout way:

```
Inductive Foo : Nat -> Prop :=
| A : Foo n -> Foo n
| B : isPrime n = true -> Foo n.
```

Any proof of `Foo` will be an arbitrary nesting of recursive calls to `A` followed by a final call to `B`, where an unlikely check (whether `n` is a prime number) is called. Our density analysis for outputting the single Nat argument would assign `A` to be Partial because it is just a recursive call, and it would assign `B` to be Checking, because `isPrime` is a function and cannot be automatically inverted, so its truth has to be checked. Since there are no inputs, there is only one input shape in the Pattern Trie, covered by both `A` and `B`. The density of an input shape is the best density of its covering constructors, so the input shape would take the better density, Partial, and therefore the inductive would be Partial.

While it is true that at every step, the generator can successfully call the handler for `A`, it is just delaying the inevitable base case check from `B`. Remedying this limitation may require a more complicated information flow analysis to determine what domain of inputs are given to recursive hypotheses, not just to the top level generator, but that remains future work. In practice, while such examples can be constructed adversarially, we have not observed this limitation in our experiments.

5 Evaluation

In this section, we evaluate many aspects of our approach, from its effectiveness to its applicability, by answering the following five research questions:

RQ1 Is our approach useful for testing real-life systems?
RQ2 How does our approach compare to the state-of-the-art in terms of bug-finding performance?
RQ3 Is our approach robust to reorderings of hypotheses in inductively defined specifications?
RQ4 Is our approach applicable in a diverse range of scenarios?
RQ5 How does each metric component contribute to selecting the final schedule?

5.1 Cedar Differential Testing

To demonstrate how our approach can help find real bugs in practical systems, we turn to a recent example of effective formal verification in industry: the Cedar authorization policy language [10]. The Cedar team used a technique they dubbed *verification guided development* (VGD), which combines formal verification and testing to validate a real software system. Cedar has an efficient Rust implementation, but the team also maintains a formally verified model in Lean.

The implementation and model are kept in sync using property based testing, leveraging a handwritten random generator of well-typed Cedar expressions and policies that they use to look for mismatches between the interpreter and the model. The well-typed expression generator, like most generators satisfying complex preconditions, was very expensive to construct in time and effort, taking at least 6 months to write and spanning approximately 5000 lines of code[1]. Such a large generator is also bound to have bugs of its own, and it is incredibly difficult to maintain as the system evolves.

We set out to discover if we could obtain similar benefits automatically, using our fully automatic generation support. To that end, we developed an inductively defined Rocq model of the Cedar typechecker, and brought it up to speed with Cedar's Rust implementation using solely the derived generators for debugging. The only non-derived aspect of testing was a pretty printer that allowed us to send generated Cedar expressions and schema as strings to be parsed by Cedar's Rust implementation. During this testing process, we found a number of bugs in the translation of typing from Lean functions to Rocq inductive relations, as well as a bug in Cedar's Rust implementation.

In more detail, we specified the following theorem, which states that given a well-formed entity schema, an environment valid with respect to that schema, a well-formed Cedar type, and a Cedar expression that is well-typed under that environment with the given type, then after pretty-printing and parsing by Rust Cedar, the expression (embedded in a trivial policy) will be well-typed.

```
Theorem prop_no_type_error :
  forall ets acts rs e es ty expr,
    WfSchema [MkName "EntityType1" []; MkName "EntityType2" []] (MkSchema ets acts) ->
    ActionSchemaToRequestTypes acts [] rs ->
    SchemaToEnvironments (MkSchema ets acts) rs (cons e es) ->
    WfCedarType [MkName "EntityType1" []; MkName "EntityType2" []] ty ->
    HasType (somepaths []) e expr ty ->
    no_type_error_with_env (MkSchema ets acts) e expr = true.
Proof. quickchick.
```

Within a second of invoking the `quickchick` tactic, our approach derived an efficient generator which generates a well-formed schema, an environment derived from the schema, a well-formed Cedar type, and finally a Cedar expression `expr` with the generated type under the generated environment and schema. The derived generator then ran `no_type_error_with_env`, which pretty prints the schema, environment, and expression and asked Rust Cedar to parse and type-check it, asserted that it should return true.

As we uncovered counterexamples, we updated the pretty printer and typing inductive relationin Rocq and invoked the `quickchick` tactic which derived a fresh generator for well typed terms that iteratively became closer in function to the Rust implementation. In other words, we never needed to maintain a separate hand-tuned generator for well-typed terms throughout the process.

Rust Implementation Bug. Our tests discovered a difference of behavior between the Rust Cedar implementation and its Lean formalization. In particular, we found that in conditional expression branches that are allowed to be ill-typed, attempting to reference an undefined entity still raises a type error, despite being in an untyped section of an expression. This bug was already known to the Cedar team when we reported it, and it has since been fixed. [2]

Translation Bugs. We also found many bugs with our own implementation of the Cedar specification and our pretty printer. These errors turned out not to be bugs in the original implementation, as that one has been heavily tested already, However, these bugs still demonstrate the way a

[1]https://github.com/cedar-policy/cedar-spec/blob/main/cedar-policy-generators
[2]https://github.com/cedar-policy/cedar-spec/pull/779

high-quality, automatically derived generator can explore behaviors deep in the software stack to find a wide range of potential mistakes. A description of these bugs can be found in Table 1.

The first set of bugs we encountered and quickly resolved were in the **pretty printer**, caused by structural differences in how Rocq and Rust store Cedar's AST. For instance, consider bug 7 concerning the pretty printer for record literals. Rocq's inductive relations often work best for generation and proof when the data they constrain can be built up iteratively. As a result, we internally represented record (and set) literals as linked lists with well-formedness constraints, built using Cedar expression constructors `recExprCons` and `recExprNil` for prepending and the empty record respectively. Without a separate recursive helper pretty printer solely for records, the original pretty printer for Cedar expressions displayed records and sets incorrectly which led to parse errors.

Other bugs were related to Cedar's peculiar **typing rules** for if expressions. Cedar's type system not only has type `Bool` but also subtypes `True` and `False` for Boolean expressions which can statically be resolved to either true or false. When an if expression's condition is known statically to be true or false, then only the branch that will provably be taken is required to be well-typed. The other branch can be arbitrary Cedar syntax. This, in conjunction with the peculiar representation of sets and records needing well-formedness constraints to be valid syntax in Rocq led to bugs 8 and 9, as the arbitrary Cedar syntax was not constrained by record or set literal well-formedness and therefore caused parse failures when pretty printed.

Finally, a third category of bugs stemmed from limitations of the **schema definition syntax** that were not encoded in Cedar's type system. For instance, a portion of the environment termed the "action" is, in the Rust implementation, hard-coded to be of type "Action", whereas the type system allows the action to be of any defined type. Additionally, the schema syntax does not support the `True` or `False` types, only `Bool`, but this is not a limitation of the typechecker. Other examples include the type system not encoding the parser-enforced requirement that schema definitions cannot have duplicate names.

5.2 Bug-Finding Performance

To further evaluate the performance of our approach, we turn to ETNA [36], a property-based testing evaluation platform that contains a series of workloads and generation strategies to compare against. In particular, ETNA workloads are comprised of a correct baseline, as well as a collection of mutants, i.e. bugs that can be injected in the baseline. ETNA also offers tools for analyzing and visualizing how effective a generation strategy is at uncovering those mutants: the more mutants found quicker, the better the strategy. For our evaluation, we use two ETNA workloads, binary search trees (BST) and simply typed lambda calculus (STLC), which contain useful strategies to compare against: a naive baseline (type-based properties and generators) and two state-of-the-art toplines (handwritten properties with handwritten or QuickChick-derived generators).

ETNA then measures how long each property takes to discover these injected counterexamples and creates "buckets" classifying their effectiveness: bugs found in less than 0.1 seconds, less than 1 second, less than 2 seconds, and so on. Each such "bucket" corresponds roughly to bugs found immediately, quickly, and so on in an interactive setting. The results are shown in Figure 12.

The BST case study considers a suite of 18 theorems specifying the intended BST behavior and 8 mutations, each relevant to some subset of theorems, resulting in 52 relevant combinations in total. The left of Figure 12 depicts the evaluation results for BST, with counterexamples stemming from all 52 mutation-theorem combinations were discovered in under a second by every strategy except the type-based approach. Our approach, labeled TheoremGenerator, finds all 52 counterexamples in under a tenth of a second, on par with the Bespoke and older QuickChick approach. Both the bespoke handwritten properties and the ones using derived generators rely on significant user

#	Component	Root Cause	Observed Effect	Evidence	Size	Tests	Discards	Time (s)
1	Rocq generation	Overly broad string generation leaked arbitrary strings into `EntityName` and `EntityUID` generation	Bogus entity/action namespaces such as `owner`, `readers`, or `Hicks` appeared in generated schemas and expressions, causing false mismatches with Cedar	rediscovered	2	1	0	0.003251
2	Rocq typing (`TAction`)	`action` was typed by the UID entity name instead of the fixed `Action` namespace	Type mismatch vs. Cedar on action attribute access; Rocq could type expressions that Cedar rejected because Cedar always treats `action` as type `Action`	rediscovered	2	181	30	0.123389
3	Rocq request/action schema generation	Action schema entries were not constrained to use the `Action` namespace	Generated request/action UIDs did not align with Cedar's action normalization, producing false request-validation/typechecking mismatches	rediscovered	2	1	0	0.002807
4	Rocq typing (`TRecCons`)	No uniqueness constraint on record keys	Duplicate-key records were accepted by Rocq but rejected by the Cedar parser	rediscovered	2	165	22	0.119812
5	Rocq typing (`THasAttrEntityReq`)	Returned `tt_` for required entity attributes, whereas Cedar returns `anyBool`	Unsound branch pruning in Rocq; Cedar still required branch compatibility	rediscovered	2	63	12	0.050407
6	Rocq schema well-formedness	Schema attributes allowed `tt_`/`ff_` booleans, although Cedar schemas only admit `Bool`	Serializer collapsed these to `Bool`, creating branch-typing mismatches with Cedar	rediscovered	2	418	90	0.310213
7	Pretty-printer (records as linked lists)	Records are encoded as linked lists in Rocq (`recExprCons`/`recExprNil`), but the printer flattened malformed tails as if they were record continuations	Incorrect concrete syntax, including structurally wrong or nested record renderings, leading to Cedar parse failures	rediscovered	3	574	193	1.077438
8	Pretty-printer (record tail handling)	When a record tail was not a proper record list, the fallback printer branch emitted partial text and separators anyway	Trailing commas and malformed record literals that Cedar rejected at parse time	rediscovered	3	574	193	1.077438
9	Pretty-printer (set tail handling)	Set expressions are list-encoded similarly; malformed tails were still pretty-printed as list continuations	Malformed set literals, including dangling separators, causing parse errors	rediscovered	2	11	4	0.011316
10	Policy-wrapper artifact (empty set literals)	Differential harness typechecked expressions by embedding them into a synthetic policy; Cedar policy validation rejects empty set literals in that context	False mismatches: Rocq-generated terms with empty sets failed in Cedar for policy-syntax reasons, not core typing differences	rediscovered	2	4	2	0.006255
11	Rust Cedar vs. Lean spec	Rust typechecker checked for undefined entity usage inside a conditional branch that should have been ignored under path-sensitive typing	Behavioral mismatch with the Lean formalization; bug confirmed by Cedar team and later fixed	rediscovered	2	24	4	0.020735
12	Rocq schema well-formedness	The type system did not encode the parser-enforced requirement that schema definitions have unique names	Schemas accepted by Rocq were rejected by the Cedar parser/validator as duplicate definitions	rediscovered	2	1	0	0.002737
13	Rocq typing (`TRecCons`)	Record literal fields were allowed to carry schema-style optionality	Rocq could assign a record literal a type with optional fields, but Cedar treats fields that appear in a record literal as required, causing branch-join mismatches	rediscovered	4	6187	1166	38.782679

Table 1. Bugs fixed during differential testing of the Rocq Cedar model against the Rust Cedar implementation, alongside time to rediscover when injected as mutants.

tuning: the bespoke properties choose both a size of tree to generate (depth 5) and bounds for the values of the BST (0 through 40); properties with QuickChick-derived generators select a different size (depth 10) and bounds (0 through 100) to suit its needs. Both also need to explicitly specify generation order and generation modes. Our approach can work out of the box, but also allows for seamlessly specifying a particular size if needed, and does not hard code tree element bounds—instead, it uses a generation mode for the BST relation that simultaneously constructs a valid BST and its bounds.

The STLC case study considers two theorems and ten mutations, but each mutation was relevant to both theorems, yielding a total of 20 mutation-theorem pairs. The two theorems assert the preservation property for a single step of evaluation and for forty steps respectively. The results

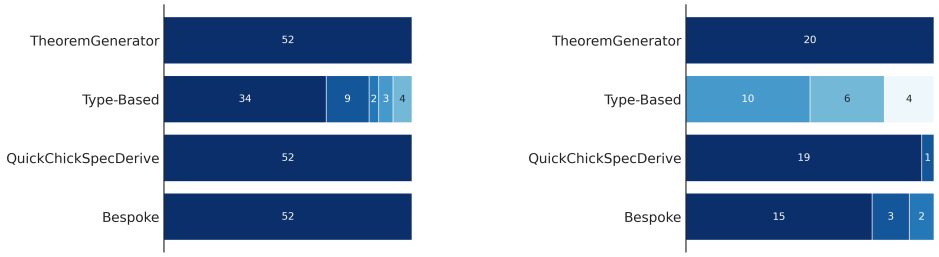

Fig. 12. ETNA buckets for the BST (left) and STLC (right) workloads. Color groupings indicate number of inject bugs found in under a time limit: the darkest bucket corresponds to bugs found, on average, in under 0.1s, the next to bugs found in under 1s, the next in under 2s, the next in under 5s, and the next in under 10s. The most pale bucket (only in the Type-Based generator for STLC) corresponds to bugs not found within the 10 second timeout. Timeouts are considered 10 seconds in the average.

(Figure 12) again demonstrate that our approach is on par with both handwritten and specification-derived generators, and that it is even (marginally) superior to both of them: it is the only generator which finds all 20 counterexamples in under a tenth of a second.

5.3 Robustness to Reordering

Using these same case studies, we will also demonstrate that our approach is robust to different hypotheses orderings, while at the same time showcasing that the current implementation of QuickChick is not. In this section, we compare the effectiveness of generators derived under different hypothesis orderings inductive relations in BST and STLC.

5.3.1 Reordering STLC. For STLC, we compare the results of testing a preservation theorem, which depends on generators for the `typing` relation, which determines well-typedness, and the `bigstep` relation, which determines how terms reduce. The well-typedness relation has only one constructor that has more than a single precondition, its App typing rule, which has two preconditions. The big-step relation also has one constructor with more than one precondition, its App evaluation rule, which has four preconditions. Both of them will be shown in Figures 14 and 15 below.

To test preservation, QuickChick requires the user to select generators for each of its variables, e, e', and τ, derive the selected generators (and those generators required by the selected generators), and compose them into a property test. Using QuickChick's API:

```
Definition preservation_prop :=
  sized (fun s => forAll arbitrary (fun t =>
  forAllMaybe (genST (fun e => typing ∅ e t)) (fun e =>
  forAllMaybe (genST (fun e' => bigstep e e')) (fun e' =>
  collect (size e) (typing ∅ e' t ?? 50))))).
```

Our approach instead only requires calling the `quickchick` tactic from within the proof of the theorem. The `quickchick` tactic, in less than a second, derives all required generators without user input, constructs the property test, tests the theorem, and reports that 10,000 tests succeeded.

For each reordering choice of the two constructors of `typing` and `bigstep`, we derived the generators using both QuickChick and our approach, and compared the speed and average number of constructors used in the STLC terms generated (not including type annotations, which can be arbitrarily large without increasing term complexity). The left side of Figure 13 displays the results for all 12 (6 choices for bigstep's App rule and 2 from `typing`'s App rule) ordering combinations for both QuickChick and our approach. In all but three of these combinations, QuickChick's property fails to terminate within 30 seconds.

Half of the orderings (those with typing order `t2`) fail due to order of the `typing` hypotheses: when the typing constraint for the functional argument appears before the argument one, Quickchick derives a very inefficient checker constraint on the applied function before the `typing` constraint on the argument in typing's App rule.

This ordering (Figure 14) causes QuickChick's property to hang because it leads to a very inefficient checker for its `typing` conclusion. The only unconstrained variable in `typing`'s App rule for a checker is that of the type of the argument, `tfrom`. Values for the argument type must be enumerated until one is found that makes the two preconditions true. Given that there is only a single type that would agree with both preconditions, and there are infinitely many types, this is disastrously inefficient. QuickChick does attempt to use the first precondition to instead enumerate the correct argument type in a

```
| typ_app : forall fe xe tfrom tto g,
    typing g fe (TArrow tfrom tto) ->
    typing g xe tfrom ->
    typing g (App fe xe) tto
```

Fig. 14. Application constructor for typing in Rocq

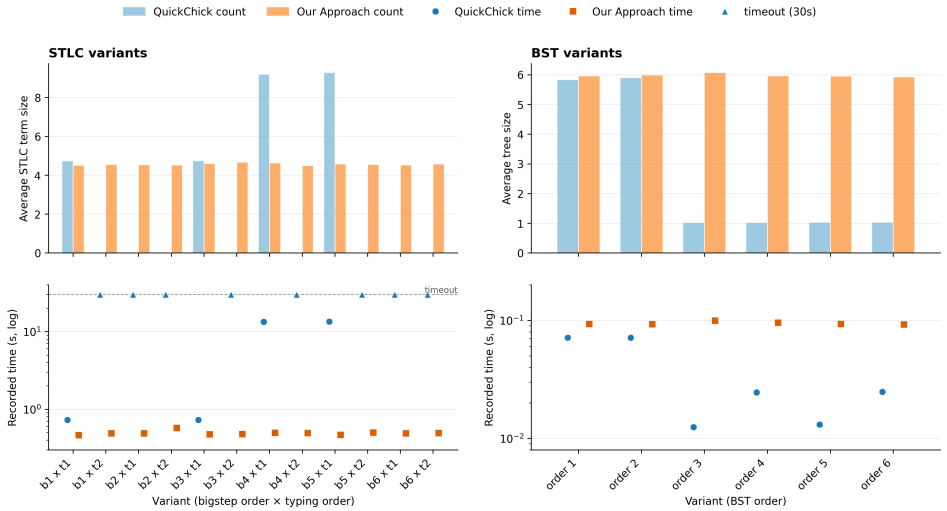

Fig. 13. Time to generate 10,0000 tests and average size of generated STLC and BST terms.

constrained manner, but the user cannot derive this enumerator: derivation fails due to the ordering of the preconditions in the App rule. With this faulty ordering, the enumerator first attempts to infer the type of the function. This leaves the second precondition with no unconstrained variables, so the typing relation for the argument must be checked. Unfortunately, an efficient checker cannot yet exist. This precludes the checker from inferring the type with an enumerator, leaving it disastrously inefficient. On the other hand, in the other ordering of the App rule, the type-inferring enumerator does not need a typechecker and can just infer the types for both argument and function, and check that the argument's type equals the input type of the function, leading to an efficient checker [31].

Even in the half of orderings with an efficient checker for `typing` (those with typing order `t1`), the results are inconsistent when the three preconditions of `bigstep` are reordered. The first and third bigstep orderings (`b1 x t1` and `b3 x t1`) for QuickChick are about as successful as the consistent results for our approach. The second ordering places the evaluation of the argument of an application after the substitution of the result of its evaluation into the result of evaluating the applied function.

As a result, QuickChick decides to generate an arbitrary term as the value of the argument, and later checks that the argument steps to it. Naturally, this ordering choice leads to a hanging generator. The fourth and fifth orderings of `bigstep`'s App rule do not hang, but cause the preservation test to take 11 seconds and lead to 7000 discards, where the good orderings finish in under a second and have 0 discards. As QuickChick increases the size as it tries more terms, because the fourth and fifth orderings had so many discards, their terms are given much larger fuel, which is why they appear to have such a high average constructor count. In actuality, those constructors are disproportionately Abs. Very little of that excess size goes into App constructors, which are often discarded due to the poor bigstep ordering. This is caused by ordering the function

```
| big_app : forall e1 e2 x t body v v',
  bigstep (subst x v body) v' ->
  bigstep e1 (Abs x t body) ->
  bigstep e2 v ->
  bigstep (App e1 e2) v'
```

Fig. 15. Application constructor for big-step in Rocq

evaluation after the substitution evaluation, which, being in function form, forces an arbitrary generation of the variable, in turn leading to discards.

All in all, only 2 out of 12 orderings allow QuickChick to efficiently test preservation. Our approach, on the other hand, produces the very same property and generators regardless of ordering decisions, and those choices lead to efficient tests which finish in a fraction of a second, have a broad distribution of sizes, and no discards.

5.3.2 Reordering BST. We performed a similar study on a property that says the BST relation is maintained through insertion.

```
Theorem insert_BST : forall (t : Tree) (lo x hi : nat),
    bst t lo hi -> between lo x hi -> bst (insert x t) lo hi.
```

This version of the BST relation and theorem leads to 6 possible orderings. Both the baseline and our approach both produce trees at roughly the same speed and with no discards across all orderings, but the sizes of the generated trees are vastly different. 4 out of 6 possible orderings lead to QuickChick generators whose average tree size generated is 1, the size of a leaf, while our approach consistently produces higher average tree sizes, even higher than the best QuickChick generators, with almost 6 nodes and leaves per generated tree on average.

5.4 Applicability in PLF

To demonstrate the broader applicability of our technique, we select several interesting theorems from Programming Language Foundations [33], the second volume of the Software Foundations series of online textbooks which deals with programming language metatheory. The theorems are selected to include interesting relations as preconditions, ignoring trivial theorems (such as unit tests) or relations that involve higher order data (such as in Hoare logic). They include further properties about STLC (using a smallstep relation), properties of a simple imperative language, Imp, and its component sublanguages (Table 2). Most of the theorems were run with increasing size terms up to a maximum of 7 until they achieved either 20000 successes (which evidences the theorem's correctness) or 40000 discards (at which point testing gives up, as far too few successes exist to have comprehensively covered the space of valid inputs). For most theorems tested, 0 generated inputs were discarded, displaying the effectiveness of our approach in selecting the optimal property.

We discuss the few exceptional cases below. The `loop_never_stops` theorem asserts that for any start and end state, an infinite loop Imp program never can start in the start and reach the end state. The discards support this negation: no input can be generated.

In three cases (`substitution_preserves_typing`, `unique_types`, and `preservation`), we also encountered a surprising amount of discards. In the case of `substitution_preserves_typing`, the selected schedule is Backtracking. The theorem has the following shape:

```
Theorem substitution_preserves_typing :
  forall Gamma x U v t S,
    typing ((x, U) :: Gamma) t S ->
    typing [] v U ->
    typing Gamma ([x:=v] t) S.
```

The selected schedule first generates a typing derivation whose context is required to have the shape `(x, U) :: Gamma`, exposing the variable and type needed for substitution.

It then generates a closed value of type U and checks the conclusion of the theorem:

```
B: ((x, U) :: Gamma, t, S) ← typing (x_U_Gamma^o t^o S^o)
P: v ← typing [] v^o U
   check (typing Gamma ([x:=v] t) S)
```

The source of discards is the required shape of the generated context: if the first typing generator produces an empty context, then its result cannot match (x, U) :: Gamma, and the schedule backtracks. This is precisely the failure mode captured by Backtracking density. The generator is not relying on arbitrary generation followed by a restrictive check; instead, it generates a well-typed term and then requires one generated output to have a particular shape. Retrying that local backtracking step substantially reduces discards: one retry reduces discards from 2,739 to 307, two retries reduce them to 80, and nine retries to 20, while still completing in under 8 seconds.

In the case of unique_types, the reason is related to a concern we have mostly put off for now: treatment of sizes. QuickChick uses size as a limit on the depth of generated terms; in the case of generators for inductive relations, that depth is a limit on the size of the proof derivation that the generated term satisfies the relation. However, for certain relations the size of terms generated this way can be larger than the depth of the proof. As a result, when using a similar bound in subsequent checks (such as a typing hypothesis in unique_types), that check might need additional fuel to terminate. A simple combinator can be used to provide such additional fuel automatically to generators and checkers later in a sequence—we discuss that more in the next section. The result, shown as `unique_types with 2x+1 size retry`, eliminates these discards.

In the case of preservation, despite the generators for the typing and smallstep relation being at least Partial, roughly half of the well-typed terms cannot step, as they are already values (abstractions or boolean constants). Even with the size combinator discussed above, the testing characteristics of this property barely change. Still, aside from such trivial discards, the selected property generates and checks complex STLC terms successfully.

Table 2. QuickChick theorem-testing results

Theorem Name	Language	Time (s)	Successes	Discards	Size
substitution_preserves_typing	STLC	8.04	20,000	2,717	5
with 1 retry	STLC	7.98	20,000	322	5
with 2 retries	STLC	7.87	20,000	74	5
with 9 retries	STLC	8.45	20,000	20	5
weakening_empty	STLC	2.21	20,000	0	all
unique_types	STLC	1.27	20,000	6,798	5
unique_types with 2x+1 size retry	STLC	3.04	20,000	0	5
progress	STLC (w/ smallstep)	1.34	20,000	0	all
preservation	STLC (w/ smallstep)	6.61	20,000	19,092	5
preservation with 2x+1 size retry	STLC (w/ smallstep)	6.96	20,000	19,140	5
subject_expansion (falsified!)	STLC (w/ smallstep)	.002	2	0	all
type_soundness	STLC (w/ multistep)	2.15	20,000	0	all
step_deterministic	Addition	0.41	20,000	0	all
stack_step_deterministic	stack-based arith	0.091	20,000	0	all
aeval_if_aevalR	arith	0.27	10,000	0	all
beval_if_bevalR	boolean + arith	0.34	10,000	0	all
ceval_deterministic	Imp	2.62	10,000	0	3
plus2_spec	Imp program spec	0.047	20,000	0	all
XtimesYinZ_spec	Imp program spec	0.018	20,000	0	all
loop_never_stops	Imp program spec	0.36	0	40,000	all
no_whiles_if_no_whilesR	Imp	7.13	20,000	0	all

Table 3. Ablation of the schedule-ranking metrics.

Suite	Policy	Decisions	Multi	Tied	Narrowed	Selected	Changes
STLC	DENSITY-THEN-VARS	282	16	9	9	9	–
STLC	VARS-THEN-DENSITY	282	16	0	0	0	0
STLC	VARS-ONLY	282	16	0	0	0	0
STLC	DENSITY-ONLY	282	16	9	0	0	6
BST	DENSITY-THEN-VARS	42	3	3	0	0	–
BST	VARS-THEN-DENSITY	42	3	3	1	0	0
BST	VARS-ONLY	42	3	3	0	0	0
BST	DENSITY-ONLY	42	3	3	0	0	0
Cedar	DENSITY-THEN-VARS	1380	425	417	145	65	–
Cedar	VARS-THEN-DENSITY	1380	425	403	88	37	0
Cedar	VARS-ONLY	1380	425	403	0	0	67
Cedar	DENSITY-ONLY	1380	425	417	0	0	92

5.5 Metric Component Ablation

Our scheduler compares candidate schedules using a two-component score: density and variable dependency. By default, these scores are compared lexicographically, with density as the primary component and variable dependency as the secondary component. This choice was effective across our case studies, but it leaves open the question of how much each component contributes to the final schedules selected by the algorithm. In particular, readers might reasonably wonder whether variable dependency is redundant as a secondary metric after density, whether conversely density is redundant as a secondary metric after variable dependency, or whether the choice of which component is the primary metric substantially affects the resulting schedules.

To evaluate these possibilities, we performed an ablation study over our core case studies: BST, STLC, and Cedar. We reran schedule selection under four policies: our default policy, DENSITY-THEN-VARS; the reverse policy, VARS-THEN-DENSITY; and two single-component ablations, DENSITY-ONLY and VARS-ONLY. This lets us distinguish two questions: whether reversing the priority of the two components changes the selected schedules, and whether removing either component changes the selected schedules. In Table 3, for each policy, we report the total number of scheduling decisions to make (*Decisions*), the number of decisions with more than one option (*Multi*), the number of decisions whose candidate schedules are tied under the primary metric (*Tied*), the number of decisions whose ties are reduced in size by the secondary metric (*Narrowed*), the number of ties resolved to a single best choice by the secondary metric (*Selected*), and the number of selected schedules that differ from the default policy as a result (*Changes*).

The STLC typing and big-step case study illustrates the importance of variable dependency as a tie-breaker. Under the standard DENSITY-THEN-VARS policy, nine density ties occur, all of which are narrowed by the secondary variable-dependency metric; without it, six selected schedules would differ. By contrast, when variable dependency is primary, no ties occur, leaving density unused. Indeed, variable dependency alone selects the same schedules as DENSITY-THEN-VARS.

The BST case study differs from STLC in the role of the secondary metric. Each of its three multi-option scheduling decisions is tied under either primary metric, but only density narrows a tie when used secondarily; variable dependency provides no additional discrimination. Nevertheless, every policy ultimately selects the same schedules.

The significantly more complex Cedar case study provides the strongest evidence that the two metrics are complementary. Cedar requires far more scheduling decisions than the other case

studies, and many more opportunities for the ranking policy to matter. Under the default DENSITY-THEN-VARS policy, 417 of the 425 multi-option decisions contain at least one density tie, and the secondary variable-dependency metric narrows such ties in 145 decisions, uniquely selecting among tied candidates in 65 of them. Reversing the priority gives a similar picture: under VARS-THEN-DENSITY, 403 multi-option decisions contain a variable-dependency tie, and the secondary density metric narrows such ties in 88 decisions, uniquely selecting among tied candidates in 37.

Despite many tie-breaking opportunities, reversing the two metrics does not change any of the 1380 selected schedules in Cedar. This suggests that density and variable dependency often agree about the best schedules, even when they distinguish candidates in different ways. However, the single-component ablations show that neither metric is redundant. Using only variable dependency changes 67 selected schedules relative to the default policy, while using only density changes 92. Thus, the two metrics capture overlapping but not identical aspects of schedule quality: their order matters less than the fact that both are present. That said, for Cedar, the majority of ties are neither narrowed nor resolved by the current heuristic. This unresolved gap indicates that further efforts at improving the schedule selection heuristic are worthwhile.

6 Discussion

There are three aspects of property-based testing that contribute crucially to its effectiveness that we have mostly steered clear of discussing so far in the paper: sizes, shrinking, and distribution. In terms of the size of generated inputs, the interactions between different inductive relations pose a novel interesting challenge and opportunity; on the other hand, shrinking and distributions are pervasive concerns across the PBT literature, and we inherit their treatment from QuickChick.

Size. As seen in the previous section, size bounds on generated inputs can have significant implications for performance. In particular, treating the input size as a limit on the depth of all generations and checks in a theorem can lead to running out of fuel: for example, if a fuel is used to generate a well-typed term and the same fuel is used as the bound on the number of steps in a big-step evaluator, it is possible (but not too common) to actually run out of fuel!

We introduced a combinator that executes a generator or a checker, and locally backtracks picking a larger size if it runs out of fuel. For now, we allow users to enable this combinator (keeping in line with QuickChick's design of letting expert users fine tune its automated output), but a more principled treatment of size across relations is an interesting open problem.

Shrinking. One of the most important components of property-based testing after generation is minimizing counterexamples before presenting them to the user. Traditionally, this is achieved by a process called *shrinking*: once a counterexample is found, a PBT framework tries increasingly smaller variations of it until it reaches a local minimum that still falsifies the property under test. The two main strategies for shrinking are *external* and *internal*: in external shrinking, a user or type-derived function is used to create a list of candidate inputs from a given counterexample, and they are all progressively tried in a best-first search approach; in internal shrinking, the generators are reused during the shrinking process regenerating increasingly smaller counterexamples.

The choice of shrinking strategy is *completely orthogonal* to the approach discussed in this paper. As we built our approach on top of QuickChick, which uses external shrinking, and as our properties are explicit in the preconditions they impose (which means that we can use them to filter out any smaller counterexamples that violate them), we have opted for a straightforward type-driven external shrinking strategy. That said, if QuickChick incorporates internal shrinking support, that capability should readily translate to the advances we proposed in this paper.

Distributions. Another important concern in property-based testing is the distribution of generated test inputs. Frameworks such as Haskell's QuickCheck and Rocq's QuickChick offer generator combinators that pick among multiple sub-generators while annotating them with weights that shape the resulting distribution. The most common such generator is `frequency`:

```
frequency : list (nat * G A) -> G A,
```

which takes a list of generators together with natural numbers, and induces a discrete distribution among the options. For example, `frequency [(1, ret 0),(2, ret 42)]` generates 0 one third of the time ($\frac{1}{1+2}$) and 42 the other two thirds.

Such weights can have a significant impact on testing, and in our implementation we again follow the exact same approach as prior work on QuickChick: the default option is dictated by established folklore—weight equal to 1 for non-recursive constructors and weight proportional to the current size for recursive ones—we allow expert users to override these weights either with commands during derivation time, or afterwards by inspecting and modifying the derived properties. Recent work has shown that distributions can be automatically tuned [41], and any future work in this domain should straightforwardly apply in our setting as well.

7 Related Work

Testing in Proof Assistants. Testing lemmas and conjectures before attempting to prove them in a proof assistant is a natural well-explored idea. Beginning with [14], which introduced the first QuickCheck for Agda, testing tools have been implemented for a variety of proof assistants, offering varying degrees of automation. Most frameworks offer type-based generation either automatically or at little effort to users (e.g. aided by typeclasses): Agda did so early on while also offering proofs of completeness [15], a decade later Isabelle offered enumeration-based testing [3], and QuickChick followed suit soon thereafter [32], and recently Lean introduced its Plausible[8] framework.

More sophisticated approaches take specifications or other forms of properties into account. The most related line of work is that of QuickChick, which we have thoroughly discussed throughout the paper as we're building on top of it and generalizing it. Lampropoulos et al. [24] developed a way of generating inputs for an inductive relation after users specify the generation mode and Paraskevopoulou et al. [31] generalized that approach to allow for deriving enumerators and checkers. As discussed throughout the paper, neither of these approaches had a way of estimating the efficiency of various recursive or non-recursive generators, enumerators, and checkers they'd rely on, so they defaulted to requiring users to input modes and derive all such instances needed, while also addressing preconditions in the order that they appear. This work fully generalizes both approaches, offering more push-button automation while retaining the customizability of its predecessors. Also in Rocq, earlier work [42] also explores a mode analysis for inductive relations, but only attempts to extract purely functional content from them. The second prominent line of work is the pioneering work of Isabelle's QuickCheck [2, 3], which derived enumerators satisfying constraints for specifications in a subset of Isabelle. That work also heavily relies on a logic programming inspired mode analysis, however its approach is local: it doesn't recursively consider the efficiency of other enumerators needed, and is therefore less powerful. Recent work in Lean [20] proposed a synthesis approach that targets Lean specifications creating correct-by-construction generators, but does not support inductively defined ones. In other proof assistants, early work for Agda [26] used a primitive form narrowing-based search techniques to offer push-button test automation (in the style of Lazy SmallCheck [35] and follow ups [7, 16, 23]), which takes advantage of laziness to prune large parts of the input space quickly. The efficiency of such approaches, however, is naturally limited by the laziness of the program being tested. More recently, van der Rest and Swierstra [43] explored generic enumeration of indexed type families also in Agda, which

could, in principle, be used for property-based testing, but the focus was on correctness (rather than testing performance) and there was still non-trivial manual effort involved. Finally, ACL2 [5], an automated theorem prover for Lisp, also offers a sophisticated testing framework which was tightly integrated with its invariant discovery capabilities to help disprove conjectured generalizations and lemmas. The lack of the rigid structure of a dependently typed setting meant that there was less information to automatically leverage, but the kind of tight integration provided between testing and proving would definitely be a welcome addition for proof engineers.

Logic Programming. Outside of random testing and proof assistants, mode analysis is an established technique in logic and functional logic programming for determining whether variables should be inputs or outputs. Logic programming systems have employed mode analyses early on to optimize their runtime or memory overheads [11, 37], follow-ups putting mode analysis into practice in languages like Mercury [38], and recent work on using a simple mode analysis to convert undirected logical relations into functional programs on streams in MiniKanren [44]. The key difference is that in the context of generation, we're not only interested in whether a mode is valid, we also want to compare the behavior of different modes as generators for random data.

At the same time, logic-programming-based approaches have also been used for data generation. Prolog-style search has been proposed to tackle the specific problem of generating well-typed lambda terms, primarily with the goal of studying their combinatorial properties [1, 39]. Moreover, constrained logic programming (CLP) has been successfully used to generate data structures with complex invariants written in a declarative style [12, 13], leveraging specialized CLP solvers during generation to yield significant speedups. However, such approaches still perform search during generation, whereas we attempt to eliminate the need to search during generation altogether.

Finally, functional and functional logic programming frameworks have been leveraging laziness to improve generation, by lazily generating only the parts of data structures that are necessary to ensure constraints are satisfied, which has been shown to be effective when the constraints are sufficiently lazy [6, 16–18, 23, 26, 35]. Such advances are largely orthogonal to the work presented in this paper: adopting a more lazy approach to generation could potentially improve our generation approach, but it wouldn't solve the underlying scheduling problem.

8 Conclusion and Future Work

In this paper, we proposed a new approach that makes it easy to effectively test theorems. We start from infrastructure provided by QuickChick, which provides powerful building blocks for writing tests in Rocq, but with some expert assembly required. Our key contribution, a density metric for inductive relations, allows us to automatically find a high-quality test without user intervention. Along the way, we also use our density metric to improve QuickChick's internal algorithms. Our approach bridges the gap between effective and usable testing approaches for theorem provers, giving users of all experience levels access to almost instant feedback about their theorems.

In the future, while our notion of density is generally applicable, our implementation is currently specialized to theorems written with inductive relations in Rocq. Moving forward, we hope to adapt our approach to work on a wider range of theorems and in a wider range of languages. The key challenge is to adapt density to work for predicates that are not expressed as inductive relations. This would increase our system's performance on some theorems in Rocq, but critically it could also be used to help improve data generation for property-based testing in more traditional programming languages.

Data Availability Statement

Our implementation as a QuickChick extension will be made publicly available through Github. We intend to submit an artifact for evaluation containing this implementation, reproducing all experiments contained in this paper.

Acknowledgments

We want to thank Benjamin Pierce, John Hughes, Koen Claessen, Michael Hicks, and the anonymous reviewers for their constructive feedback that helped improve the paper. This work was funded by NSF #2145649, CAREER: Fuzzing Formal Specifications. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Maciej Bendkowski, Katarzyna Grygiel, and Paul Tarau. 2017. Boltzmann Samplers for Closed Simply-Typed Lambda Terms. In *Practical Aspects of Declarative Languages*, Yuliya Lierler and Walid Taha (Eds.). Springer International Publishing, Cham, 120–135.

[2] Lukas Bulwahn. 2012. The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof. In *2nd International Conference on Certified Programs and Proofs (Lecture Notes in Computer Science, Vol. 7679)*. Springer, 92–108.

[3] Lukas Bulwahn. 2012. Smart Testing of Functional Programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (Lecture Notes in Computer Science, Vol. 7180)*. Springer, 153–167.

[4] Lukas Bulwahn. 2013. *Counterexample Generation for Higher-Order Logic Using Functional and Logic Programming*. Ph. D. Dissertation. Technische Universität München. http://d-nb.info/1033891142/34

[5] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. 2011. Integrating Testing and Interactive Theorem Proving. In *10th International Workshop on the ACL2 Theorem Prover and its Applications (EPTCS, Vol. 70)*. 4–19. http://arxiv.org/abs/1105.4394

[6] Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 8475)*. Springer, 18–34. doi:10.1007/978-3-319-07151-0_2

[7] Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015). doi:10.1017/S0956796815000143

[8] Lean Community. 2025. Plausible. https://github.com/leanprover-community/plausible

[9] Richard H. Connelly and F. Lockwood Morris. 1995. A generalization of the trie data structure. *Mathematical Structures in Computer Science* 5, 3 (1995), 381–418. doi:10.1017/S0960129500000803

[10] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew M. Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 118 (April 2024), 28 pages. doi:10.1145/3649835

[11] Saumya K. Debray and David S. Warren. 1988. Automatic mode inference for logic programs. *The Journal of Logic Programming* 5, 3 (1988), 207–229. doi:10.1016/0743-1066(88)90010-6

[12] Kyle Dewey, Lawton Nichols, and Ben Hardekopf. 2015. Automated Data Structure Generation: Refuting Common Wisdom. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 32–43. doi:10.1109/ICSE.2015.26

[13] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 725–730. doi:10.1145/2642937.2642963

[14] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. 2003. Combining Testing and Proving in Dependent Type Theory. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science, Vol. 2758)*. Springer, 188–203. http://www.cse.chalmers.se/~peterd/papers/Testing_Proving.pdf

[15] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. 2004. Random Generators for Dependent Types. In *First International Colloquium Theoretical Aspects of Computing (Lecture Notes in Computer Science, Vol. 3407)*. Springer, 341–355. doi:10.1007/978-3-540-31862-0_25

[16] 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 *24th European*

Symposium on Programming (Lecture Notes in Computer Science, Vol. 9032). Springer, 383–405.
[17] Sebastian Fischer and Herbert Kuchen. 2007. Systematic generation of glass-box test cases for functional logic programs. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*. ACM, 63–74. http://www-ps.informatik.uni-kiel.de/~sebf/pub/ppdp07.html
[18] Justin Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2024. Generating Well-Typed Terms that are not "Useless". In *Proceedings of the ACM on Programming Languages (PACMPL), Volume POPL*. doi:10.1145/3632919
[19] Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science, Vol. 1032. Springer, Berlin, Heidelberg. doi:10.1007/3-540-60761-7
[20] Harrison Goldstein, Hila Peleg, Cassia Torczon, Daniel Sainati, Leonidas Lampropoulos, and Benjamin Pierce. 2026. The Search for Constrained Random Generators. In *Proceedings of the ACM on Programming Languages (PACMPL), Volume PLDI*. doi:10.1145/3808329
[21] Ralf Hinze. 2000. Generalizing generalized tries. *Journal of Functional Programming* 10, 4 (2000), 327–351. doi:10.1017/S0956796800003713
[22] Leonidas Lampropoulos. 2018. *Random Testing for Language Design*. Ph. D. Dissertation. University of Pennsylvania.
[23] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, 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)*. doi:10.1145/3009837.3009868
[24] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating Good Generators for Inductive Relations. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*. doi:10.1145/3158133
[25] Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing In Coq*. Electronic textbook.
[26] Fredrik Lindblad. 2007. Property Directed Generation of First-Order Test Data. In *8th Symposium on Trends in Functional Programming*. Intellect, 105–123.
[27] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-Driven QuickChecking of Compilers. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 15 (Aug 2017), 23 pages. doi:10.1145/3110259
[28] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37
[29] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph. D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.
[30] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 91–97. doi:10.1145/1982595.1982615
[31] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing Correctly with Inductive Relations. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3519939.3523707
[32] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Denes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *6th International Conference on Interactive Theorem Proving (ITP)*. doi:10.1007/978-3-319-22102-1_22
[33] Benjamin C. Pierce. 2018. *Software Foundations*. Electronic textbook.
[34] Jacob Prinz and Leonidas Lampropoulos. 2023. Merging Inductive Relations. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3591292
[35] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *1st ACM SIGPLAN Symposium on Haskell*. ACM, 37–48. doi:10.1145/1543134.1411292
[36] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. doi:10.1145/3607860
[37] Zoltán Somogyi. 1987. A System of Precise Models for Logic Programs. In *International Conference on Logic Programming*. https://api.semanticscholar.org/CorpusID:27257200
[38] Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. 1996. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming* 29, 1-3 (October-December 1996), 17–64. http://www.mercurylang.org/information/papers/jlp.ps.gz
[39] Paul Tarau. 2015. On Type-directed Generation of Lambda Terms. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015*. http://ceur-ws.org/Vol-1433/tc_12.pdf
[40] The Rocq team. 1984-now. *The Roq Proof Assistant*. Version 9.0. https://rocq-prover.org/doc/V9.0.0/refman/index.html

[41] Ryan Tjoa, Poorva Garg, Harrison Goldstein, Todd D. 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), 894–921. doi:10.1145/3763082

[42] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. 2012. Producing Certified Functional Code from Inductive Specifications. In *Second International Conference on Certified Programs and Proofs (CPP) (Lecture Notes in Computer Science, Vol. 7679)*. Springer. http://cedric.cnam.fr/~delahaye/papers/relext-coq%20%28CPP%2712%29.pdf

[43] Cas van der Rest and Wouter Swierstra. 2022. A completely unique account of enumeration. *Proc. ACM Program. Lang.* 6, ICFP, Article 105 (Aug. 2022), 27 pages. doi:10.1145/3547636

[44] Ekaterina Verbitskaia, Igor Engel, and Daniil Berezun. 2024. A Case Study in Functional Conversion and Mode Inference in miniKanren. 107–118. doi:10.1145/3635800.3636966

A Order to Schedule Algorithm

Input: V : set of variables
π : permutation of hypotheses
F : subset of variables fixed by inputs
Output: Schedule S
$\sigma \leftarrow F$ // instantiated variables
$S \leftarrow [\,]$
foreach $H \in \pi$ **do**
  $O \leftarrow \text{OutputPossibleVars}(H) \setminus \sigma$
  $N \leftarrow \text{NoOutputVars}(H) \setminus \sigma$
  foreach $v \in N$ **do**
    $S \leftarrow S \cdot [\, v \leftarrow \text{arbitrary} \,]$
    $\sigma \leftarrow \sigma \cup \{v\}$
  end
  if $O \neq \emptyset$ **then**
    $S \leftarrow S \cdot [\, O \leftarrow H \,]$
    $\sigma \leftarrow \sigma \cup O$
  else
    $S \leftarrow S \cdot [\, \text{check}(H) \,]$
  end
end
foreach $v \in (V \setminus \sigma)$ **do**
  $S \leftarrow S \cdot [\, v \leftarrow \text{arbitrary} \,]$
  $\sigma \leftarrow \sigma \cup \{v\}$
end
return S

Algorithm 1: Order to Schedule

For every hypothesis in an ordering, converting it to a sequence of steps in its schedule amounts to collecting the uninstantiated variables it constrains and partitioning them based on whether they can be used as outputs or not. For an example of a variable that cannot be used as an output, consider the case when the variable is an argument to a function. Functions are opaque to our analysis, and we have no way of generating only elements that are in the image of an arbitrary function. After parititioning, for the variables that cannot be used as outputs, we add steps to arbitrarily generate them, followed by a single generation step for all the potential outputs satisfying the hypothesis. If there are no possible outputs, this generation step is replaced by a check, as no variable constrained by the hypothesis remains uninstantiated. After all hypotheses have been consumed, the schedule ends by arbitrarily instantiating any variables that are not constrained by any hypothesis.

Received 2026-03-17; accepted 2026-06-10