SPECTACULAR: Finding Laws from 25 Trillion Terms (Extended)

Matthías Páll Gissurarson
Chalmers Institute of Technology
Gothenburg, Sweden
pallm@chalmers.se

Diego Roque
Dark Forest Technologies
New York, USA
diego9627@gmail.com

James Koppel
MIT
Cambridge, USA
jkoppel@mit.edu

***Abstract*—We present SPECTACULAR, a new tool for automatically discovering candidate laws for use in property-based testing. By using the recently-developed technique of ECTAs (Equality-Constrained Tree Automata), SPECTACULAR improves upon previous approaches such as QUICKSPEC: it can explore vastly larger program spaces and start generating candidate laws within 20 seconds from a benchmark where QUICKSPEC runs for 45 minutes and then crashes (due to memory limits, even on a 256 GB machine). Thanks to the ability of ECTAs to efficiently search constrained program spaces, SPECTACULAR is fast enough to find candidate laws in more generally typed settings than the monomorphized one, even for signatures with dozens of functions.**

I. INTRODUCTION

Testing is the art of checking that a program works in some scenarios in order to gain evidence that it works in all. This is especially relevant in the age of LLMs, where establishing ground-truth to verify auto-generated programs is important. In its basic form, a developer must manually create a set of sample inputs and the expected behavior on each. For 20 years, the Haskell community has boasted their ability to automate this by writing down just a few properties, and letting a *property-based testing* tool [1] generate the inputs automatically. But this only replaces hard labor with hard thought: it is still difficult to think of the right properties.

Yet every program implies its set of properties. By generating and testing a vast number of properties that might hold for a given program, a developer need merely select from the smörgåsbord that does hold. This is the idea of QUICKSPEC, capable of generating interesting properties on numerous data structures starting with just a list of functions to consider — so long as that list is small. Beyond the single digits, the exponential growth overwhelms its search abilities.

We introduce SPECTACULAR, a new tool for automatically discovering program properties which uses advances in program synthesis to search spaces of candidate programs which are orders-of-magnitude larger than can be searched by QUICKSPEC. For example, for the 5 functions and 3 constants in Figure 1, SPECTACULAR finds 60 laws compared to QUICKSPEC's 28; even running in a restricted mode, it still finds more laws than QUICKSPEC in less than half the time. It does this thanks to its use of the recently-introduced ECTA (Equality-Constrained Tree Automata) data structure [2], capable of compactly representing a space of trillions of possible programs, and efficiently enumerating all the ones which are well-typed (or satisfy any other property encodable with equality constraints). The benefits over QUICKSPEC get larger for larger modules. Thanks to ECTAs and our custom enumeration, SPECTACULAR can start generating laws within minutes from the space of all terms up to size 6 on a signature with 92 functions and constants, a space of over 25 trillion terms, with memory consumption never exceeding 1GB. QUICKSPEC, on the same benchmark gets stuck enumerating terms of size 4 and crashes from memory exhaustion after 45 minutes on a machine with 256GB of RAM. And it does all this in only 1400 LOC, compared to QUICKSPEC's 8300.

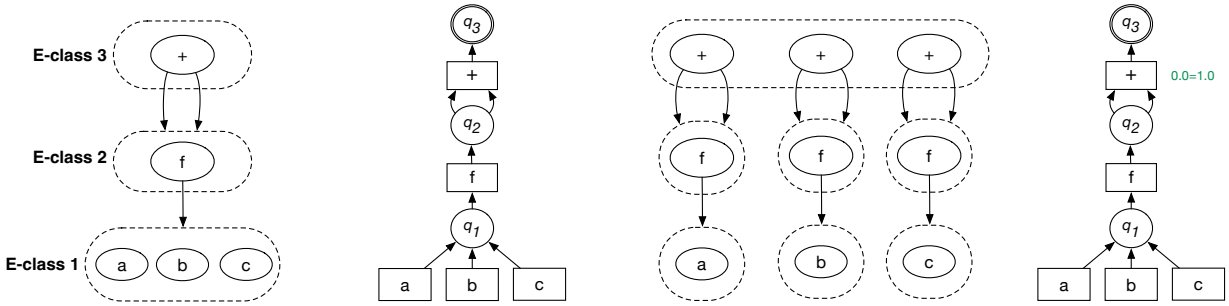
 https://doi.org/10.1109/ICST57152.2023.00035

```
main = tacularSpec [
  con "reverse" (reverse :: [a] -> [a]),
  con "++" ((++) :: [a] -> [a] -> [a]),
  con "[]" ([] :: [a]),
  con "map" (map :: (a -> b) -> [a] -> [b]),
  con "length" (length :: [a] -> Int),
  con "concat" (concat :: [[a]] -> [a]),
  con "0" (0 :: Int),
  con "1" (1 :: Int) ]
```

Figure 1: Example signature for SPECTACULAR, presentation slightly simplified.

```
reverse (reverse xs) == xs
reverse (xs ++ ys) == reverse ys ++ reverse xs
map f (concat lists) == concat (map (map f)) lists)
length (xs ++ ys) == length xs + length ys
```

Figure 2: Example laws generated by SPECTACULAR from the signature in figure 1.



(a) E-graph for $\mathcal{T}$ **(b)** FTA for $\mathcal{T}$ **(c)** E-graph for $\mathcal{U}$ **(d)** ECTA for $\mathcal{U}$

Figure 3: Representations of $\mathcal{T} = \{\mathsf{f}(t_1) + \mathsf{f}(t_2)\}$ and $\mathcal{U} = \{\mathsf{f}(t) + \mathsf{f}(t)\}$, where $t, t_1, t_2 \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.

In summary, this paper makes the following contributions:

- An automated program property discovery approach based on modern program synthesis techniques, specifically ECTAs.
- The SPECTACULAR tool, capable of discovering program properties orders-of-magnitude faster than previous approaches on large examples. The source is available at https://zenodo.org/record/7565003 [3], along with scripts used to generate the benchmarks in the evaluation. Note that SPECTACULAR itself is in the spectacular sub-folder.

II. BACKGROUND

A. Equality-Constrained Tree Automata (ECTAs)

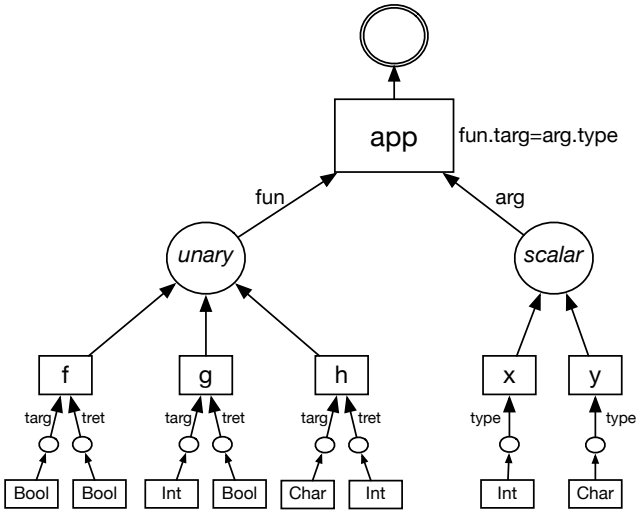


Figure 4: ECTA representing all well-typed size-two terms in the environment $\Gamma_1 = \{x : \texttt{Int}, y : \texttt{Char}, f : \texttt{Bool} \to \texttt{Bool}, g : \texttt{Int} \to \texttt{Bool}, h : \texttt{Char} \to \texttt{Int}\}$.

Equality-constrained tree automata (ECTAs) [2] are a new data structure for representing and enumerating a large space of terms with constraints between subterms. They are most easily explained as an alternative to e-graphs [4], [5], [6] suitable when different subterms cannot be chosen independently. We begin with a brief discussion of e-graphs; see Willsey et al [6] for additional background. We will then give an abridged version of the explanation of ECTAs from [2].

E-graphs = Independence.

Consider selecting a term from some large space of possibilities, where each successive node from the top down is a distinct choice. E-graphs are a compact representation of such spaces where each choice can be made independently. For instance, consider the space $\mathcal{T} = \{\mathsf{f}(t_1)+\mathsf{f}(t_2)\}$ where $t_1, t_2 \in C = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. An e-graph can be constructed by first constructing a node "E-class 1" representing the choice $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, then a node "E-class 2" representing $\{\mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{b}), \mathsf{f}(\mathsf{c})\}$, then a node "E-class 3" which sums two independent choices drawn from E-class 2. Figure 3a depicts this e-graph. Though the size of this space is clearly quadratic in $|C|$, the size of its e-graph is linear. Thanks to this ability, e-graphs have seen application from program synthesis [6], [7] to superoptimization [8] to semantic code search [9] to theorem proving [4]. E-graphs are now known [10], [11] to be equivalent representationally to *finite tree automata* (FTAs). Figure 3b shows an FTA that represents the same term space as the e-graph in figure 3a.

Tree Automata.

A finite tree automaton (FTA) consists of *states* (circles) and *transitions* (rectangles), with each transition connecting zero or more states to a single state. Intuitively, FTA transitions correspond to e-graph nodes, and FTA states correspond to e-classes. Importantly, both data structures, along with the similar version space algebras [12], thrive on spaces where terms share some top-level structure, while their divergent subterms can be chosen *independently* of each other.

Challenge: Dependent Joins. Consider now the term space $\mathcal{U} = \{\mathsf{f}(t) + \mathsf{f}(t)\}$, where $t \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, that is, a sub-space of $\mathcal{T}$ where both arguments to f *must be the same term*. Such "entangled" term spaces arise naturally in many domains. For example, in term rewriting or logic programming, we want to represent the subset of $\mathcal{T}$ that matches the non-linear pattern $X + X$. More relevantly, we want to represent the space of well-typed Haskell terms, where, in each application, the type of an argument must equal the parameter type of the function.

Existing data structures are incapable of fully exploiting shared structure in such entangled spaces. Figure 3c shows an e-graph representing $\mathcal{U}$: here, the $+$ cannot be reused because

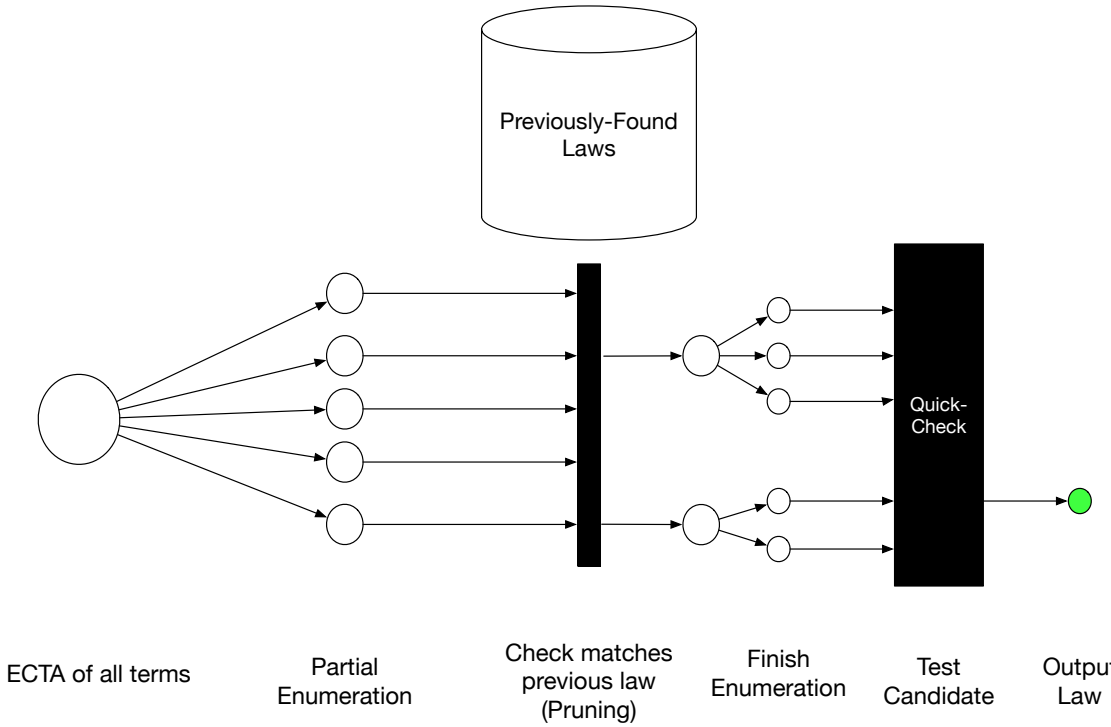


Figure 5: An overview of SPECTACULAR

its children are *independent*, whereas our example requires a dependency between the two children of $+$.

Solution: ECTA. ECTAs address this problem by representing dependent spaces by tree automata whose transitions can be annotated with *equality constraints*. For example, figure 3d shows an ECTA that represents the term space $\mathcal{U}$. It is identical to the FTA in figure 3b save for the constraint $0.0 = 1.0$ on its $+$ transition. This constraint restricts the set of terms accepted by the automaton to those where the sub-term at path 0.0 (the first child of the first child of $+$) equals the sub-term at path 1.0 (the first child of the second child of $+$). The constraint enables this ECTA to represent a dependent join while still fully exploiting shared structure, unlike the e-graph in figure 3c.

Most importantly, ECTAs come equipped with **efficient algorithms for enumerating all terms that satisfy the constraints** based on constraint processing via automata intersection, made available in the optimized ECTA library.

Type-driven Synthesis. A striking example of the power of ECTAs is HECTARE [2], the main existing application of ECTAs. HECTARE is designed as a replacement for HOOGLE+ [13], which solves the problem of *polymorphic type-driven program synthesis*: given a Haskell type such as `(a -> a) -> a -> Int -> a`, intended to take a function f and apply it n times to some input x, it synthesizes a small Haskell program of this type, included the intended solution `\g x n -> foldr ($) x (replicate n g)`. HOOGLE+ clocks in at over $4,000$ lines, using an SMT encoding specifically tuned for this problem. HECTARE is a measly 400 lines: it constructs an ECTA using the ECTA library, and simply runs the standard enumeration procedure. And yet HECTARE is $8\times$ faster.

The SPECTACULAR tool in this paper uses a similar encoding to HECTARE to build an ECTA representing the space of well-typed Haskell programs of a given signature, but tuned to the set of types occurring in the functions of interest, and uses finer-grained enumeration to reduce the number of candidate terms inspected by over $1000\times$ on some benchmarks.

Figure 4 shows an example of a simplified ECTA for types, where type variables are restricted to just one of a few base types. We refer to the ECTA paper [2] for discussion of this encoding and its generalization to arbitrary polymorphism.

B. QuickCheck and the QuickSpec problem

QuickCheck: is a property-based testing framework that uses observational equivalence based on generating arbitrary data and testing to establish the validity of properties [1].

QuickSpec: is a state-of-the-art theory exploration system for Haskell that uses QuickCheck to automatically generate laws based on a set of functions called a *signature* [14]. Originally a naive equation generation system, QUICKSPEC V2 and onwards uses sophisticated techniques based on enumerating terms and *schemas* in order to quickly explore a system of equations [14]. We use QUICKSPEC as the gold standard to compare against in this paper, and many of the techniques we use in SPECTACULAR are based on the techniques used in QUICKSPEC, albeit augmented with ECTA. However, QUICKSPEC does struggle with large signatures, as we detail further in section IV.

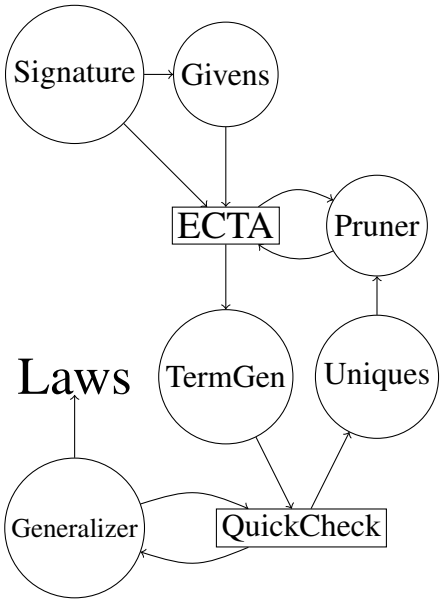


Figure 6: An overview of the SPECTACULAR loop

III. THE SPECTACULAR TOOL

The design of SPECTACULAR is inspired by QUICKSPEC's [14] approach of property discovery by finding and maintaining a set of unique terms which are used for comparison with newly enumerated terms. We emulate QUICKSPEC's signature interface for ease of comparison. We use an ECTA to efficiently represent and enumerate the space of well-typed terms of increasing size that are compared to each other to find properties. We make non-trivial changes to the enumeration of the ECTA to efficiently *prune redundant terms*, reducing the amount of terms to be examined. An overview of SPECTACULAR can be found in figure 5, with an overview of the loop itself in figure 6. We denote Haskell data types with red text, a la `Type`. An example output of SPECTACULAR is provided in figure 7. Here, `x_Int` denotes an arbitray `Int`, `xs_[A]` an arbitrary list of `A`s, and so on.

A. Signatures

Users interact with SPECTACULAR by defining a signature that describe the system of equations that they want to explore. We give an example signature consisting of a collection of list functions in figure 1.[1] This shall serve as our running example. Note that SPECTACULAR is capable of handling of much larger spaces, so it can handle entire modules instead of just manually-specified signatures.

B. Supplying Givens

To generate and check equations, SPECTACULAR needs to be able compare values of a given type for equality and to generate arbitrary values of that type. When compiling Haskell with GHC, this is done by instance resolution during compile

[1]The pseudocode in this figure is almost-identical to the real code, which uses a mechanism to create runtime representations of types and type variables.

```
Laws according to Haskell's (==):
---------------------------------
1. 0 == length []
2. [] == reverse []
3. length xs0_<[A]> == length (reverse xs0_<[A]>)
4. xs0_<[A]> == reverse (reverse xs0_<[A]>)
5. [] == ((++) []) []
6. xs0_<[A]> == ((++) []) xs0_<[A]>
7. xs0_<[A]> == ((++) xs0_<[A]>) []
8. [] == (map f0_<A -> A>) []
9. f1_<A -> A> x0_<A>
 ==  f1_<A -> A> (f1_<A -> A> (f1_<A -> A> x0_<A>))
10. length xs0_<[A]>
 == length ((map f0_<A -> A>) xs0_<[A]>)
11. (map f0_<A -> A>) (reverse xs0_<[A]>)
 == reverse ((map f0_<A -> A>) xs0_<[A]>)
12. length (((++) xs0_<[A]>) xs1_<[A]>)
 == length (((++) (reverse xs0_<[A]>)) xs1_<[A]>)
13. reverse (((++) xs0_<[A]>) xs1_<[A]>)
 == ((++) (reverse xs1_<[A]>)) (reverse xs0_<[A]>)
14. ((++) xs0_<[A]>) (((++) xs1_<[A]>) xs2_<[A]>)
 == ((++) (((++) xs0_<[A]>) xs1_<[A]>)) xs2_<[A]>
15. ((++) (reverse xs0_<[A]>)) xs1_<[A]>
 == reverse (((++) (reverse xs1_<[A]>)) xs0_<[A]>)
Fully monomorphic phase finished..199 terms examined.
43 unique terms discovered.
Starting phase with more types....
Monomorphic phases finished..335 terms examined.
50 unique terms discovered.
Starting mono-polymorphic phase....
16. [] == concat []
17. ((++) (concat xs0_<[[A]]>)) (concat xs1_<[[A]]>)
 == concat (((++) xs0_<[[A]]>) xs1_<[[A]]>)
18. xs0_<[[A]]> == (map ((++) [])) xs0_<[[A]]>
19. xs0_<[[A]]>
 == (map reverse) ((map reverse) xs0_<[[A]]>)
Mono-polymorphic phase done! 1662 terms examined.
96 unique terms discovered.
Starting fully-polymorphic phase....
20. (map f0_<A -> B>) (concat xs0_<[[A]]>)
 == concat ((map (map f0_<A -> B>)) xs0_<[[A]]>)
Done! 3558 terms examined.
100 unique terms discovered
```

Figure 7: Example SPECTACULAR output for terms up to size 5 in 4 phases for the signature in figure 1. in 3.3sec/39MB

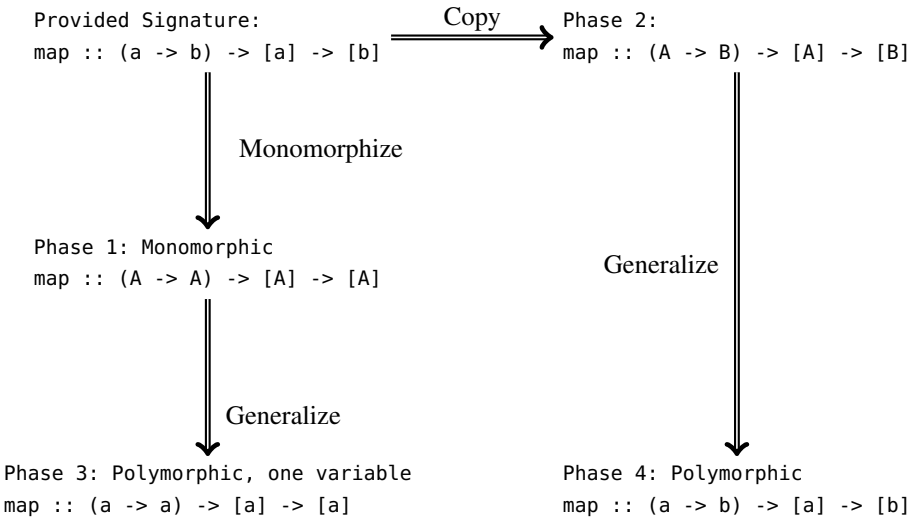


Figure 8: The four phases of SPECTACULAR. Each consecutive phase uses laws and uniques found in the previous phases, but is slightly more general than the last when it comes to the types it explores.

time to find the corresponding arbitrary data generators and equality functions, and insert them into the generated code. However, since SPECTACULAR is generating and evaluating equations at *run-time*, the associated instances must be resolvable at run-time as well. This means that SPECTACULAR requires a run-time mechanism for instance resolution for arbitrary data generators and equalities. This can be done using the `Dynamic` datatype, allowing us to look up and store the associated instances in a data structure at compile-time, and defer the selection of which instances to use to run-time. This means that it can test terms without having to resort to code generation, making the tool simpler and faster. SPECTACULAR does this by looking at the signature and generating its *universe of types* [14], meaning all function arguments and return types. for figure 1, these would be `[A]`, `[[A]]`, `[B]`, `(A -> B)`, and `Int`. This universe is used to generate the *givens* for the signature, which are the equality functions and random variable generators for types in the universe which can then be looked up at run-time. Generating the givens at compile-time from the universe of types means that the users do not have to manually provide these instances, though it comes with the caveat that types that are not in the signature are not considered synthesis targets.

C. Enumerating Terms

An efficient way to explore the space of equations is to enumerate *terms* instead of equations themselves [14]. However, it is important to enumerate only *well-typed* terms. For example, for the terms `reverse :: [a] -> [a]` and `(++) :: [a] -> [a] -> [a]` with added givens `xs :: [a]` and `ys :: [a]`, there are 5460 possible programs of size ≤ 6 of which only 128 are well-typed and 84 are base values!

ECTAs: To efficiently represent the space of well-typed Haskell terms of a given type, SPECTACULAR constructs an ECTA for a fixed size n. The ECTA library provides an efficient interface for enumeration. It conceptually provides the following API:

```
partiallyEnumerate :: ECTA -> PartiallyEnumeratedTerm
```

where a partially-enumerated term can be thought of as a template like `map _ (reverse _)`, where the two underscores stand for ECTAs representing some smaller space of terms[2]. This allows SPECTACULAR to decide whether to expand or discard a branch (see section III-D).

Phasing: Ordering is important when generating laws. As a simple example, if the tool discovers `reverse (reverse x) == x` early, large swaths of the search space need not be enumerated at all, and undesired redundant rules such as `tail (reverse (reverse x)) == tail x` will not be generated. To increase efficiency, and to provide the user results immediately, SPECTACULAR splits its search into several *phases*, each using a larger search space than the previous. Each phase is further stratified by size, guaranteeing that smaller terms are discovered before larger. An overview is provided in figure 8.

1) **Monomorphic**: The first phase monomorphizes all types representing type variables, so that `b`, `c`, and `d` all become the concrete type `A`, where `A` is an arbitrary constant type. For example, the function `map :: (a -> b) -> [a] -> [b]` from figure 1 is monomorphized into `map :: (A -> A) -> [A] -> [A]`. This yields a very small search space, enumerable in seconds. Searching this space first allows SPECTACULAR to discover rewrites that allows aggressive pruning in later phases. This phase can discover rules such as `reverse (reverse xs) == xs`.
2) **Uninterpreted**: The second phase replaces all type variables with distinct constant types (such as `A` and `B`). This means that `map :: (a -> b) -> [a] -> [b]` is specialized

[2]Partially-enumerated terms also store equality constraints between the unenumurated parts; in this case, that the argument type of `map`'s first argument must match the element type of the list in the second argument.

into `map :: (A -> B) -> [A] -> [B]`. This phase does not generally result in many *laws*, but does discover a few *unique*, previously unseen terms from the new types, such as `snd (x_A,x_B) :: B`, used in later phases.

3) **Single-variable polymorphism**: This phase replaces all type variables with the single type variable `a`, but treats this variable as standing for an arbitrary type. For example, `map :: (a -> b) -> [a] -> [b]` becomes `map :: (a -> a) -> [a] -> [a]`. Unlike in the previous phases, this time it contains a proper type variable, `a`. This is the approach taken in QUICKSPEC [14], and allows SPECTACULAR to find most of the laws, unless they require more than one type variable. An example law first discovered at this phase is `reverse (concat reverse xs) == concat (map reverse xs)`.
4) **Arbitrary polymorphism** (optional): This phase generalizes all the type variable representing types in the signature into type variables. This means that `map` will have the polymorphic type `map :: (a -> b) -> [a] -> [b]`. This phase is not run by default because of the vast size of the search space. When running this phase on large term sizes, progress grinds to a halt. An example of a law first discovered at this phase is `concat (reverse (map reverse lists)) == reverse (concat lists)`. Finding this is harder in a monomorphic setting: when `map :: (a -> b) -> [a] -> [b]` becomes monomorphized to `map :: (a -> a) -> [a] -> [a]`, the argument function `f` must have the same input and output type, `f :: a -> a`. But the function `concat :: [[a]] -> [a]` returns a different type than its input.

D. Pruning

The key to efficient exploration of the equation space is the pruning that SPECTACULAR applies *directly* in the ECTA during enumeration. As described in section III-C, ECTA enumeration is based on a loop that does repeated expansion of partially-enumerated terms, which are terms for which there are still some choices to be made. The enumeration runs in a monadic environment that captures the branching that happens during enumeration when choices are made. This means that It is helpful to avoid exploring branches known to only contain terms containing a sub-term that can be rewritten. As an example, if the tool has previously discovered that `x == reverse (reverse x)`, it can discard any partially-enumerated terms containing `reverse (reverse _)`, since any such term is equivalent to the smaller term containing just `x`. Choosing the right pruning strategy is important, we want to be as aggressive as possible while still being sound, in order to find all the relevant equations without having to enumerate and check an intractable amount of terms. In SPECTACULAR, the pruning strategy is based on matching the non-unique terms that SPECTACULAR has discovered up until that point (i.e. terms that are equivalent to any of the unique terms) with the partial terms being enumerated. These non-unique terms are either direct rewrites of an expression (such as `reverse [] => []`), or include a variable, e.g. `xs ++ [] => xs`.

Matching partial terms: A partial term matches another term when the top-level symbol are the same and all of their sub-expressions are the same. However, it might be the case that some sub-expressions of the partial term are not enumerated yet. In that case, SPECTACULAR *suspends* the pending sub-matches, and runs them whenever the pending partial term gets enumerated, allowing the pruning of the branch as soon as SPECTACULAR knows enough about any of the partial terms to make the decision that this branch will not be productive. Enumeration of a term always starts from the top-down, so most of the time the pending matches are suspended on any of the sub-expressions of the top-level term. When a match is found, SPECTACULAR immediately stops enumeration of that branch, and continues with the next one.

E. Interleaving Testing and Enumeration

For freshly generated terms, SPECTACULAR starts by trying to rewrite it to a smaller term using the previously-discovered equations. Since it explores the space of terms in order of size, the existence of a rewrite to a smaller term means that an equivalent term has already been inspected, and thus any laws using the larger term would be redundant. The larger term can thus be discarded. A lot of the performance comes from being able to discard a term before it is tested or even before it is generated. By interleaving testing and enumeration [14], SPECTACULAR can learn rewrites that allow it to prune more aggressively, making it a lot more efficient than generating all the terms at once and then start testing. In SPECTACULAR, this is done by generating terms in batches, and generating and testing all terms of a given *type* and size before proceeding to the next one. This allows us to learn all the rewrites for a certain size before proceeding to a bigger size, and so SPECTACULAR can prune aggressively when we know a sub-term has a rewrite. Generating per type is also beneficial, since SPECTACULAR might generate the same expression at a different type multiple times (e.g. `[] == [] ++ [] :: [a]`, `[] ++ [] :: [Int]`, etc.). By learning the rewrite for one type and generalizing it, SPECTACULAR can prune those expressions at other types.

F. Testing of Terms

SPECTACULAR tracks a set of *unique terms* of each type that are not morally equivalent [15] to any other term SPECTACULAR has encountered so far. When a new candidate term has been generated, SPECTACULAR tests it against all the other unique terms we've discovered of that type. Let `xs :: [a]` be the unique term and `xs ++ []` be the candidate term. By using the generated equality instance SPECTACULAR can construct the term representing the equality `xs == xs ++ []`. Once we have a term, we have to turn it into a `Property` that we can test using QuickCheck. Using the equality and random variable generators from the given we generated from the signature, we have all the components of the term represented as `Dynamic` instances. This means that we do not need to do any compilation or code-generation step, we can immediately use the random variable generators to generate

variable assignments, and apply the dynamic representations of the functions and equality to generate the `Property`.

Implementation: To do this, SPECTACULAR generates a `Dynamic` containing a `Property`. It then generates a variable assignment for every variable in the expression, making sure that the same variable gets the same value, no matter how many copies there are in the expression. A specific GADT is needed to represent the `Dynamic` generator, since we must ensure that the generated values are `Typeable`, so that we can wrap them in `Dynamic` once they've been assigned. The function then generates the `Dynamic` representation of each of the sub-expressions, by looking up the value in the variable assignment map or looking up the `Dynamic` representation of the function from the signature. To support the monomorphization of the type variables, we need to use a function that circumvents the checks done by `Dynamic` at run-time, so that e.g. `A` can be used in place of `B`. `A` and `B` and other type variable representatives are defined as `data A = A Any`, and so can be safely coerced between. Since we only synthesize *well-typed* expressions, we know these coercions are safe. By generating a `Dynamic` representation of the property this way, we can efficiently test equations for validity.

G. Generalization of Laws

To reduce the search space even further, SPECTACULAR only enumerates terms with the one variable for each type, and reuses that variable whenever a variable of that type is needed. This means that when we generate the associativity check, it will be discovered as `(xs ++ xs) ++ xs == xs ++ (xs ++ xs)` This is based on the observation that [14] if an equation holds for any arbitrary `xs` and `ys`, it must in particular also hold whenever `xs == ys`. This means it suffices to explore terms with however many copies of the *same* variable, (`xs` in our example) and then to generalize the law once found. To generalize a law, SPECTACULAR generates all possible variations of the law by renaming each variable and adding more variables as needed. This would generalize the trivial law `(xs ++ xs) ++ xs == xs ++ (xs ++ xs)` to

- `(xs ++ xs) ++ ys == xs ++ (xs ++ ys)`,
- `(xs ++ ys) ++ xs == xs ++ (ys ++ xs)`, and also
- the actual law: `(xs ++ ys) ++ zs == xs ++ (ys ++ zs)`.

We make sure to generate these laws so that the variables are always in the same order to avoid duplicates and remove the ones that are equivalent up to renaming, We then test the most general law first (the one with the most variables) and so on until we find a law that hold (if none is found, the original law is the most general). This way we use variables as a limited form of *schemas* [14]. When a law has been discovered, its most general form is reported, and SPECTACULAR continues until all the phases are finished for all types and type constructors.

IV. EVALUATION

We evaluate the performance of SPECTACULAR against QUICKSPEC. We match the parameters when possible in both tools. We take the examples from the QUICKSPEC repository, including two shown in the paper, and adapt them to SPECTACULAR. This involves removing any QUICKSPEC specific options from the signatures, and adding implementations of random data generators of the user-provided types defined by the example so that we can run QuickCheck.

These tests were run on a cloud-based machine with 32GB of RAM and 6 Intel Xeon E312xx @2GHz 64bit vCPUs.

The signatures we consider are as follows:

- Lists: The signature shown in figure 1, repeated here. All involving basic list functions. This has 6 components.

```
main = tacularSpec [
  con "reverse" (reverse :: [a] -> [a]),
  con "++" ((++) :: [a] -> [a] -> [a]),
  con "[]" ([] :: [a]),
  con "map" (map :: (a -> b) -> [a] -> [b]),
  con "length" (length :: [a] -> Int),
  con "concat" (concat :: [[a]] -> [a]),
  con "0" (0 :: Int),
  con "1" (1 :: Int) ]
```

- Octonions: This example defines an octonion data type and an Arbitrary instance for it. The signature is then defined as below, and has 3 components:

```
main = tacularSpec [
 con "*" (* :: Oct -> Oct), -- product
 con "inv" (recip :: Oct -> Oct) -- inverse
 con "1" (1 :: Oct)] -- identity
```

- Regex: This example defines a Regex algebra, including an equality based on NFAs, and the signature contains the standard Kleene operations. This has 7 components.

```
main = tacularSpec [
con "char" (Char :: Sym -> Regex Sym),
con "any" (AnyChar :: Regex Sym),
con "e" (Epsilon :: Regex Sym),
con "0" (Zero :: Regex Sym),
con ";"
 (Concat :: Regex Sym -> Regex Sym -> Regex Sym),
con "|"
 (Choice :: Regex Sym -> Regex Sym -> Regex Sym),
con "*" (star :: Regex Sym -> Regex Sym)]
```

- ListMonad: This signature contains the basic monad functions instantiated for List, as well as concatenation. This has 4 components.

```
main = tacularSpec [
 con "return" (return :: A -> [A]),
 con ">>=" ((>>=) :: [A] -> (A -> [B]) -> [B]),
 con "++"  ((++) :: [A] -> [A] -> [A]),
 con ">=>"
  ((>=>) :: (A -> [B]) -> (B -> [C]) -> A -> [C]) ]
```

- HugeLists: The benchmark from the QUICKSPEC paper, mentioned in the abstract, consisting of 33 list functions from `Prelude`, ranging from standard functions such as `length`, to more exotic functions like `(>=>)`, as well as some internal functions from QUICKSPEC like `usort` that uses a different implementation of sort. See figure 11 for

Spec	Tool	Time (s)	Memory	Laws
Lists	SPECTACULAR (P2)	3.54	21.4 MB	32
	QUICKSPEC	8.55	100.2 MB	28
	SPECTACULAR (P3)	55.44	88.7 MB	73
	SPECTACULAR (P4)	105.25	155.34 MB	93
Octonions	SPECTACULAR (P2)	0.63	19.9 MB	8
	SPECTACULAR (P3)	0.76	20.9 MB	8
	SPECTACULAR (P4)	0.92	21.1 MB	8
	QUICKSPEC	0.97	20.3 MB	15
Regex	SPECTACULAR (P2)	10.64	16.9 MB	42
	SPECTACULAR (P3)	14.83	17.2 MB	42
	SPECTACULAR (P4)	19.5	17.3 MB	42
	QUICKSPEC	458.9	88.9 MB	64
ListMonad	SPECTACULAR (P2)	0.90	15.3 MB	8
	SPECTACULAR (P3)	1.85	22.7 MB	8
	QUICKSPEC	2.44	30.5 MB	11
	SPECTACULAR (P4)	53.56	213.7 MB	42
HugeLists (3)	SPECTACULAR (P2)	0.30	11.7 MB	22
	SPECTACULAR (P3)	0.68	15.8 MB	27
	SPECTACULAR (P4)	2.12	29.2 MB	33
	QUICKSPEC	251.4	160 MB	49
HugeLists (4)	SPECTACULAR (P2)	0.73	13.8 MB	37
	SPECTACULAR (P3)	4.43	35.9 MB	68
	SPECTACULAR (P4)	91.13	125.3 MB	90
	QUICKSPEC	>3600	12.3 GB	-
HugeLists (5)	SPECTACULAR (P2)	2.92	23.4 MB	66
	SPECTACULAR (P3)	83.7	106.9 MB	144
	SPECTACULAR (P4)	>3600	301.3 MB	-
	QUICKSPEC	>3600	12.7 GB	-
HugeLists (6)	SPECTACULAR (P2)	20.70	40.2 MB	99
	SPECTACULAR (P3)	3164.75	434.4 MB	211
	SPECTACULAR (P4)	>3600	434.4 MB	-
	QUICKSPEC	-	-	-
HugeLists (7)	SPECTACULAR (P2)	201.51	54.6	186
	SPECTACULAR (P3)	>3600	383.2	-
	SPECTACULAR (P4)	>3600	381.4	-
	QUICKSPEC	-	-	-

Table I: Performance of SPECTACULAR against QUICKSPEC. Here (Pn) refers to until what phase we run SPECTACULAR. Note that we did not attempt running QUICKSPEC on HugeLists (6) and (7), due to the timeout already being hit in (5). Due to different patterns of exploration, SPECTACULAR and QUICKSPEC sometimes disagree on the number of laws. One example is in HugeLists, SPECTACULAR finds that `sort` from the prelude is the same as `usort` from the QUICKSPEC internals used in the benchmarks. SPECTACULAR thus discards any laws that mention `usort` in favor of `sort`, whereas QUICKSPEC does not, and reports additional laws involving `usort`. However `usort == sort` is not a *true* equivalence, since `usort` discards duplicates! In this case, the underlying arbitrary data generator does not generate duplicate elements so SPECTACULAR reports this as a law. This highlights the fact that care must be taken to interpret the "laws" only in the context of their generators. A timeout is denoted with (>3600), and the maximum resident memory of the process up to that point is given, e.g. 12.3 GB for QUICKSPEC on HugeLists (4). The benchmarks used are taken from QUICKSPEC, generated using the benchmark script in https://zenodo.org/record/7565011 [16].

the spec itself.

On both tools we look for terms of size up to 7, unless a particular size is specified in parenthesis next to the signature. We stopped the execution in the experiment at 3600 seconds, though on a different machine QUICKSPEC *did* finish for HugeList (4) in 38 minutes, whereas HugeLists (4) P4 took 9 seconds on the same machine.

A. *Improvements upon* QUICKSPEC

Performance: As seen in table I and figure 10, SPECTACULAR is generally faster than QUICKSPEC, and consistently using less memory. This difference is more stark when scaling the size of terms we look for and the size of the signature, as we can see in the HugeLists benchmark.

The memory improvement as seen in figure 9, in particular on the examples with larger signatures, makes it feasible to run on bigger sizes in a memory constrained environment, like cloud instances where memory rather than time is more expensive. It also shows that the limiting factor for SPECTACULAR is time and not memory requirements.

Running SPECTACULAR until phase 2 gives adequate performance, with the trade-off being fewer laws. Each subsequent phase is more expensive but often returns new kinds of laws. We also see that for the cases where polymorphism isn't present (Octonions and Regex), the performance penalty with respect to phase 2 is reasonable. A non-trivial law like `length (concat xs) == sum ((map length) xs)` (HugeLists (4)) can be discovered in < 30 seconds by SPECTACULAR but takes an hour or more in QUICKSPEC. Being able to control which parts of the type space to explore makes users of SPECTACULAR able to adapt the search to their specific requirements.

Scalability: As stated before, the biggest difference in performance happens with HugeLists. Under similar parameters to QUICKSPEC, SPECTACULAR completes its search in less time and with less memory. This gives evidence that SPECTACULAR scales better than QUICKSPEC. In particular, the memory consumption does not blow up like in QUICKSPEC. In a different test, done with a x2gd.4xlarge AWS instance[3],

[3]x2gd.4xlarge instances have 16 2.5GHz vCPUs and 256GB of RAM

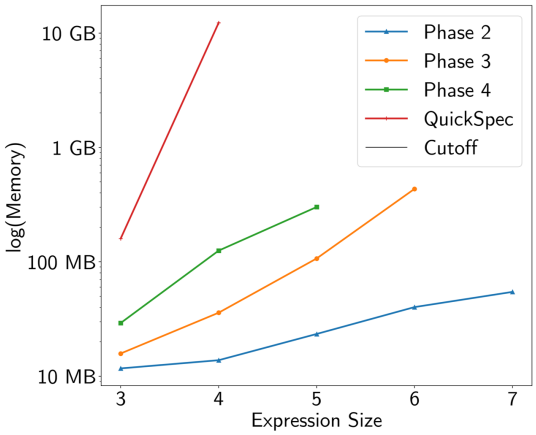


Figure 9: HugeList memory use

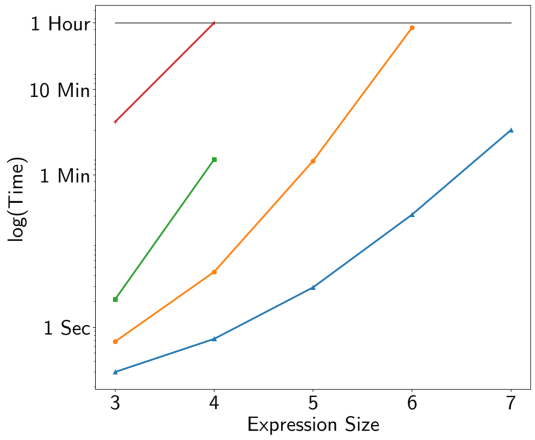


Figure 10: HugeList compute time

SPECTACULAR generated properties for HugeLists (6) within half a minute for P2. On the other hand, QUICKSPEC ran out of memory while generating terms of size 4 after 45 minutes. For P2, SPECTACULAR even manages to finish for the default term size 7, whereas QUICKSPEC does not return any properties for terms bigger than 4.

Differences in Discovered Laws: QUICKSPEC does have an advantage when it comes to the *heuristics* it uses during exploration, allowing them to quickly find laws like commutativity, associativity and distributivity, using hard-coded heuristics during the search phase and generalizing templates such as `(_ + _)` [14]. This is particularly important for specifications such as the Octonions, where these form a majority of the laws, allowing QUICKSPEC to find laws beyond the term size using these heuristics as a guide. These are laws with multiple variations like `(x*x)*y = (x*(x*y))` and `(x*y)*x = x*(y*x)`, whereas SPECTACULAR finds only `x*(x*y) = (x*x)*y`. SPECTACULAR does perform better for cases like Lists, where SPECTACULAR finds laws such as `reverse (concat xss) == concat (map reverse (reverse xss))`, `concat (concat xsss) == concat (map concat xsss)`, and `map length xss == map length (map (map f) xss)` which QUICKSPEC does not, though the difference in could be explained by the tactics used for search space enumeration and merging of laws.

V. RELATED WORK

The story of QUICKSPEC and its offshoots is often told narrowly: it extends property-based testing. But automatically discovering useful laws has far greater reach. In this section, we discuss both immediately-related work in property-based testing and program synthesis, and similar techniques used in math, physics, and other parts of computer science.

Property-Based Testing (PBT): Property-Based Testing [17] is the checking of software correctness by finding *properties* that should hold of a correct implementation and gathering evidence they hold, often by random testing [1]. It has vast research literature and multiple industrial libraries [18], [19]. In Haskell, it was popularized by QuickCheck [1], and adopted as a golden standard for testing libraries.

Synthesis of PBT Properties: The pioneer in the problem of automatically generating properties for property-based testing is QUICKSPEC [20], [14]. It has been applied to lemma discovery in automated theorem proving [21], [22], and extended with mutation-testing [23] and the ability to discover inequalities and conditional laws [24], [25]. Variants have also been implemented using comparison of symbolic rather than concrete terms [26], [27].

Data-Driven Invariant Generation: In pure functional programming, data from random testing can only be collected about the final output of a term. In imperative programming, such data can also be collected about the intermediate states of a function, and used to suggest invariants that hold at that point. This is the idea of Daikon [28].

Daikon has spawned a massive amount of follow-up research as well as three for-profit companies (most notably Agitar [29]). Of special relevance, Daikon-like techniques have been used to discover properties used in program verification, namely loop invariants [30] and simulation relations for equivalence checking [31]. Do note that Daikon-like techniques are primarily restricted to properties that hold of a single function, while property-based testing is primarily concerned with hyperproperties/hypersafety, comparing multiple programs.

Symbolic Regression: Symbolic regression [32] is the problem of finding the best mathematical formula to fit a dataset. It has been used to generate equations defining mathematical constants [33], physical laws [34], and conjectures over generalized integers [35]. Notable recent work exploits symmetry and learned features for inductive bias, discovering a great number of famous physics formulas. [36], [37].

Of less relevance to this work is the field sometimes called Automated Theorem Discovery, developing systems which propose mathematical theorems by means other than data [38], [39], [40], [41], [42].

Enumerative Program Synthesis: Both SPECTACULAR and QUICKSPEC are *enumerative program synthesizers*, em-

```
main = tacularSpec [
 con "length" (length :: [A] -> Int),
 con "sort" (sort :: [Int] -> [Int]),
 con "scanr"
   (scanr :: (A -> B -> B) -> B -> [A] -> [B]),
 con "succ" (succ :: Int -> Int),
 con ">>="  ((>>=) :: [A] -> (A -> [B]) -> [B]),
 con "snd" (snd :: (A, B) -> B),
 con "reverse" (reverse :: [A] -> [A]),
 con "0" (0 :: Int),
 con "," ((,) :: A -> B -> (A, B)),
 con ">=>"
  ((>=>) :: (A -> [B]) -> (B -> [C]) -> A -> [C]),
 con ":" ((:) :: A -> [A] -> [A]),
 con "break"
  (break :: (A -> Bool) -> [A] -> ([A], [A])),
 con "filter" (filter :: (A -> Bool) -> [A] -> [A]),
 con "scanl"
  (scanl :: (B -> A -> B) -> B -> [A] -> [B]),
 con "zipWith"
  (zipWith :: (A -> B -> C) -> [A] -> [B] -> [C]),
 con "concat" (concat :: [[A]] -> [A]),
 con "zip"  (zip :: [A] -> [B] -> [(A, B)]),
 con "usort" (usort :: [Int] -> [Int]),
 con "sum" (sum :: [Int] -> Int),
 con "++"  ((++) :: [A] -> [A] -> [A]),
 con "map"  (map :: (A -> B) -> [A] -> [B]),
 con "foldl"
  (foldl :: (B -> A -> B) -> B -> [A] -> B),
 con "takeWhile"
  (takeWhile :: (A -> Bool) -> [A] -> [A]),
 con "foldr"
  (foldr :: (A -> B -> B) -> B -> [A] -> B),
 con "drop" (drop :: Int -> [A] -> [A]),
 con "dropWhile"
  (dropWhile :: (A -> Bool) -> [A] -> [A]),
 con "span"
  (span :: (A -> Bool) -> [A] -> ([A], [A])),
 con "unzip" (unzip :: [(A, B)] -> ([A], [B])),
 con "+" ((+) :: Int -> Int -> Int),
 con "[]" ([] :: [A]),
 con "partition"
  (partition :: (A -> Bool) -> [A] -> ([A], [A])),
 con "fst" (fst :: (A, B) -> A),
 con "take" (take :: Int -> [A] -> [A]) ]
```

Figure 11: The HugeLists benchmark from QUICKSPEC we use to compare SPECTACULAR and QUICKSPEC when there are many terms in scope. Note that from this spec, SPECTACULAR also adds additional generators for the types involved, and constants such as empty lists of various types.

ploying both application-specific and standard techniques from this field. Gulwani et al [43] gives a review of this area.

VI. CONCLUSION

SPECTACULAR is an efficient tool for discovering laws for property-based testing, and has the potential to scale to settings where law discovery has previously been intractable, such as settings with generalized types. In doing so, we hope to continue to grow the usability of property-based testing. Beyond testing, the recent development of ECTA-based synthesis techniques hints at great leaps in the general usability of synthesis outside limited domains, and their simple implementation promises easy integration into more tools.

Future Work

SPECTACULAR is a recent development, and there are still many avenues to explore using SPECTACULAR.

Creating generators on-the-fly: One of the challenges for SPECTACULAR is that it is unaware of recursive generators, e.g. `Arbitrary [a] => Arbitrary [[a]]` can be derived from `Arbitrary a => Arbitrary [a]`, and so on. In the current implementation, these must be generated and made available in the ECTA for terms that require a list of arbitrary depth, e.g. `concat (concat xs) == concat (map concat xs)`, which requires a generator `xs :: [[[a]]]`. Currently, this is done by generating instances specifically for lists during initialization, and these instances added to the signature. However, integrating the available type-classes (and specifically the generators of arbitrary data) into the ECTA itself would be more efficient, as the current implementation of adding a list type for every type in the signature makes the search space a lot bigger.

Synthesizing non-equations and implications: Theory-exploration usually focuses on tautologies such as equations, but properties often only hold for a subset of the domain such as positive integers. ECTAs are good for encoding such dependencies between premises and conclusions and should excel at synthesizing such implications and other non-equations.

Efficient node-based pruning: Branch-pruning reduces the number of inspected terms by orders of magnitude, but most of the time is still spent on expanding unification variables to enumerate out the next term instead of testing of terms. But there are tree-automata algorithms that can eliminate all undesired terms before even beginning enumeration [44], which we hope to extend to ECTAs.

More directed enumeration: The ECTA-based technique allows a lot finer control over how the program space is enumerated, and the simplicity of the ECTA allows our implementation to do a lot more exploration on which branches to select in order to generate more valuable laws (i.e. more general ones first, etc.). Exploring how to control the enumeration from the outside to direct it towards parts likely to contain laws is an exciting avenue of research, and offering better enumeration heuristics could greatly speed up the current exploration. Of special interest is would be the ability to heuristically direct the enumeration towards such laws as associativity and commutativity that often hold for many data-structures.

Rapid exploration of modules: SPECTACULAR is quite efficient at generating terms in a fully monomorphized setting, even for large signatures. This could allow users to rapidly explore the properties of a module without having to manually specify which functions are interesting in a signature. One issue however is how to generate the dynamic instances and generators at runtime, though some combination of template haskell and using GHC as a library might be feasible.

Rapid on-demand generation of properties: Properties can be used to prevent overfitting in program repair, as well as help with fault-localization [45]. Properties are scarce in the wild which limits the use of property-based repairs. Being able to rapidly generate properties for a module when it is stable can be of great use in program repair to fix small bugs that might creep in during development, and can serve as a checkpoint for a current state of a module and function as regression tests, ensuring that repairs are patching the issue and not meddling with the correctness of the whole system.

Efficiency of overall pipeline: While SPECTACULAR uses an efficient data-structure for coming up with potential candidates to test, the overall efficiency of the pipeline could be improved, for example by improving the testing part by storing values to quickly reject false properties in a manner similar to QUICKSPEC [14]. For testing whole modules, determining the "interesting" parts of the modules will be important.

Applicability beyond Haskell: The speed of SPECTACULAR is highly dependent on the type-system of Haskell, which allows us to massively restrict the search space for valid terms and test only well-typed programs. In languages such as Python, the amount of "well-typed" terms becomes harder to model. However, with recent additions such as type hints, an ECTA based approach to synthesizing Python programs might be possible. In general, languages that allow random data-generation and have some constraints on the "shape" of valid terms will admit techniques similar to SPECTACULAR, though care must be taken to accurately model and/or sandbox side-effects, but this has been done for both C and Erlang [46].

ACKNOWLEDGEMENTS

We want to thank Moa Johansson and Nicholas Smallbone from the QuickSpec team for answering our questions and helping us understand its limitations, and John Hughes for his excellent feedback. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knuth and Alice Wallenberg Foundation.

REFERENCES

[1] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000, pp. 268–279.

[2] J. Koppel, Z. Guo *et al.*, "Searching entangled program spaces," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, 2022.

[3] M. P. Gissurarson, J. Koppel, E. de Vries, Z. Guo, and D. A. R. Montoya, "Tritlo/spectacular: ICST 2023," Jan. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7565003

[4] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A Theorem Prover for Program Checking," *Journal of the ACM (JACM)*, vol. 52, no. 3, pp. 365–473, 2005.

[5] G. Nelson and D. C. Oppen, "Fast Decision Procedures Based on Congruence Closure," *J. ACM*, vol. 27, no. 2, pp. 356–364, 1980.

[6] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021.

[7] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock, "Synthesizing structured cad models with equality saturation and inverse transformations," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 31–44.

[8] Y. Yang, P. Phothilimthana, Y. Wang, M. Willsey, S. Roy, and J. Pienaar, "Equality Saturation for Tensor Graph Superoptimization," in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 255–268.

[9] V. Premtoon, J. Koppel, and A. Solar-Lezama, "Semantic Code Search via Equational Reasoning," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1066–1082.

[10] J. Pollock and A. Haan, "E-Graphs Are Minimal Deterministic Finite Tree Automata (DFTAs) · Discussion #104 · egraphs-good/egg," 2021. [Online]. Available: https://github.com/egraphs-good/egg/discussions/104

[11] J. Koppel, "Version Space Algebras are Acyclic Tree Automata," 2021.

[12] O. Polozov and S. Gulwani, "FlashMeta: A Framework for Inductive Program Synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 107–126.

[13] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova, "Program synthesis by type-guided abstraction refinement," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 12:1–12:28, 2020.

[14] N. Smallbone, M. Johansson, K. Claessen, and M. Algehed, "Quick Specifications for the Busy Programmer," *Journal of Functional Programming*, vol. 27, p. e18, 2017.

[15] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons, "Fast and loose reasoning is morally correct," *ACM SIGPLAN Notices*, vol. 41, no. 1, pp. 206–217, 2006.

[16] N. Smallbone, M. Algehed, S. Maguire, K. Claessen, T. S. Kerckhove, M. P. Gissurarson, and moajohansson, "Tritlo/quickspec: ICST 2023 benchmark," Jan. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7565011

[17] G. Fink and M. Bishop, "Property-Based Testing: A New Approach to Testing for Assurance," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 4, pp. 74–80, 1997.

[18] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing Telecoms Software with Quviq QuickCheck," in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, 2006, pp. 2–10.

[19] D. R. MacIver, Z. Hatfield-Dodds *et al.*, "Hypothesis: A New Approach to Property-Based Testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[20] K. Claessen, N. Smallbone, and J. Hughes, "QuickSpec: Guessing Formal Specifications Using Testing," in *International Conference on Tests and Proofs*. Springer, 2010, pp. 6–21.

[21] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Automating Inductive Proofs Using Theory Exploration," in *International Conference on Automated Deduction*. Springer, 2013, pp. 392–406.

[22] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen, "Hipster: Integrating Theory Exploration in a Proof Assistant," in *International Conference on Intelligent Computer Mathematics*. Springer, 2014, pp. 108–122.

[23] R. Braquehais and C. Runciman, "FitSpec: Refining Property Sets for Functional Testing," in *Proceedings of the 9th International Symposium on Haskell*, 2016, pp. 1–12.

[24] ——, "Speculate: Discovering Conditional Equations and Inequalities about Black-Box Functions by Reasoning from Test Results," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, 2017, pp. 40–51.

[25] C. Smith, G. Ferns, and A. Albarghouthi, "Discovering Relational Specifications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 616–626.

[26] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva, "Automatic Synthesis of Specifications for First Order Curry Programs," in *Proceed-*

ings of the 14th symposium on Principles and practice of declarative programming, 2012, pp. 25–34.

[27] M. Alpuente, M. A. Feliú, and A. Villanueva, "Automatic Inference of Specifications Using Matching Logic," in *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, 2013, pp. 127–136.

[28] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[29] M. Boshernitsan, R. Doong, and A. Savoia, "From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing," in *Proceedings of the 2006 international symposium on Software testing and analysis*, 2006, pp. 169–180.

[30] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A Data Driven Approach for Algebraic Loop Invariants," in *European Symposium on Programming*. Springer, 2013, pp. 574–592.

[31] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-Driven Equivalence Checking," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 391–406.

[32] M. Schmidt and H. Lipson, "Distilling Free-Form Natural Laws from Experimental Data," *science*, vol. 324, no. 5923, pp. 81–85, 2009.

[33] G. Raayoni, S. Gottlieb, Y. Manor, G. Pisha, Y. Harris, U. Mendlovic, D. Haviv, Y. Hadad, and I. Kaminer, "Generating Conjectures on Fundamental Constants with the Ramanujan Machine," *Nature*, vol. 590, no. 7844, pp. 67–73, 2021.

[34] P. Langley, "Data-Driven Discovery of Physical Laws," *Cognitive Science*, vol. 5, no. 1, pp. 31–54, 1981.

[35] H. Ferguson, D. Bailey, and S. Arno, "Analysis of PSLQ, An Integer Relation Finding Algorithm," *Mathematics of Computation*, vol. 68, no. 225, pp. 351–369, 1999.

[36] S.-M. Udrescu and M. Tegmark, "AI Feynman: A Physics-Inspired Method for Symbolic Regression," *Science Advances*, vol. 6, no. 16, p. eaay2631, 2020.

[37] S.-M. Udrescu, A. Tan, J. Feng, O. Neto, T. Wu, and M. Tegmark, "AI Feynman 2.0: Pareto-Optimal Symbolic Regression Exploiting Graph Modularity," *Advances in Neural Information Processing Systems*, vol. 33, pp. 4860–4871, 2020.

[38] D. B. Lenat, "Automated Theory Formation in Mathematics," in *IJCAI*, vol. 77. Citeseer, 1977, pp. 833–842.

[39] D. B. Lenat and J. S. Brown, "Why AM and EURISKO Appear to Work," *Artificial intelligence*, vol. 23, no. 3, pp. 269–294, 1984.

[40] R. L. McCasland and A. Bundy, "MATHsAiD: A Mathematical Theorem Discovery Tool," in *2006 Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2006, pp. 17–22.

[41] R. McCasland, A. Bundy, and S. Autexier, "Automated Discovery of Inductive Theorems," *Special Issue of Studies in Logic, Grammar and Rhetoric on Computer Reconstruction of the Body of Mathematics: From Insight to Proof: Festschrift in Honor of A. Trybulec*, vol. 10, no. 23, pp. 135–149, 2007.

[42] S. Colton, A. Bundy, and T. Walsh, "On the Notion of Interestingness in Automated Mathematical Discovery," *International Journal of Human-Computer Studies*, vol. 53, no. 3, pp. 351–375, 2000.

[43] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program Synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[44] M. D. Adams and M. Might, "Restricting grammars with tree automata," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–25, 2017.

[45] M. P. Gissurarson, L. Applis, A. Panichella, A. van Deursen, and D. Sands, "PropR: Property-Based Automatic Program Repair," in *The 44th IEEE/ACM In-ternational Conference on Software Engineering (ICSE)*. IEEE/ACM, 2022.

[46] J. Hughes, "Experiences with quickcheck: testing the hard stuff and staying sane," in *A List of Successes That Can Change the World*. Springer, 2016, pp. 169–186.