Don't Go Down the Rabbit Hole: Reprioritizing Enumeration for Property-Based Testing

Segev Elazar Mittelman
segevem@umd.edu
University of Maryland
USA

Aviel Resnick
avielr@seas.upenn.edu
University of Pennsylvania
USA

Ivan Perez
ivan.perezdominguez@nasa.gov
KBR Inc @ NASA Ames Research Center
USA

Alwyn E. Goodloe
agoodloe@nasa.gov
NASA
USA

Leonidas Lampropoulos
leonidas@umd.edu
University of Maryland
USA

Abstract

Combinatorial testing is a popular software engineering technique for effectively testing programs that operate on parameters drawn from small, finite domains (such as configuration options), by providing a principled way of systematically exploring interactions between them. Property-based testing, on the other hand, is a widely used technique for testing the correctness of functional programs that usually operate on large, infinite domains (such as algebraic data types), by randomly generating or enumerating inputs.

In this paper, we show how to extend and apply ideas from the narrow scope of combinatorial testing techniques to the broader scope of property-based testing applications. In particular, we develop a novel way of pruning the input search space while still ensuring that a diverse set of constructor patterns appear among the set of generated tests. In our implementation, we integrate a state-of-the-art enumeration-based property-based testing framework, LazySearch, with a state-of-the-art combinatorial testing tool, NIST's ACTS, and demonstrate how it can significantly speed up the effectiveness of testing—up to more than 20× in the case of a prior System F case study from the literature.

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

Keywords: property-based testing, combinatorial testing, enumeration, generation, functional programming

Haskell '23, September 8–9, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0298-3/23/09.
https://doi.org/10.1145/3609026.3609730

ACM Reference Format:
Segev Elazar Mittelman, Aviel Resnick, Ivan Perez, Alwyn E. Goodloe, and Leonidas Lampropoulos. 2023. Don't Go Down the Rabbit Hole: Reprioritizing Enumeration for Property-Based Testing. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium (Haskell '23), September 8–9, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3609026.3609730

1 Introduction

Suppose that you are working on the design and implementation of a new typed functional language. During this development you will most likely need to establish confidence in the metatheoretic properties of such a system. For instance, consider *preservation*, which states that the type τ of a term e in any given context Γ is invariant under a single evaluation step:

```
preservation Γ e τ =
  hasType Γ e τ ==> hasType Γ (step e) τ
```

How would one go about testing such a property? *Property-based testing* offers a promising solution: generate many inputs, repeatedly execute `preservation`, and check if any of the inputs invalidate it. Naturally, the success of this process hinges on how the inputs are generated.

A natural way of generating such inputs is to use randomness: generate random contexts, expressions, and types; filter out all generated configurations which violate the well-typedness precondition; and then check that, for those that remain, performing an evaluation step does not yield an ill-typed term. Unsurprisingly, such a generate-and-filter approach is not very effective: a lot more configurations are ill-typed than not.

The standard solution to this so-called *precondition problem* is to *handwrite* a generator that produces well-typed terms directly. Unfortunately, writing such generators can be extremely challenging. For example, constructing an effective generator for well-typed lambda terms for each of

simply typed lambda calculus (STLC) [18], STLC with effects [15], or System F [7] constituted a significant research contribution in itself. More recent solutions to the same problem attempt to use automatic methods for obtaining such generators, but they usually rely on language features not available in mainstream languages, such as specifications written in the form of inductive relations in Coq [14].

An alternative way of generating inputs is to systematically enumerate the elements of the input space. While an enumerate-and-filter approach is also ineffective for precisely the same reason, enumeration opens up a particularly interesting avenue of addressing the precondition problem: Claessen et al. [2] leverage laziness to prune large parts of the search space at once, using the structure of the precondition itself to guide the generation process.

For concreteness, let's turn back to our running example of testing preservation and consider `Terms` that contain at least `Abstractions`, `Applications`, (DeBruijn-based) `Variables`, and `Constants`:

```
data Term = Abs Term | App Term Term
          | Var Nat  | Const Int
          | ...
```

If we were to follow a naive enumerate-and-filter approach, many of the syntactically well-formed lambda terms generated would be filtered out for being ill-typed. For instance, all of the terms in Fig. 1 are ill-typed, as they are applications of a non-functional value (the constant `0`).

The main observation behind Claessen et al. [2] is that the entire space of terms of a similar form on the left—applications of an integer constant—could be pruned out by observing that

```
App (Const undefined) undefined
```

can be classified as ill-typed in a lazy language like Haskell without yielding an error. This approach is currently implemented in the LazySearch [4] library and shows significant promise. For example, as we will see in Section 5, in a case study from the literature in testing preservation for System F [6], LazySearch finds all bugs (although it takes several minutes for some of them) where a naive enumerate-and-filter with something like SmallCheck is incapable of finding almost anything [23].

Still, just like all enumeration-based approaches, the effectiveness of LazySearch greatly depends on the enumeration order [16]. Turning again to our running example, although all the terms in Fig. 2 are well-typed, they are extremely similar: they are applications of an identity function to some other term. As a result, they will most likely not lead to interestingly different execution behaviors. As a result, enumeration can spend a lot of time in a particular (and uninteresting) part of the search space before exploring new ones. And sometimes that's acceptable! If, for example, we had enough time to exhaust the entire search space, then enumeration order is not as important of a concern.

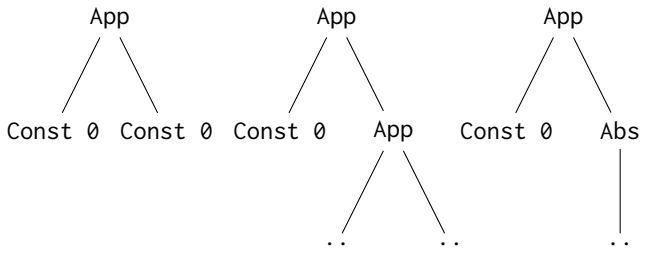

Figure 1. Terms that are ill-typed.

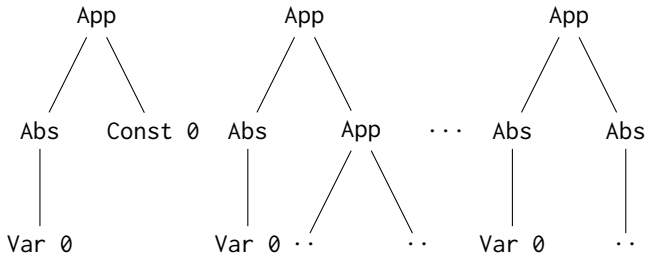

Figure 2. Terms that are well-typed.

Sadly, that is not always the case. Testing time is finite and, even in our running example, the search space of well-typed terms is too large to exhaust. Ideally, we could use *fair* and compositional enumeration combinators to write enumerators that will, by default, provide some guarantees that we will not ignore parts of the input space for too long [16].

Unfortunately, such techniques don't play well with automatic approaches in the style of LazySearch: under the hood, LazySeach constructs an indexing function from natural numbers to elements of an algebraic datatype (such as lambda terms), and can potentially prune, using laziness, ranges from the numeric domain that correspond to an entire space of values in the codomain. However, at the same time, the flat numeric domain of this mapping obscures the recursive structure that is required to use explicit fair combinators.

Is there any hope? *Given a finite amount of testing time, can we systematically avoid falling down the rabbit hole of fully exploring valid but largely homogeneous subspaces before visiting others?* In this paper, we provide such a solution using ideas from *combinatorial testing*.

Combinatorial testing [17] is a software engineering technique for testing software with multiple configuration options [11], by providing a principled way of exploring interactions between such options. However, out-of-the-box combinatorial testing tools (such as ACTS [10]) can presently only be used to test programs that operate on variables drawn from small, finite domains.

Our work bridges the gap between such tools and leverages their power to test functional programs that usually operate on potentially infinite algebraic data types with multiple constructors. In particular, our contributions are:

- We expand the applicability of out-of-the-box combinatorial testing techniques to functional languages, describing a new notion of combinatorial coverage for algebraic data types, and use these techniques to improve upon LazySearch [4] (Section 3).
- We propose Radix Coverage, a technique for increasing the applicability and efficiency of combinatorial testing tools for tests with parameters spanning drastically larger ranges than state-of-the-art software can currently handle (Section 4).
- We evaluate our techniques on two case studies:
  1. We first turn to the Traffic Collision Avoidance System (TCAS), a piece of avionics software used historically to benchmark combinatorial testing procedures [19]. We show that Radix Coverage offers a viable avenue to blackbox-testing in this setting, reducing testing times from multiple hours down to mere minutes—where such speedups were only possible by manually cutting down the search space using expert domain-specific knowledge (Section 5.1).
  2. The second case study focuses on applying Radix Coverage in a functional setting, by testing a Haskell implementation of System F from the literature [6]. When compared to LazySearch, our implementation provides significant benefits on harder-to-find bugs, reducing time to failure up to more than 20× (Section 5.2).

2 Combinatorial Testing

We begin by presenting some background information on combinatorial testing and its purpose, as well as describing a state-of-the-art tool, ACTS (Automated Combinatorial Testing for Software), that we will leverage later in the paper.

2.1 Combinatorial Testing

Combinatorial testing is a software engineering technique for efficiently testing a system that has many possible input configurations. Going back to our running example of developing a programming language, consider how one could test whether compiler optimizations for it preserve the original semantics of a program. Such optimizations are often bug-riddled, and the situation becomes especially bug-prone when multiple optimizations are enabled simultaneously [11]. For concreteness, suppose we have 5 toggleable program transformations: lambda lifting (*ll*), stream fusion (*sf*), common subexpression elimination (*cse*), specialization (*sp*) and function inlining (*inl*):

```
optimize :: Bool   -- ll
         -> Bool   -- sf
         -> Bool   -- cse
         -> Bool   -- sp
         -> Bool   -- inl
         -> Prog   -- p
         -> Prog
optimize ll sf cse sp inl p = ...
```

Let's also assume, for now, that we already have some test suite of programs and we are only focusing on interactions between optimizations—we will return to the problem of generating the programs themselves later in the paper. How can we ensure that our optimizer behaves correctly? One naive way would be to ensure that all $2^5 = 32$ combinations of optimizations preserve the semantics of the programs in our test suite. However, such an approach quickly becomes intractable as we add more and more optimizations.

Combinatorial testing offers an attractive alternative: rather than covering *all* interactions between such optimizations, we could cover all *t-way* interactions between them. For example, we could test all 2-way interactions with just the following 6 configurations shown in Table 1. Indeed, for *any* pair of optimizations, any distinct toggling of the pair can be found in some row of the table!

Table 1. Two-strength covering array for optimization toggling.

	ll	*sf*	*cse*	*sp*	*inl*
1	True	True	False	False	False
2	True	False	True	True	True
3	False	True	True	False	True
4	False	False	False	True	False
5	False	True	False	True	True
6	False	False	True	False	False

More generally, combinatorial testing weakens the notion of full coverage, going from testing every possible combination of values of *all* input parameters, to exhaustively testing every possible combination of values of *each t-sized subset of input parameters*, for some small number t.

In exchange for this weakening, t-way coverage combinatorial testing achieves comparable efficacy in terms of bug finding with substantially fewer test cases. Each instantiation of a t-sized subset of input parameters is termed a *t-strength interaction*. Each full input configuration of all n parameters then covers one t-strength interaction for each of the $\binom{n}{t}$ subsets. If it was possible for each input configuration to cover a unique t-strength interaction for all $\binom{n}{t}$ subsets, and if each input parameter spanned v possible values, then, in only v^t tests, full t-strength coverage would be obtained (as opposed to the v^n tests required for traditional full coverage).

While this uniqueness property is not possible in practice, the actual number of required tests factoring in the need

to repeat interactions is only worse by a factor logarithmic in the total number of parameters in the input configuration. When compared to, v^n, the number of all possible input combinations under exhaustive coverage, t-strength combinatorial coverage's reduction to only $O(v^t log(n))$ test cases represents a significant improvement. One might argue that the t-strength combinatorial coverage of a test suite is a poor measure of its bug-finding efficacy, but it has been empirically shown in actual software [11] that the majority of bugs are found by strength 3 and there are diminishing returns approaching 100% bugs found as the strength approaches 6.

2.2 ACTS

How does one arrive at the table above? To construct an efficient t-strength combinatorial coverage test suite, we can turn to ACTS [10], a tool produced by the National Institute of Science and Technology (NIST) which automatically constructs *combinatorial covering arrays* when given a specification of a system's configuration parameters and a desired interaction strength. The rows of a covering array each represent one concrete configuration to test, with the columns of the covering array each holding values associated with the parameter they are labeled by. Using the specification, ACTS constructs an array while minimizing the total number of rows, and therefore test cases, required to obtain the desired strength of combinatorial coverage.

To construct a covering array for our system, one could handwrite the ACTS specification shown in Figure 3. The `[Parameter]` section specifies the name, type, and values allowed for a given parameter. The `[Relation]` section lists the relations which determine the interaction strength of the parameters in the covering array. For our case, a single relation including all the parameters in the array and given a strength t, ensures that the covering array has interaction strength t. Note that t should be replaced with a value from 1 to the number of parameters being related, as there cannot be an interaction larger than the number

```
[System]
Name: test

[Parameter]
ll  (boolean) : True,False
sf  (boolean) : True,False
cse (boolean) : True,False
sp  (boolean) : True,False
inl (boolean) : True,False

[Relation]
R1 : (ll, sf, cse, sp, inl, t)

[Constraint]
```

Figure 3. ACTS specification of combinatorial configuration.

of variables that are interacting. The `[Constraint]` section allows one to list simple restrictions on the values of variables. For instance, if the specialization optimization was known to be incongruent with function inlining, the constraint `!((specialization && inlineFunctions))` could be listed under the `[Constraint]` section.

The first step to bridging the gap between combinatorial testing and high-level property-based testing is to provide a high-level API to access ACTS from a language like Haskell (Figure 4). To this end, we provide two functions: The function `constructTestArrayValues`, which, given a test specification, calls ACTS and provides a list of input configurations – the combinatorial covering array. The function `mkTest`, which is a smart constructor that builds a test specification for a given predicate, list of constraints, list of variables (including types, names, and values), and intended interaction strength. Continuing our running example, we also require the property we wish to test, `optimizationCorrect`, which asserts that the combination of enabled optimizations preserves the semantics of the interpreter.

3 To Infinity And Beyond

Combinatorial covering arrays allow for efficiently testing configurations of small-range parameters like the boolean toggles from the optimization configuration. But what about parameters with larger domains? What about parameters whose domain is infinite—such as the well-typed terms from the introduction? Naturally, a covering array cannot contain such infinite-range parameters, as it would require infinite rows to cover. A different approach is therefore necessary.

So what can we do instead? Rather than trying to store an infinite numbers of concrete values in a covering array, we will *partition* the entire space of values into a finite number of distinct (potentially overlapping) infinite subspaces in a way that can be recursively applied to further partition each subspace. Algebraic data types offer a natural such partitioning strategy: let the constructors dictate it. Then, we will leverage combinatorial testing to ensure that all interactions between constructors are exercised when testing. To make things more concrete, let's return to our running example of testing a programming language's design and implementation.

Partitioning via Example. Consider the datatype of terms for our language, which has two recursive constructors, `Abs` and `App`, and two base-case constructors `Var` and `Const`.

```
data Term = Abs Term | App Term Term
          | Var Nat  | Const Int
```

We can construct a reasonable single-level partitioning of the space of `Terms` by partitioning based on the top-level constructor, the root of the `Term`'s constructor tree. Because

```
type Strength = Int

constructTestArrayValues :: Read a => Test a -> IO [a]
constructTestArrayValues = -- Defined elsewhere

optimizationTest :: Strength -> Test (Bool, Bool, Bool, Bool, Bool)
optimizationTest strength = mkTest predicate constraints inputVariables strength
  where
    predicate = optimizationCorrect
    constraints = []
    inputVariables = map boolParam ["ll", "sf", "cse", "sp", "inl"]

    boolParam name = Param (PBool name [True, False])

>>> constructTestArrayValues (optimizationTest 2)
[(True ,True,False,False,False),(True ,False,True ,True ,True ),
 (False,True,True ,False,True ),(False,False,False,True ,False),
 (False,True,False,True ,True ),(False,False,True ,False,False)]
```

Figure 4. Example of Radix Coverage test specification.

we discriminate based on a single constructor, we can define an *indexing type I* as follows.

```
data I = Abs' | App' | Var' | Const'
```

`Abs'` is the index for the infinite space of `Terms` whose root constructor is an abstraction, whereas `Var'` is the index for the infinite space of `Terms` which are just free variables, because `Var` as a root constructor implies no more recursive `Terms`. Using this indexing type, we can define a characteristic function which checks whether a given `Term` is inside the partition with a given index in `I`.

```
member :: I -> Term -> Bool
member Abs'   (Abs _  ) = True
member App'   (App _ _) = True
member Var'   (Var _  ) = True
member Const' (Const _) = True
member _      _         = False
```

This serves perfectly fine as a definition of the first-level partition, but, in order to construct a second-level partition, we need to inspect not only the root constructor of the overall `Term` but also the root constructors of the direct recursive children of the `Term` as well, in order to construct the subpartitions in a self-similar fashion. Generally, to construct a partition of depth f (which we call call the *fineness* of the partition), we need to inspect down to the f-depth constructors in the `Term` tree. A more general version of `member` therefore needs to accept a list of `I` as follows:

```
member :: [I] -> Term -> Bool
member (Var'   : is) (Var _  ) = True
member (Const' : is) (Const _) = True
member (Abs'   : is) (Abs e  ) =
  member is e
member (App'   : is) (App e1 e2) =
  member is e1 || member is e2
member []            _         = True
member _             _         = False
```

Examining the behavior of this function, and in particular the disjunction in the `App'` case, we see that a term is a member of the subspace indexed by a list of indices if there is a path of nested constructors, starting from the root of the term's constructor tree, that matches the list's sequence of constructor indices. Any and all indices following an index corresponding to a terminal constructor are disregarded. All other constructors in the term besides those matching the path are left unconstrained by the membership predicate. Therefore, each list of indices defines an entire space of structurally constrained terms. Allowing terminal constructors to appear at any point in the indexing list sacrifices injectivity; `[App',Const',App']` and `[App',Const',Const']` map to the same space, leading to repeated testing of the same input space. However, doing so embeds all lower fineness spaces into higher fineness spaces which allows multiple fineness levels to be tested in the same run.

So where does combinatorial testing come into play? If we were to partition the space of terms into 4^f subspaces of fineness f, and then explore all of them, we would be back to square one: attempting to exhaustively explore a vast space. What if, instead, we were to focus on *interactions* between indices constraining the choices of constructors at different nesting depths. We could build a minimal set of subspaces such that, for some *strength* $t < f$, and for every t-sized subset of indices at distinct nesting depths, each possible

configuration of constructor indices at the specified depths is contained in at least one row of the resulting covering array.

For example, such a partition with fineness of $f = 4$ and strength of $t = 2$ is shown in Table 2 consisting of only 20 rows. Each such row represents one of the $4^f = 256$ structural 4-depth patterns which fully partition the `Term` type.

Table 2. Covering array for `Terms` ($f = 4, t = 2$).

	I1	I2	I3	I4
1	Abs'	Abs'	App'	App'
2	Abs'	App'	Var'	Var'
3	Abs'	Var'	Const'	Const'
4	Abs'	Const'	Abs'	Abs'
5	App'	Abs'	Var'	Const'
6	App'	App'	Const'	Abs'
7	App'	Var'	Abs'	App'
8	App'	Const'	App'	Var'
9	Var'	Abs'	Const'	Var'
10	Var'	App'	Abs'	Const'
11	Var'	Var'	App'	Abs'
12	Var'	Const'	Var'	App'
13	Const'	Abs'	Abs'	Var'
14	Const'	App'	App'	Const'
15	Const'	Var'	Var'	Abs'
16	Const'	Const'	Const'	App'
17	Var'	Abs'	Const'	Abs'
18	Abs'	App'	Var'	App'
19	Var'	Var'	App'	Var'
20	Var'	Const'	Abs'	Const'

Indeed, pick *any* 2-wide configuration of indices. Then at least one row of the array will exhibit that pattern at every possible depth! For instance, the configuration `[App', Abs']` is covered by rows 5 (App' at depth 1, Abs' at depth 2), 6 (App' at depth 1, Abs' at depth 4, and App' at depth 2, Abs' at depth 4), 7 (App' at depth 1, Abs' at depth 3), 10 (App' at depth 2, Abs' at depth 3), and 11 (App at depth 3, Abs at depth 4), ensuring interactions between these constructors will be present at all possible depths in *at least* 1 subspace in the resulting partition.

Combinatorial Testing for ADTs. None of the steps taken to construct the example above made use of some unique feature of `Term`, so the same technique straightforwardly generalizes to create covering arrays for f-fine partitions on any simple algebraic data type: Take I to be the set of constructor tags for the data type; define membership in the f-fineness partition for a list of indices of length f to mean that there must be some path from the root down f constructors so that the constructors on the path match those listed as constructor tags in the indexing list; create a covering array for interactions between constructors using an off-the-shelf tool like ACTS.

Back to our original problem, however, of testing a property such as `preservation`, how do we generate data in a given subspace of a partition? That's where LazySearch comes in. Each row r in the covering array can be interpreted as a list of indices, and therefore gives rise to a particular instantiation of the (purely structural) `member` predicate, which is an *excellent* candidate for LazySearch's lazy pruning algorithm.

For example, to test `preservation` we can:

1. Construct a partition of the data type of terms based on its constructors.
2. Conjoin the precondition of the property (`hasType`) with the membership predicate corresponding to each row in the array.
3. Run LazySearch with the combined precondition, enumerating the space of terms that both satisfy the original precondition and match the constructor pattern dictated by the array row to search for a counterexample.

Naturally, we would also like to avoid staying in a particular subspace indefinitely or for too long. Therefore, we also give a fixed time allotment to testing each pattern generated in the array, thus ensuring that the enumerative testing procedure explores a diverse range of top-level constructor patterns, thus increasing the likelihood of finding bugs within a limited amount of time.

4 Large-Range Parameter Testing with Radix Coverage

The approach described in the previous section can be very effective (see Section 5), but using an out-of-the-box version of ACTS for the generation of the combinatorial array comes with a drawback: NIST's Practical Combinatorial Testing guide states that the reasonable upper limit on the number of values per parameter is between 8 and 10, and going any higher would make covering array generation time infeasibly long [22, 24]. For algebraic data types, that means that our approach cannot efficiently handle types with more than ~10 constructors. Moreover, this becomes a bigger problem in numeric domains: for instance, combinatorial testing has long been used by NIST to test avionics software such as the traffic collision avoidance system (TCAS), and some of that system's parameters range over three-digit natural numbers. The current solution? Use white-box knowledge of the system to pick a small number of values within that range to cover.

Can we do better? Yes! We can break up a single large-range parameter into several smaller-range parameters, each with 10 or fewer values. Then, any given concrete covering array contains values for all such subparameters and we

can use these values to reconstruct a value for the original large-range parameter.

To keep things concrete, let's focus on integer parameters, such as those of TCAS, and suppose they range from 0 to 999. If we were seeking pairwise combinatorial coverage between such a parameter and another boolean one (which consists of two values), a complete covering array would consist of $1000 * 2 = 2000$ rows, the first few of which are depicted on the left side of Table 3.

If, on the other hand, we considered the natural split of the integer parameter into three subparameters (one for each of the original parameter's digits) and then pursued pairwise coverage between all four parameters (three digits and one boolean), the resulting covering array would only have 100 rows, the first few rows of which are presented in the right side of Table 3. Because the interaction strength of the array is two (which is less than the number of ways we partitioned the integer parameter), not every number in the original range will appear, leading to a drastic reduction in the number of rows.

For example, consider how the first row has the combined integer parameter as 001 but the next row skips all the way to 012, skipping ten numbers in between. This is because while each pair of digits needs to appear in the resulting array, not every triple of digits does. Concretely, the second row is not 002 as the covering array generation seeks to pack as many unseen pairwise interactions into each new row as possible. 001 already covers 002's 00x interaction, and 012 would cover the new 01x interaction in addition to the 0x2 that 002 would provide. While this reduction in rows is beneficial in terms of reducing required testing time, 90% of the intended range of integers to test over are skipped, which is far from ideal.

We can rectify this issue of dropping large intended portions of the testing data's range by increasing the interaction strength *within* the three subparameters while keeping the overall strength to 2. This would make the digits fully interact, ensuring that each element of the integer parameter's range is found in at least one row. This increases the number of rows in the covering array to 1000 (appearing in the middle of Table 3, still reducing the 2000 rows needed for full coverage by half and effectively lowering coverage and therefore saving testing time, but without cutting out any of the intended test data. Note that since each number only appears once, either True or False but not both can be paired with each value, differing from full coverage as displayed on the left in the same table.

This property that every value of the original range must appear in at least one row of the generated covering array is desirable as a counterweight to the effective loss in interaction strength between parameters split into subparameters and the other parameters in the configuration as shown in Section 5.1. In general, if we split a parameter into p subparameters, we would need to ensure that its set of subparameters is related with interaction strength p, on top of the interaction strength of the overall covering array. Each parameter being split would require its own local interaction strength. Thankfully, such a *mixed-strength* coverage is already supported by ACTS.

As described so far, this technique of splitting combinatorial coverage array parameters into subparameters shows promise, but it has two flaws. First, recall that ACTS can only guarantee up to interaction strength 6, with only experimental support for higher strengths. So, if the range of a variable exceeds a million values, this method does not guarantee that all members of the range will be represented in the array. To solve this issue, we can increase the base that we write a number in, requiring fewer digits to express it. Therefore, instead of splitting a range of size k into $\lceil \log_{10}(k) \rceil$ digit subparameters with values $[0, 1, \ldots 9]$, we split the range into $\lceil \log_b(k) \rceil$ subparameters with values $[0, 1, \ldots b-1]$, with b chosen such that the number of base-b digit subparameters does not exceed 6. In practice, we choose the smallest such b because having more total parameters increases the array construction complexity only logarithmically, whereas increasing the total number of values per parameter increases complexity polynomially.

Second, consider what happens when k, the size of the integer parameter range, is not a simple power of the base b. In that case, we have two choices: only generating values up to the largest power of b smaller than k, which would leave out a potentially significant chunk of the range, or generating values up to the smallest power of b greater than or equal to k, which would include many undesired values in the final covering array that we may not know how to process. To solve this, we simply add the following constraint to the covering array generation which forbids the reconstruction of the subparameters to exceed the maximum value of the range, where v_i are the subparameter names, supposing there are p subparameters total:

```
maximumValue .>=. parameterValue

where
  parameterValue =
    foldr foldop (val 0) digits
  digits = [v_{p-1}, v_{p-2}..v_0]

  foldop v_i acc =
    numVar v_i .+. (val b .*. acc)
```

This iteration works on contiguous ranges of natural numbers, but what of large ranges? One could theoretically construct a unique method of splitting up each type that balances type-unique considerations about maximizing interaction strength and other desirably qualities, but our goal is to construct an automatic method. Given any finite range of any values, notice there's a bijection between it and the finite range of naturals from 0 to one less than the size of the range

Table 3. First few rows of a full combinatorial testing covering array (left), Radix Coverage array (middle), and lower-mixed strength covering array (right).

Int	Bool	100s	10s	1s	Bool	100s	10s	1s	Bool
0	False	0	0	0	False	0	0	1	False
0	True	0	0	1	True	0	1	2	True
1	False	0	0	2	False	0	2	3	False
1	True	0	0	3	True	0	3	4	True
2	False	0	0	4	False	0	4	5	False
2	True	0	0	5	True	0	5	6	True
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

by indexing into the range. In the following, we leverage this idea to provide a final definition of our approach, which we name Radix Coverage.

Radix Coverage. Radix Coverage constructs covering arrays for large but finite range parameters of any type. It uses a contiguous range of natural numbers as an index into the large range, splits the large natural number parameter into several mutually interacting variables in an ideal base, or radix. These subparameters use mixed coverage to ensure they fully interact, so that each value in the range appears in at least one row of the covering array. These variables are constrained to prevent their overall value exceeding the size of the range. Radix Coverage automatically recombines the digits from the constructed arrays into indices and obtaining the test values by indexing into the original range of test values.

5 Evaluation

To evaluate Radix Coverage and its extension to recursive algebraic data types, we present two case studies—one for each technique. Both case studies make use of mutation testing [9] to compare the efficacy of our approaches against baseline results from the tools which they augment. Mutation testing is a technique for comparing the quality of various testing approaches, by artificially injecting bugs, called "mutants", in the system under test and evaluating how many are found, or "killed", and how quickly.

5.1 TCAS (Traffic Collision Avoidance System)

For our first case study, we turn to the Traffic Collision Avoidance System (TCAS), which has long been used to benchmark the efficiency of combinatorial testing methods [19].

Previous studies using TCAS as an evaluation metric relied on a standard set of 38 realistic, distinct mutations that aim to capture common programming errors. Our implementation mirrors this set of mutations, which includes logical errors (e.g. $\leq$ instead of $<$), incorrect arithmetic operators, inverted logical branches, missing conditions, and incorrect constants. TCAS takes as input 12 parameters regarding flight and location data including: 3 boolean inputs, 3 enumeration types with either 3 or 4 values apiece, and 6 integer inputs. In return, TCAS produces one recommendation to maintain a safe distance between two aircraft.

To evaluate its efficacy and efficiency, Radix Coverage was tested at varying interaction strengths ($t \leq 6$) and choices for integer variable ranges to find the implanted mutations in TCAS. We simulated three variable range scenarios (for each of the TCAS integer variables). To represent a whitebox testing environment following prior work , one simulation used sets of less than 10 handpicked values for each variable, where the values were chosen knowing the likely boundary cases of the program. The other two simulations we chose represent a more blackbox testing environment, where the intended critical ranges of each variable are unknown to the tester, and therefore larger ranges are required to stand a reasonable chance of finding bugs. The first gave all 6 integer parameters a range of size 101, starting from 0 and incrementing by 10 up to 1000 to do a scattered search with an intermediate sized range. The second gave all integer parameters a range of size 1001 consisting of all numbers from 0 to 1000 to do a more exhaustive, consecutive-range search.

For each setting of interaction strength and variable range choice, we additionally generated baseline results in terms of time-to-counterexample by using plain ACTS to find the bugs. Given that Radix Coverage is an augmentation of ACTS, this baseline demonstrates how much of an improvement the technique provides. These experiments were ran on a machine with 4 double-threaded Intel Core i7 cores running at 1.9GHz.

RadixCov vs. Whitebox ACTS. Considering the first white-box scenario of hand picked ranges consisting of ten integers, ACTS outperforms Radix Coverage in number of bugs found, but greatly sacrifices memory usage as interaction strength increases. At $t = 6$, Radix Coverage is able to

Table 4. Black-Box $[0 \ldots 1000]$ Integer Ranges.

	t = 2		t = 3	
	Runtime	Bugs Found	Runtime	Bugs Found
Radix Coverage	2.53 Minutes	26	53.23 Minutes	28
ACTS	46.29 Hours	18	DNF	DNF

Table 5. Black-Box $[0, 10, \ldots 1000]$ Integer Ranges.

	t = 2		t = 3		t = 4	
	Runtime	Bugs Found	Runtime	Bugs Found	Runtime	Bugs Found
Radix Coverage	45.6 Sec.	9	148.1 Sec.	30	25.98 Min.	35
ACTS	102.71 Sec.	17	76.58 Hours	35	DNF	DNF

find 26 bugs compared to ACTS finding 35 bugs in the same period of time, with ACTS requiring 7.3 times as much memory. However, Radix Coverage excels at black-box use with large integer ranges, which greatly increases its applicability in scenarios with limited knowledge of critical and boundary value ranges for the software's input parameters.

RadixCov vs. Blackbox ACTS. The second scenario uses basic physical intuition of the problem to give a contiguous 1001 value range for each of the six integer parameters. The inputs are distances, so they should be non-negative, and they deal with collision, so 1000 feet apart seems like a reasonable maximum. Testing both Radix Coverage and plain ACTS against all 38 mutations of TCAS yields the results in Table 4.

Not only does Radix Coverage far outperform ACTS in terms of bugs found in this blackbox scenario, the improvement in runtime to discover counterexamples is staggering. By providing a combinatorial testing method that can handle several parameters with ranges exceeding 1000 values in minutes, Radix Coverage turns combinatorial testing into a practical blackbox testing technique.

RadixCov vs. Blackbox ACTS in scattered domain. The third scenario further explores black-box testing by taking the large range from the second scenario and leaving only a tenth of its values, the 101 multiples of 10 from 0 to 1000, as choices for each of the 6 integer variables TCAS uses. Using these ranges as input to both Radix Coverage and ACTS produces the results contained in Table 5.

Once again, we see that Radix Coverage far extends the feasible size of integer ranges beyond what is possible with ACTS. Past strength 2, these large ranges required for black-box testing become intractable for ACTS, as the test arrays become overly large. Radix Coverage on the other hand handles well through strength 4. One downside of Radix Coverage is that due to the variable splitting technique at its core, the effective interaction strength of the generated covering array is lower, as variables are only interacting with parts of split variables, not with their entirety. From this data, the bug-finding efficacy seems to indicate that there is an effective drop of 1 in interaction strength when compared to plain ACTS, as Radix Coverage finds the same number of bugs at interaction strength t as ACTS does with interaction strength $(t - 1)$. Fortunately, this loss is easily compensated by how much quicker Radix Coverage handles higher interaction strengths than ACTS.

Overall, the TCAS case study shows that Radix Coverage has the ability to reduce operations that require days of computation to mere minutes, and can facilitate true black-box testing of software while leveraging combinatorial methods.

5.2 System F

To evaluate our extension of combinatorial testing to handle recursive algebraic data types, we turn to a mutation testing based case study from the literature [6] that consists of a System F implementation, as well as a suite of bugs (mostly dealing with DeBruijn indices). In particular, the property being tested is very similar to the running example we've been using throughout the paper, in that it has a well-typedness precondition. More concretely, the property states that for all well-typed terms e and for all mutants m, both a single step and a parallel reduction strategy agree with their original (unmutated) versions:

```
mutationIrrelevant m e =
  typeChecks e ==>
    (stepwise NoMutant e == stepwise m e)
 && (parallel NoMutant e == parallel m e)
```

If either reduction strategy's result with the mutation enabled differs from its result with the mutation disabled, then the bug implanted by the mutation has been discovered. The more bugs found in this manner from the suite of 20 available mutations and the less time it takes to find them, the better.

To determine the effectiveness of our technique, we compare its performance against that of LazySearch, the enumeration testing tool that our technique augments. These

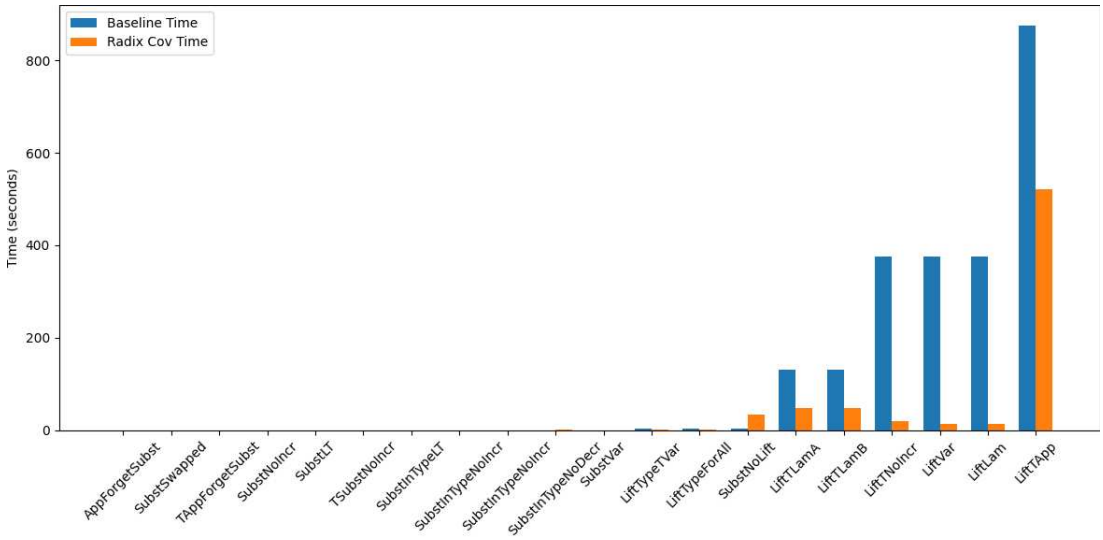

Figure 5. Time to bug discovery (in seconds) for LazySearch (in blue) and our extension (in orange).

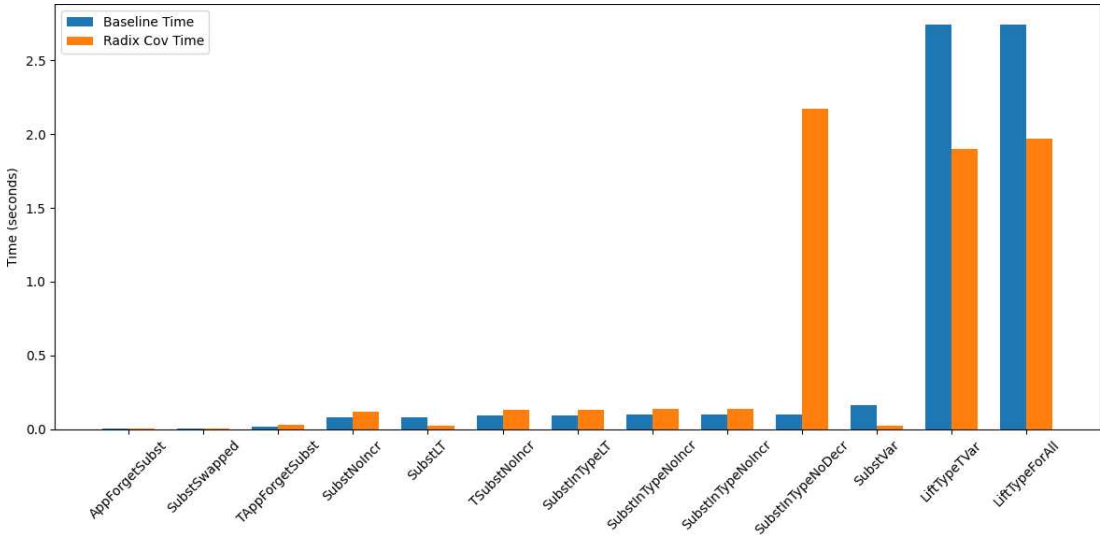

Figure 6. Zoomed-in version of Figure 5 for the easier-to-find bugs.

experiments were ran on a machine with 18 double-threaded Intel Core i9 cores running at 3.00GHz.

We started by generating baseline values by running LazySearch on the property described above for each of the 20 mutations and recorded the time until a counterexample is found. The results are shown in Figure 5 in blue. These baseline results stratify the mutations into two groups: quick-to-find bugs, which can be found in less than 3 seconds, and slow-to-find bugs, which take upwards of two minutes (between 130 and 876 seconds). There are 14 quick-to-find bugs in the baseline and 6 slow-to-find ones. As the quick-to-find bugs are indistinguishable in Figure 5, we provide a zoomed-in version with only these (Figure 6).

Against this baseline we measured the performance of our tool, functional Radix Coverage, on the same metric, measuring its time to finding a counterexample for each of the 20 mutations. The results are shown in Figure 5 in orange. Our technique shows dramatic improvements in bug finding speed for the 6 slow-to-find bugs when compared to the baseline. The slowest-to-find bug in the baseline has a 1.68 fold improvement of 876 seconds to 522 seconds, shaving nearly 6 minutes off the search time. The other five slow-to-find mutations are found strikingly faster than the baseline, reducing their search time to under 48 seconds. In order of increasing baseline time for those five, Radix Coverage provided a 2.72, 2.78, 18.41, 28.56, and 28.59 fold improvement in discovery time. As can be seen in Figure 6, the 13 out of 14

quick-to-find bugs displayed stay under 3 seconds with our technique, with most performing similarly to the baseline. The 14^{th} quick-to-find bug, SubstNoLift, which is displayed on the main graph and has a baseline discovery time of 4.49 seconds, is the one somewhat significant loss for our technique, with Radix Coverage finding it 7.33 times slower. But this only makes for a 32.91 second discovery, which is in line with the time required for most of the slow-to-find bugs with Radix Coverage's dramatic improvements.

Choosing Hyperparameters. One thing that is not clear from the discussion above is that both LazySearch and Radix Coverage rely on hyperparameters.

LazySearch takes as a parameter the maximum size of terms it will enumerate and search through while looking for a counterexample. Starting with maximum size 1 and incrementing while LazySearch misses at least one counterexample, we found that to discover counterexamples for all 20 mutations, LazySearch needs to enumerate well-typed terms up through size 13, so that is the size we selected to determine our baseline for comparison.

Radix Coverage on the other hand requires selecting several more hyper-parameters in addition to size: *time-budget per pattern*, *interaction strength*, and *fineness level*. To understand the effect of these parameters, we run several experiments varying each parameter in isolation.

We found that:

- As we increase the time-budget per pattern, we explore each pattern more thoroughly, making deeper bugs easier to uncover, but at the same time potentially increase the time-to-failure for shallower bugs.
- As we increase the interaction strength, we explore more interactions between patterns; in the limit, infinite interaction strength would recover exhaustive enumeration. As a result, this can significantly increase time-to-failure.
- The effect of fineness level is mixed and dependent on the other three parameters. At a low size, a higher fineness can speed up bug-finding as it leads to finer partitions; at the same time, however, combining high fineness with high interaction strength can decrease the efficiency of testing as the total number of patterns significantly increases.

Due to the trade-offs and complementary strengths and weaknesses of the various configurations of hyperparameters, in our experiment we decided to take a balanced approach. We ran three competing configurations for each mutation. One which has a low interaction strength (1), low fineness (1), and medium time-budget (50 seconds)—which can quickly yield results for shallow bugs: essentially this allocates an equal amount of time to each constructor. A second configuration has a medium interaction strength (5), high fineness (5), and high time-budget (75 seconds)—which provides a more thorough exploration of the deeper patterns in the input space. And a third configuration with high interaction strength (7), medium fineness (3), and low time-budget (10 seconds)—which ensures more interactions between constructors are exercised, but at a shallower depth than the second one. While the ACTS manual claims to only support interaction strengths up to 6, it can go higher when mixed-strength coverage is used (such as in RadixCov), which we leveraged to counteract the effective interaction strength reduction caused by RadixCov's parameter splitting.

6 Related Work

The potentially related literature on combinatorial and property-based testing is vast. In this section we only discuss the most closely related work, starting with recent work that also attempts to bridge combinatorial testing and functional programming, and then turning to other potentially related approaches in the two fields. For a more thorough exposition, we refer the reader to the combinatorial testing survey of Nie and Leung [17].

Combinatorial Testing for Functional Programming. The closest related work is the recent work of Goldstein et al. [6], which also aims to bring the power of combinatorial coverage to property-based testing. In their work, they introduce a similar notion of combinatorial coverage for algebraic datatypes. In fact, our notion of coverage can be completely encoded in their regular tree expressions. However, that expressivity comes at a price: there is no way to reconcile their notion of coverage with out-of-the-box combinatorial testing tools, which is why they developed a method of thinning random generators without the use of covering arrays. Moreover, their approach relies on the existence of such a random generator that must be provided by the user, a process that can take significant effort depending on the complexity of the property under test, to the point of constituting a significant research contribution in itself [8, 15, 18, 25]. As a result, we view the two approaches as largely orthogonal: the approach in this paper pushes the boundaries of completely black-box testing, using state-of-the-art automatic testing tools to efficiently find errors; Goldstein et al., on the other hand, squeeze every ounce of performance from expertly handwritten generators, but only if such generators exist.

Combinatorial Testing for Context-Free Grammars. While Goldstein et al. is the only approach we are aware of that brings combinatorial testing ideas to the functional setting, a number of other lines of work extend combinatorial testing to handle other domains, such as context-free grammars. In particular, Lämmel and Schulte define combinatorial coverage between nodes in a grammar by first bounding the space up to some depth, and then allowing users to manually specify what interactions need to be covered [12]. Similarly, Salecker and Glesner define combinatorial coverage by mapping depth-bounded grammar derivations to sets of

parameter choices [21]. In contrast, our approach can handle the infinite domains traditionally associated with algebraic data types, by partitioning the infinite space into a finite number of (potentially) infinite subspaces, and then defining combinatorial coverage by focusing on interactions between the constructors that define those subspaces.

Enumerative Property-Based Testing. As we discussed throughout the paper, our technique focuses on enumerative property-based testing by improving upon LazySearch [4]. There are other approaches in the same domain, such as SmallCheck [20] or the more efficient LeanCheck [1], which offer property-based testing infrastructure and support generation of inputs through exhaustive enumeration. We chose to build upon LazySearch instead, as it provides support for effectively pruning the input search space to filter out invalid inputs (a capability which the other tools do not provide), and we were able to seamlessly leverage the same mechanism to achieve our partitioning.

Random Property-Based Testing. Finally, as discussed in the introduction, property-based testing tools [3, 13] often leverage randomness to sample from the large space of potential inputs, an approach which comes with both advantages and disadvantages compared to exhaustively enumerating inputs up to some size. Thoroughly comparing enumerative and randomized approaches lies way beyond the scope of this paper and we refer the interested reader in the recent work of Shi et al. [23] who undertake such a case study. Instead, in this work, we focused on enumerative testing which provides a black-box automatic method of handling preconditions for properties written in Haskell and we provide a way of extracting more value out of a finite testing budget.

7 Conclusion and Future Work

In this paper, we showed how techniques from combinatorial testing can be used to improve the efficiency of a state-of-the-art enumeration-based property-based testing tool for Haskell, by reprioritizing the enumeration order to be more evenly distributed across the input space. In the future, we would like to further explore the effects of the various hyperparameters using a more thorough benchmark such as the recent one of Shi et al. [23] (which is not yet publicly available as of the time of writing) and apply our techniques to more domains.

Data Availability Statement

Code implementing Radix Coverage, its extension to algebraic data types, and the evaulation experiments are available in the Zenodo artifact [5].

Acknowledgments

We thank Harrison Goldstein, Benjamin C. Pierce, John Hughes, Michael Hicks, and the rest of the PLUM lab for their helpful comments. This work was supported by the NSF under award #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 NSF). This manuscript has been authored by Ivan Perez, an employee of KBR under Prime Contract No. 80ARC020D0010 with the NASA Ames Research Center.

References

[1] Rudy Matela Braquehais. 2017. *Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing.* Ph. D. Dissertation. University of York. http://etheses.whiterose.ac.uk/19178/

[2] 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. https://doi.org/10.1007/978-3-319-07151-0_2

[3] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM, 268–279. http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf

[4] Jonas Duregard. 2022. LazySearch: A Library for finding values satisfying a lazy predicate. https://hackage.haskell.org/package/lazy-search

[5] Segev Elazar Mittelman, Aviel Resnick, Ivan Perez, Alwyn E. Goodloe, and Leonidas Lampropoulos. 2023. *Artifact for Paper: Don't Go Down the Rabbit Hole: Reprioritizing Enumeration for Property-Based Testing.* https://doi.org/10.5281/zenodo.8162198

[6] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover: Combining Combinatorial and Property-Based Testing.

[7] Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. 2022. Random Testing of a Higher-Order Blockchain Language (Experience Report). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP).* https://lemonidas.github.io/pdf/BlockchainTesting

[8] Cătălin Hriţcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Dénès, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. 2016. Testing Noninterference, Quickly. *Journal of Functional Programming (JFP); Special issue for ICFP 2013* 26 (April 2016), e4 (62 pages). https://doi.org/10.1017/S0956796816000058 Technical Report available as arXiv:1409.0393.

[9] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. http://crest.cs.ucl.ac.uk/fileadmin/crest/sebasepaper/JiaH10.pdf

[10] Raghu Kacker. 2013. ACTS: A Combinatorial Test Generation Tool. Proceedings of Sixth IEEE International Conference on Software Testing, Verification and Validation ICST 2013, Luxembourg, -1.

[11] D Richard Kuhn, Raghu N Kacker, and Yu Lei. 2010. Practical combinatorial testing. *NIST special Publication* 800, 142 (2010), 142.

[12] Ralf Lämmel and Wolfram Schulte. 2006. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3964)*, M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko (Eds.). Springer, 19–38. https://doi.org/10.1007/11754008_2

[13] Leonidas Lampropoulos. 2018. *Random Testing for Language Design*. Ph. D. Dissertation. University of Pennsylvania.

[14] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating good generators for inductive relations. *PACMPL* 2, POPL (2018), 45:1–45:30. https://doi.org/10.1145/3158133

[15] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-Driven QuickChecking of Compilers. *Proc. ACM Program. Lang.* 1, ICFP, Article 15 (aug 2017), 23 pages. https://doi.org/10.1145/3110259

[16] Max S. New, Burke Fetscher, Robert Bruce Findler, and Jay McCarthy. 2017. Fair enumeration combinators. *Journal of Functional Programming* 27 (2017), e19. https://doi.org/10.1017/S0956796817000107

[17] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* 43, 2, Article 11 (Feb. 2011), 29 pages. https://doi.org/10.1145/1883612.1883618

[18] 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* (Waikiki, Honolulu, HI, USA) *(AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

[19] SIR Reporitory. 2015. TCAS Case Study. https://sir.csc.ncsu.edu/portal/bios/tcas.php

[20] 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. http://www.cs.york.ac.uk/fp/smallcheck/smallcheck.pdf

[21] Elke Salecker and Sabine Glesner. 2012. Combinatorial Interaction Testing for Test Selection in Grammar-Based Testing. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche (Eds.). IEEE Computer Society, 610–619. https://doi.org/10.1109/ICST.2012.148

[22] Kaushik Sarkar and Charles J. Colbourn. 2017. Upper Bounds on the Size of Covering Arrays. *SIAM J. Discrete Math.* 31, 2 (2017), 1277–1293. https://doi.org/10.1137/16M1067767

[23] 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)*. https://lemonidas.github.io/pdf/ETNA

[24] Jose Torres-Jimenez and Eduardo Rodriguez-Tello. 2012. New bounds for binary covering arrays using simulated annealing. *Inf. Sci.* 185, 1 (2012), 137–152. https://doi.org/10.1016/j.ins.2011.09.020

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 283–294. https://doi.org/10.1145/1993498.1993532

Received 2023-06-01; accepted 2023-07-04