Do Judge a Test by its Cover
Combining Combinatorial and Property-Based Testing

Harrison Goldstein[1], John Hughes[2],
Leonidas Lampropoulos[3], and Benjamin C. Pierce[1]

[1] University of Pennsylvania, USA
[2] Chalmers University of Technology and Quviq AB, Sweden
[3] University of Maryland, USA

Abstract. *Property-based testing* uses randomly generated inputs to validate high-level program specifications. It can be shockingly effective at finding bugs, but it often requires generating a very large set of inputs to do so. In this paper, we apply ideas from *combinatorial testing*, a powerful and widely studied testing methodology, to modify the distributions of our random generators and find bugs with fewer tests. The key concept is *combinatorial coverage*, which measures the degree to which a given set of tests exercises every possible choice of values for every small combination of inputs.

In its "classical" form, combinatorial coverage only applies to programs taking inputs of a very constrained shape: essentially, a Cartesian product of finite sets. We generalize combinatorial coverage to the richer world of algebraic data types by formalizing a class of *sparse test descriptions* based on regular tree expressions. This new definition of coverage inspires a novel *combinatorial thinning* algorithm for improving the coverage of random test generators, requiring many fewer tests to catch bugs. We evaluate the new method on two case studies, a typed evaluator for System F terms and a Haskell compiler, showing significant improvements in both.

Keywords: Combinatorial testing, Combinatorial coverage, QuickCheck, Property-based testing, Regular tree expressions, Algebraic data types

1 Introduction

Property-based testing, popularized by tools like *QuickCheck* [6], is a principled way of testing software that focuses on functional specifications rather than lists of input-output examples. A property is a formula like

$$\forall x.\ P(x,\ f(x)),$$

where f is the function under test, and P is some executable logical relationship between an input x and the output $f(x)$. The test harness generates random values for x, hoping to either uncover a counterexample—an x for which

$\neg P(x,\ f(x)))$, indicating a bug—or else provide confidence that f is correct with respect to P.

With a well designed random test case generator, property-based testing has a non-zero probability of generating *every* valid test case (up to a given size limit); property-based testing is thus guaranteed to find every possible bug that can be provoked by an input below the size limit... *eventually.* Unfortunately, since each input is generated *independently*, random testing may end up repeating the same or similar tests many times before happening across the specific input which provokes a bug. This poses a particular problem in settings like *continuous integration*, where feedback is needed quickly—ideally, there should be a way to guide the generator to a *more interesting and diverse* set of inputs, "thinning" the distribution to find bugs with fewer tests.

Combinatorial testing, a popular approach to testing from the systems literature [2, 13, 14], offers an attractive metric for judging which tests are most interesting. In its classical presentation, combinatorial testing advocates choosing tests to maximize *t-way coverage* of a program's input space—i.e., to exercise all possible choices of concrete values for every combination of t input parameters. For example, suppose a program p takes Boolean parameters `w`, `x`, `y`, and `z`, and suppose we want to test that p behaves well for for every choice of values for every pair of these four parameters. If we choose carefully, we can check all such choices—all 2-*way interactions*—with just five test cases:

```
1. w = False x = False y = False z = False
2. w = False x = True  y = True  z = True
3. w = True  x = False y = True  z = True
4. w = True  x = True  y = False z = True
5. w = True  x = True  y = True  z = False
```

You can check for yourself: for any two parameters, every combination of values for these parameters is covered by some test. For example,

"`w = False` and `x = False`"

is covered by #1, while both

"`w = True` and `x = True`" and "`w = True` and `y = True`"

are covered by #5. Thus, we get 100% pairwise coverage with just five out of the $2^4 = 16$ possible inputs.

Why is this interesting? Because bugs in real systems are often provoked by specific choices of just a few parameters. Indeed, a survey of a variety of real-world systems found that testing all 2-way parameter interactions could catch up to 93% of bugs; the same survey concluded that testing all 6-way interactions might be sufficient to catch essentially all bugs in practice [13].

If combinatorial coverage can be used to concentrate bug-finding power into small sets of tests, it is natural to wonder whether it could also be used to thin the distribution of a random generator. So far, however, combinatorial testing has mostly been applied in settings where the input to a program is just a vector of parameters, each drawn from a small finite set. Could we take it further? In particular, could we apply ideas from combinatorial testing to the setting addressed by *QuickCheck*—functional programs whose inputs are drawn from structured, potentially infinite data types like lists and trees?

Our first contribution is showing how to generalize the definition of combinatorial coverage to work with *regular tree expressions*, which themselves generalize the algebraic data types found in most functional languages. Instead of covering combinations of parameter choices, we measure coverage of *test descriptions*—concise representations of sets of tests, encoding potentially interesting interactions between data constructors. For example, the test description cons(true, ◇false) describes the set of Boolean lists that have true as their first element, followed by at least one false somewhere in the tail.

Our second contribution is a method for enhancing property-based testing using combinatorial coverage. We propose an algorithm that uses combinatorial coverage information to thin an existing random generator, leading it to more interesting test suites that find bugs more often. A concrete realization of this algorithm in a tool called *QuickCover* was able, in our experiments, to guide random generation to find bugs using an average of 10× fewer tests than *QuickCheck*. While *generating* test suites is considerably slower, this cost can be amortized over many runs of the test suite, for example in a continuous-integration setting.

In summary, we offer these major contributions:

– We generalize the notion of combinatorial coverage to work over a set of *test descriptions* and show how this new definition generalizes to algebraic data types with the help of regular tree expressions (Section 3). Section 4 describes the technical details behind the specific way we choose to represent these descriptions.
– We propose a process for guiding the test distribution of an existing random generator based on our generalized notion of combinatorial coverage (Section 5).
– Finally, we demonstrate, with two case studies, that *QuickCover* can find bugs using significantly fewer tests (Section 6) than pure random testing.

We conclude with an overview of related work (Section 7), and ideas for future work (Section 8). To set the stage, we begin with a brief review of "classical" combinatorial testing.

2 Classical Combinatorial Testing

Combinatorial testing measures the "combinatorial coverage" of test suites, aiming to find more bugs with fewer tests. Standard presentations of combinatorial testing [13] are phrased in terms of a number of separate input parameters; here, for notational consistency with the rest of the paper, we will instead assume that a program takes a single input consisting of a tuple of values.

Assume we are given some finite set $\mathcal{C}$ of *constructors*, and consider the set of n-tuples over $\mathcal{C}$:

$$\{\mathsf{tuple}_n(C_1,\ \ldots,\ C_n) \mid C_1,\ \ldots,\ C_n \in \mathcal{C}\}$$

(The "constructor" tuple_k is not strictly needed in this section, but it makes the generalization to constructor trees and tree regular expressions in Section 3 smoother.) We can use these tuples to represent test inputs to systems. For example a web application might be tested under a configuration

$$\mathsf{tuple}_4(\mathsf{Safari},\ \mathsf{MySQL},\ \mathsf{Admin},\ \mathsf{English})$$

in order to verify some end-to-end property of the system.

A *specification* of a set of tuples is written informally using notation like:

$$\mathsf{tuple}_4(\mathsf{Safari}{+}\mathsf{Chrome},\ \mathsf{Postgres}{+}\mathsf{MySQL},\ \mathsf{Admin}{+}\mathsf{User},\ \mathsf{French}{+}\mathsf{English})$$

This specification restricts the set of valid tests to those that have valid browsers in the first position, valid databases in the second, and so on. Specifications are thus a lot like types—they pick out a set of valid tests from some larger set. We make this notion of specification more formal and concrete in Section 3.

To define combinatorial coverage, we introduce the notion of *partial* tuples—tuples where some elements are left indeterminate (written $\top$). For example:

$$\mathsf{tuple}_4(\mathsf{Chrome},\ \top,\ \mathsf{Admin},\ \top).$$

A description is *compatible* with a specification if its concrete (non-$\top$) constructors are valid in the positions where they appear. Thus, the description above is compatible with our web-app configuration specification, while this one is not:

$$\mathsf{tuple}_4(\mathsf{MySQL},\ \mathsf{MySQL},\ \mathsf{French},\ \top)$$

We say a test *covers* a description—which, conversely, *describes* the test—when the tuple matches the description in every position that does not contain $\top$. For example, the description

$$\mathsf{tuple}_4(\mathsf{Chrome},\ \top,\ \mathsf{Admin},\ \top)$$

describes these tests:

$$\begin{array}{l}
\mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{MySQL},\ \mathsf{Admin},\ \mathsf{English})\\
\mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{MySQL},\ \mathsf{Admin},\ \mathsf{French})\\
\mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{Postgres},\ \mathsf{Admin},\ \mathsf{English})\\
\mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{Postgres},\ \mathsf{Admin},\ \mathsf{French})
\end{array}$$

Finally, we call a description *t-way* if it fixes exactly t constructors, leaving the rest as $\top$.

Now, suppose a system under test takes a tuple of configuration values as input. Given some correctness property (e.g., the system does not crash), a test for the system is simply a particular tuple, while a test *suite* is a set of tuples. We can then define *combinatorial coverage* as follows:

Definition 1. *The t-way* combinatorial coverage *of a test suite is the proportion of t-way descriptions, compatible with a given specification, that are covered by some test in the suite.*

We say that t is the *strength* of the coverage.

A test suite with 100% 2-way coverage for the present example can be quite small. For example,

$$\begin{gathered}\mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{Postgres},\ \mathsf{Admin},\ \mathsf{English})\\ \mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{MySQL},\ \mathsf{User},\ \mathsf{French})\\ \mathsf{tuple}_4(\mathsf{Safari},\ \mathsf{Postgres},\ \mathsf{User},\ \mathsf{French})\\ \mathsf{tuple}_4(\mathsf{Safari},\ \mathsf{MySQL},\ \mathsf{Admin},\ \mathsf{French})\\ \mathsf{tuple}_4(\mathsf{Safari},\ \mathsf{MySQL},\ \mathsf{User},\ \mathsf{English})\end{gathered}$$

achieves 100% coverage with just five tests. The fact that a single test covers many different descriptions is what makes combinatorial testing work: while the number of descriptions that must be covered is combinatorially large, a single test can cover combinatorially many descriptions. In general, for a tuple of size n, the number of descriptions is given by $\binom{n}{t}$ ways to choose t parameters multiplied by the number of distinct values each parameter can take on.

3 Generalizing Coverage

Of course, inputs to programs are often more complex than just tuples of enumerated values, especially in the world of functional programming. To apply the ideas of combinatorial coverage in this richer world, we generalize tuples to *constructor trees* and tuple specifications to *regular tree expressions*. We can then give a generalized definition of test descriptions that makes sense for algebraic data types, setting up for a more powerful definition of combinatorial coverage.

A *ranked alphabet* Σ is a finite set of atomic *data constructors*, each with a specified *arity*. For example, the ranked alphabet

$$\Sigma_{\mathsf{list(bool)}} \triangleq \{(\mathsf{cons},\ 2),\ (\mathsf{nil},\ 0),\ (\mathsf{true},\ 0),\ (\mathsf{false},\ 0)\}$$

defines the constructors needed to represent lists of Booleans. Given a ranked alphabet Σ, the set of *trees* over Σ is the least set $\mathcal{T}_\Sigma$ that satisfies the equation

$$\mathcal{T}_\Sigma = \{C(t_1,\ \dots,\ t_n) \mid (C,\ n) \in \Sigma \wedge t_1,\ \dots,\ t_n \in \mathcal{T}_\Sigma\}.$$

Regular tree expressions are a compact and powerful tool for specifying sets of trees [9, 8]. They are generated by the following syntax:

$$\begin{aligned} e \triangleq\ & \top\\ \mid\ & e_1 + e_2\\ \mid\ & \mu X.\ e\\ \mid\ & X\\ \mid\ & C(e_1,\ \dots,\ e_n) \text{ for } (C,\ n) \in \Sigma \end{aligned}$$

Each of the operations on regular tree expressions has an analog in standard regular expressions over strings: $+$ corresponds to disjunction of regular expressions, μ corresponds to iteration, and the parent-child relationship corresponds to concatenation. These expressions form a rich language for describing tree structures.

The *denotation* function $[\![\cdot]\!]$ mapping regular tree expressions to sets of trees is the least function satisfying these equations:

$$\begin{aligned}
[\![\top]\!] &= \mathcal{T}_\Sigma \\
[\![C(e_1,\ \ldots,\ e_n)]\!] &= \{C(t_1,\ \ldots,\ t_n) \mid t_i \in [\![e_i]\!]\} \\
[\![e_1 + e_2]\!] &= [\![e_1]\!] \cup [\![e_2]\!] \\
[\![\mu X.\ e]\!] &= [\![e[\mu X.\ e/X]]\!]
\end{aligned}$$

Regular tree expressions subsume standard first-order *algebraic data type* definitions. For example, the Haskell definition

```
data BoolList = Cons Bool BoolList | Nil
```

is equivalent to the regular tree expression

$$\mu X.\ \mathsf{cons}(\mathsf{true} + \mathsf{false},\ X) + \mathsf{nil}.$$

Crucially for our purposes, regular tree expressions can also be used to define sets of trees that cannot be described with plain ADTs. For example, the expression

$$\mathsf{cons}(\mathsf{true} + \mathsf{false},\ \mathsf{nil})$$

denotes all single-element Boolean lists, while

$$\mu X.\ \mathsf{cons}(\mathsf{true},\ X) + \mathsf{nil}$$

describes the set of lists that only contain true. They can even express non-local constraints like "true appears at some point in the list":

$$\mu X.\ \mathsf{cons}(\top,\ X) + \mathsf{cons}(\mathsf{true},\ \mu Y.\ \mathsf{cons}(\top,\ Y) + \mathsf{nil}))$$

This machinery smoothly generalizes the ideas from Section 3. Tuples are just a special form of trees, while specifications and test descriptions can be written as regular tree expressions. This gives us most of what we need to define algebraic data types. Recall the definition of t-way combinatorial coverage: "the proportion of (1) $\underline{t\text{-way descriptions}}$, (2) $\underline{\text{compatible with a given specification}}$, that (3) $\underline{\text{are covered by some test}}$ in the suite." What does this mean in the context of regular tree expressions and trees?

Condition (3) is easy: a test (i.e., a tree) t covers a test description (a regular tree expression) d if $t \in [\![d]\!]$.

For (2), consider some regular tree expression τ representing an algebraic data type that we would like to cover. We will use τ as our specification and say that a description d is compatible with τ if $[\![\tau]\!] \cap [\![d]\!] \neq \varnothing$. As with string regular expressions, this can be checked efficiently.

The only remaining question is (1): which set of t-way descriptions to use. We argue in the next section that the set of *all* regular tree expressions is too broad, and we offer a simple and natural alternative.

4 Sparse Test Descriptions

A naïve way to generalize the definition of t-way descriptions to regular tree expressions would be to first define the *size* of a regular tree expression as the number of operators (C, $+$, or μ) in it and then define a t-way description to be any regular tree expression of size t. However, this approach does not specialize nicely to the classical case; for example the description

$$\mathsf{tuple}_4(\mathsf{Safari} + \mathsf{Chrome},\ \top,\ \top,\ \top)$$

would be counted as "4-way" (3 constructors and 1 "+" operator), even though it is trivially covered by every possible test. Worse, descriptions of this form are not particularly compact. For example, the smallest possible description of lists where true is followed by false,

$$\mu X.\ \mathsf{cons}(\top,\ X) + \mathsf{cons}(\mathsf{true},\ \mu Y.\ \mathsf{cons}(\top,\ Y) + \mathsf{cons}(\mathsf{false},\ \mu Z.\ \mathsf{cons}(\top,\ Z) + \mathsf{nil}))$$

has size $t = 14$. Our representation should pack as much information as possible into small descriptions, making t-way coverage meaningful for small values of t and increasing the complexity of the interactions captured by our definition of coverage.

In sum, we want a definition of coverage that straightforwardly specializes to the tuples-of-constructors case and that captures interesting structure with smaller descriptions. Our proposed solution, described next, takes its inspiration from temporal logic. We first encode an "eventually" ($\diamond$) operator that allows us to write the expression from above much more compactly as $\diamond\mathsf{cons}(\mathsf{true},\ \diamond\mathsf{false})$. This can be read as "somewhere in the tree, there is a cons node with a true node to its left and a false node somewhere in the tree to its right." Then we define a restricted form of *sparse test descriptions* that use just $\diamond$, $\top$, and constructors.

4.1 Encoding "Eventually"

The "eventually" operator can actually be encoded using the regular tree expression operators we have already defined—i.e., we can add it without adding any formal power. First, define the set of *templates* for the ranked alphabet Σ:

$$\mathbb{T} \triangleq \{C(\top_1,\ \ldots,\ \top_{i-1},\ [\,],\ \top_{i+1}\ldots,\ \top_n) \mid (C,\ n) \in \Sigma,\ 1 \leq i \leq n\}$$

For each constructor C in Σ, the set of templates $\mathbb{T}$ contains $C([\,],\ \top,\ \ldots,\ \top)$, $C(\top,\ [\,],\ \top,\ \ldots,\ \top)$, etc., all the way to $C(\top,\ \ldots,\ \top,\ [\,])$, enumerating every way to place one hole in the constructor and fill every other argument slot with $\top$. (We ignore null-ary constructors in $\mathbb{T}$.) Then we define "next" ($\circ e$) and "eventually" ($\diamond e$) as

$$\circ e \triangleq \sum_{T \in \mathbb{T}} T[e]$$

$$\diamond e \triangleq \mu X.\ e + \circ X$$

where $T[e]$ is the replacement of $[\,]$ in T with e.[4] The definition of $\circ e$ says intuitively that it describes any tree $C(t_1,\ \ldots,\ t_n)$ in which e describes some direct child (i.e., t_1, t_2, and so on). Then, the definition of $\diamond e$ describes anything that described by e, plus (unrolling the μ) anything described by $\circ e$, $\circ\circ e$, and so on.

The example from the previous section,

$$\diamond\mathsf{cons}(\mathsf{true},\ \diamond\mathsf{false}),$$

should now make more sense. Our definition of $\diamond$ neatly captures the "somewhere in the tree" constraints that would otherwise necessitate a large, complex description.

4.2 Defining Coverage

Even with the eventually operator, there is still a fair amount of freedom in how we define the set of t-way descriptions. In this section we present one possibility that we call *sparse test descriptions*, which we found to be useful in practice; in future work (Section 8) we discuss another interesting option.

We define the set of sparse test descriptions for a given Σ to be the trees generated by

$$d \triangleq \top \mid \diamond C(d_1,\ \ldots,\ d_n) \text{ for } (C,\ n) \in \Sigma,$$

that is, trees consisting of constructors prefixed by $\diamond$ and $\top$. We call these descriptions "sparse" because they match specific ancestor-descendant arrangements of constructors but place no restriction on the constructors in between, due to the "eventually" before each constructor.

Sparse test descriptions are designed to be compact—in order to useful in practice and compatible with the classical definition of coverage, our descriptions should be as information-dense as possible. This is why sparse descriptions do not contain the $+$ operator: any test that covers either $C(d_1,\ \ldots,\ d_n)$ or $D(d_1,\ \ldots,\ d_m)$ will also necessarily cover $C(d_1,\ \ldots,\ d_n) + D(d_1,\ \ldots,\ d_m)$, so we do not lose any power by removing $+$. Furthermore, we do not include the μ operator directly, instead relying on $\diamond$. Intuitively, $\diamond$ captures a pattern of recursion that is general enough to express interesting non-local constraints while keeping description complexity low. This specific format for test descriptions is not the only possible choice with these properties, of course, but the case studies in Section 6 show that it works well in at least two challenging domains that are relevant to programming languages as a whole.

Finally, we can define the size of a description based on the number of constructors it contains. Intuitively, a t-way description is one with t constructors; however, in order to be consistent with the classical definition, we omit constructors whose types permit no alternatives. For example, all of the tuple constructors (e.g. tuple_4 from our running example) are left out of the size calculation.

[4] This construction is why we choose to deal with finite ranked alphabets: if Σ were infinite, $\mathbb{T}$ would be infinite, and $\circ e$ would be an infinite term that is not expressible as a standard regular tree expression.

This makes t-way sparse test description coverage correspond directly to classical t-way parameter interaction coverage.

Sparse test descriptions work nicely for signatures like Boolean lists:

$$\tau_{\mathsf{list(bool)}} \triangleq \mu X.\ \mathsf{cons}(\mathsf{true} + \mathsf{false},\ X) + \mathsf{nil}.$$

The set of all 2-way descriptions that are compatible with $\tau_{\mathsf{list(bool)}}$ is:

$$\begin{array}{ccc} \diamond\mathsf{cons}(\diamond\mathsf{true},\ \top) & \diamond\mathsf{cons}(\diamond\mathsf{false},\ \top) & \diamond\mathsf{cons}(\top,\ \diamond\mathsf{nil}) \\ \diamond\mathsf{cons}(\top,\ \diamond\mathsf{cons}(\top,\ \top)) & \diamond\mathsf{cons}(\top,\ \diamond\mathsf{true}) & \diamond\mathsf{cons}(\top,\ \diamond\mathsf{false}) \end{array}$$

Sparse descriptions also work as expected for types like

$$\mathsf{tuple}_4(\mathsf{Safari}{+}\mathsf{Chrome},\ \mathsf{Postgres}{+}\mathsf{MySQL},\ \mathsf{Admin}{+}\mathsf{User},\ \mathsf{French}{+}\mathsf{English}).$$

Despite some stray occurrences of $\diamond$, as in

$$\diamond\mathsf{tuple}_4(\diamond\mathsf{Chrome},\ \diamond\mathsf{MySQL},\ \top,\ \top),$$

the descriptions still describe the same sets of tests as the standard tuple descriptions without the uses of $\diamond$. Thus, our new definition of combinatorial coverage generalizes the classical one.

Sparse test descriptions capture a rich set of test constraints in a compact form. The real proof of this is in the numbers—see Section 6 for those—but a few more examples may help illustrate.

Arithmetic Expressions Consider the type of simple arithmetic expressions over the constants $\mathsf{0}$, $\mathsf{1}$, and $\mathsf{2}$:

$$\tau_{\mathsf{expr}} \triangleq \mu X.\ \mathsf{add}(X,\ X) + \mathsf{mul}(X,\ X) + \mathsf{0} + \mathsf{1} + \mathsf{2}.$$

This type has 2-way descriptions like

$$\diamond\mathsf{add}(\diamond\mathsf{mul}(\top,\ \top),\ \top) \text{ and } \diamond\mathsf{mul}(\top,\ \diamond\mathsf{add}(\top,\ \top)),$$

which capture different nestings of addition and multiplication.

System F For a more involved example, let's look at some 2-way sparse descriptions for a much more complex data structure: terms of the polymorphic lambda calculus, System F.

$$\begin{aligned} \tau &\triangleq \mathcal{U} \mid \tau_1 \to \tau_2 \mid n \mid \forall.\tau \\ e &\triangleq () \mid n \mid \lambda\tau.\ e \mid (e_1\ e_2) \mid \Lambda.\ e \mid (e\ \tau) \end{aligned}$$

(We use de Bruijn indices for variable binding, meaning that each variable occurrence in the syntax tree is represented by a natural number indicating which enclosing abstraction it was bound by.)

There is a fair amount of freedom in representing System F syntax as a regular tree expression. A reasonable place to start is

$$\mu X.\ \mathsf{unit} + \mathsf{var}(\textsc{Var}) + \mathsf{abs}(\textsc{Type},\ X) + \mathsf{app}(X,\ X) + \mathsf{tabs}(X) + \mathsf{tapp}(X,\ \textsc{Type}),$$

where Type is defined in a similar way and Var represents natural number de Bruijn indices.

This already admits useful 2-way descriptions like

$$\diamond\mathsf{app}(\diamond\mathsf{abs}(\top,\ \top),\ \top) \text{ and } \diamond\mathsf{app}(\diamond\mathsf{app}(\top,\ \top),\ \top),$$

which capture relationships between lambda abstractions and applications. In Section 6.1, we use descriptions like these to find bugs in an evaluator for System F expressions; they ensure that our test suite adequately covers different nestings of abstractions and applications that might provoke bugs.

With a little domain-specific knowledge, we can make the descriptions capture even more. When setting up our case study in Section 6.2, which searches for bugs in GHC's strictness analyzer, we found that it was often useful to track coverage of the *seq* function, which takes two functions as arguments, executes the first for any side-effects (e.g., exceptions), and then executes the second. Modifying our regular expression type to capture *seq* as a first-class constructor results in 2-way descriptions now include interactions like

$$\diamond\mathsf{seq}(\diamond\mathsf{app}(\top,\ \top),\ \top)$$

that encode interactions of *seq* with other more fundamental System F constructors. These interactions are crucial for finding bugs in a strictness analyzer, since *seq* gives fine-grained control over the evaluation order within a particular expression.

5 Thinning Generators with QuickCover

Having generalized the definition of combinatorial coverage to structured data types, the next step is to explore ways of using coverage to improve property-based testing.

When we first approached this problem, we planned to go "all-in" on the conventional combinatorial testing methodology of generating *covering arrays* [30], i.e., test suites with 100% t-way coverage for a given t. Rather than use an unbounded stream of random tests, we would test properties using only the tests in the covering array. However, we encountered two major problems with this approach. First, as t grows, covering arrays become frighteningly expensive to generate. While there are efficient methods for generating covering arrays in special cases like 2-way coverage [7], general algorithms for generating relatively compact covering arrays are complex and often slow [19]. Second, we found that covering arrays for sets of test descriptors in the format described above did not do particularly well at finding bugs! In a series of preliminary experiments with one of our case studies, we found that with 4-way coverage (the highest we could generate in a reasonable time-frame), our covering arrays did not reliably catch all of the bugs in our test system.

However, after some more experiments and head scratching, we discovered an alternate approach that works quite well. The trick is to embrace the random input generation that makes property-based testing so effective. In the remainder of this section, we present an algorithm that uses combinatorial coverage to "thin" a random generator, guiding it to more interesting inputs. Rather than generate a fixed set of tests in the style of covering arrays, this approach produces an unbounded stream of interesting test inputs. Then we discuss some concrete details behind *QuickCover*, a Haskell implementation of our algorithm which we use to obtain the experimental results in Section 6.

5.1 Online Generator Thinning

The core of our algorithm is the standard *QuickCheck* generate-and-test loop. Given a test generator `gen` and a property `p`, *QuickCheck* generates inputs repeatedly until either (1) the property fails, or (2) a time limit is reached. (The limit is chosen based on the user's specific *testing budget*, and can vary significantly in practice. In our experiments, we know *a priori* that a bug exists in the program, so we forego the limit entirely and just run tests until the property fails.)

```
QuickCheck(gen, p):
  repeat LIMIT times:
    # Generate 1 new input
    x = gen()
    # Check the property
    if !p(x), return False
  return True
```

Our algorithm modifies this basic one to use combinatorial coverage information when choosing the next test to run.

```
QuickCover(strength, fanout, gen, p):
  coverage = initCoverage()
  repeat LIMIT times:
    # Generate fanout potential inputs
    xs = listOf(gen(), fanout)
    # Find the input with the best improved coverage
    x = argmax[x in xs](
      coverageImprovement(x, coverage, strength) )
    # Check the property
    if !p(x), return False
    # Update the coverage information
    coverage = updateCoverage(x, coverage, strength)
  return True
```

The key idea is that, instead of generating a single input at each iteration, we generate several (controlled by the parameter `fan-out`) and select the one that increases combinatorial coverage the most. We test the property on that input, and (provided it succeeds) update the coverage information based on the test we ran.

This algorithm is generic with respect to the representation for coverage information, but the particular choice of data structure and interpretation makes a significant difference in both efficiency and effectiveness. In our implementation, coverage information is represented by a multi-set of descriptions:

```
initCoverage():
  return emptyMultiset()

coverageImprovement(x, coverage, strength):
  ds = descriptions(x, strength)
  return sum([ 1 / (count(d, coverage) + 1)
               for d in ds ])

updateCoverage(x, coverage, strength):
  return union(descriptions(x, strength), coverage)
```

At the beginning, the multi-set is empty; as testing progresses, each test is evaluated based on `coverageImprovement`. If a description d had previously been covered n times, it contributes $\frac{1}{n+1}$ to the score. For example, if a test input covers d_1 and d_2, where previously d_1 was not covered and d_2 was covered 3 times, the total score for the test input would be $1 + 0.25 = 1.25$.

At first glance, one might have expected a simpler approach based on sets instead of multi-sets. Indeed, this was the first thing we tried, but it turned out to perform substantially worse than the multiset-based one in our experiments. The reason is that just covering each description once turns out not to be sufficient to find all bugs, and, once most descriptions have been covered, this approach essentially degenerates to normal random testing. By contrast, the multi-set representation continues to be useful over time; after each description has been covered once, the algorithm begins to favor inputs that cover descriptions a second time, then a third time, and so on. This allows *QuickCover* to generate arbitrarily large test suites that continue to benefit from combinatorial coverage.

Keeping track of coverage information like this does create some overhead. For each test that *QuickCover* considers (including those that are never *run*), work needs to be done to analyze which descriptions the test covers and check those against the current multi-set. This overhead means that *QuickCover* is often slower than *QuickCheck* in terms of wall-clock time, even when it finds a bug by running significantly fewer tests. In the next section, we argue that the overhead of *QuickCover* can be amortized in the context of *continuous integration*, and the case studies in Section 6 suggest that this could yield substantial benefits. Moreover, in Section 6.2 we show an instance where *QuickCover* is actually faster than *QuickCheck* in terms of wall-clock time to find bugs because the tests themselves take a long time to run. Appendix A describes finer details around our Haskell implementation of *QuickCover*.

6 Evaluation

Since *QuickCover* adds some overhead to generating tests, we hypothesized that it would be particularly well suited to situations where each test will be run multiple times.

Of course, running the same test repeatedly *on the same code* is pointless: if it were ever going to fail, it would do so on the first run (ignoring the thorny problem of *flaky tests* [20]). However, running the same test on *successive versions* of the code is not only useful; it is standard practice in two common settings: *regression testing* (checking that code continues to work after a change) and especially *continuous integration* (where regression tests are run every time a developer checks in a new version of the code). In these settings, the overhead introduced by generating more tests than we actually run can be amortized, since the same tests may be reused very many times. The cost of *generating* the test suite is much less important than the cost of running it.

In order to validate this theory, we designed two experiments using *QuickCover*. The primary goal of these experiments was to answer the question: Does *QuickCover* actually reduce the number of tests needed to find bugs in a real system? If this were the case, *QuickCover* would be useful for continuous integration settings where small, effective test suites are key. As a secondary goal, we wanted to understand if the generator thinning overhead was *always* too high to make *QuickCover* useful for real-time property-based testing, or if there were any cases where using *QuickCover* would yield a wall-clock improvement even if tests are only run once.

Both case studies answer our primary question in the affirmative: the first case study, in particular, shows that *QuickCover* needs an average 10× fewer tests to find bugs, compared to pure random testing. We choose an evaluator for System F terms as our example because it allows us to test how *QuickCover* behaves in a small but realistic scenario that requires a fairly complex random testing setup. Our second case study replicates results from Pałka et al. [25], scaling up and applying *QuickCover* to find bugs in a production compiler, the Glasgow Haskell Compiler (GHC) [22]. This setting gives us a useful look at our second question, and shows that, in situations where tests themselves are slow to run, *QuickCover* even reduces the wall-clock time needed to find bugs in real time.

6.1 Case Study: Normalization Bugs in System F

Our first case study examines the effects of guiding a highly tuned and optimized test generator for System F [11, 27] terms, using combinatorial coverage. The generator component produces well-typed System F terms by construction (no mean feat on its own) and is tuned to produce a highly varied distribution of different terms. Despite all the care put into the base generator, we found that modifying the test distribution using *QuickCover* results in a test suite that finds bugs with substantially fewer inputs.

Generating "interesting" programs (for finding compiler bugs, for example) is an active research area. For instance, a generator for well-typed simply typed lambda-terms has been used to reveal bugs in GHC [25, 5, 16], while generating C programs without "undefined behaviors" has been used to find many bugs in production compilers [33, 26]. These cases are examples of *differential testing*: different compilers (or different versions of the same compiler) are run against each other on the same inputs to reveal discrepancies. Similarly, for this case study we tested different evaluation strategies for System F, comparing the behavior of various buggy versions to a reference implementation.

Recall the definition of System F from Section 4.2. Let $y[x/n]$ stand for substituting x for variable n in y, and $x \uparrow_n$ for lifting: incrementing the indices of all variables above n in x. Then, for example, the standard rule for substituting a type τ for variable n inside a type abstraction $\Lambda.\ e$ requires lifting τ and incrementing the de Bruijn index of the variable being substituted by one:

$$(\Lambda.\ e)[\tau/n] = \Lambda.\ e[\tau \uparrow_0 /n+1]$$

Here are two ways to get this wrong: forget to lift the variables, or forget to increment the index. Those bugs would lead to the following erroneous definitions (the missing operation is shown in red):

$$(\Lambda.\ e)[\tau/n] = \Lambda.\ e[\tau \uparrow_0 /n+1] \quad \text{and} \quad (\Lambda.\ e)[\tau/n] = \Lambda.\ e[\tau \uparrow_0 /n+1].$$

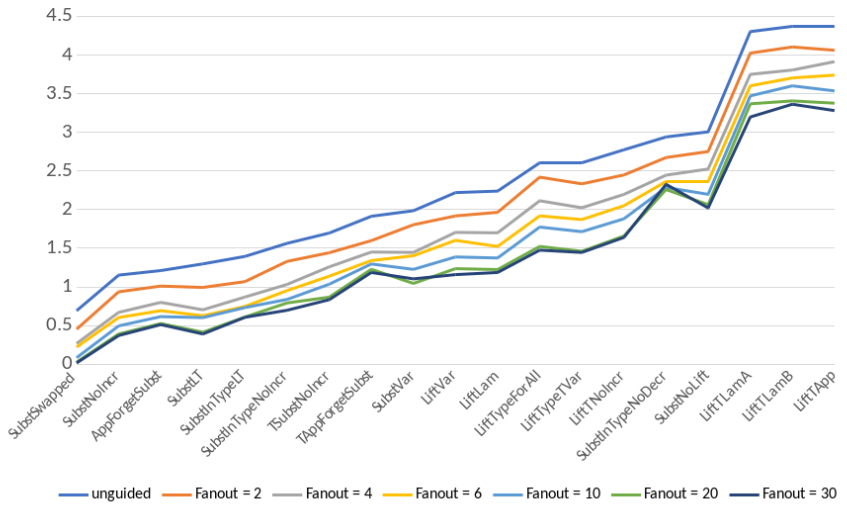
Using mistakes like these (specifically in the substitution and variable lifting functions) as inspiration, we created 19 *mutated* versions of two different evaluation relations. The mutations are explained in detail in Appendix B.

The two evaluation relations we implemented simplify terms in slightly different ways; the first implements standard big-step evaluation, and the second uses a parallel evaluation relation to fully normalize terms. (We chose check both evaluation orders, since some mutations, only actually cause a bug in one implementation or the other.) Since we were interested in bugs in either evaluation order, we tested the property:

```
eval e == eval_mutated e && peval e == peval_mutated e
```

With a highly-tuned generator as our baseline, we used both *QuickCheck* and *QuickCover* to generate a stream of test values for `e` and measured the average number of tests required to find a bug (i.e., Mean-Tests-To-Failure, or MTTF) for each approach.

Surprisingly, we found little to no difference in MTTF between 2-way, 3-way, and 4-way testing, but changing the fan-out did make a large impact. Figure 1 shows both absolute MTTF for various choices of fan-out ($\log_{10}$ scale) and the performance improvement as a ratio of un-thinned MTTF to thinned MTTF. All choices of fan-out produced better MTTF results than the baseline, but higher



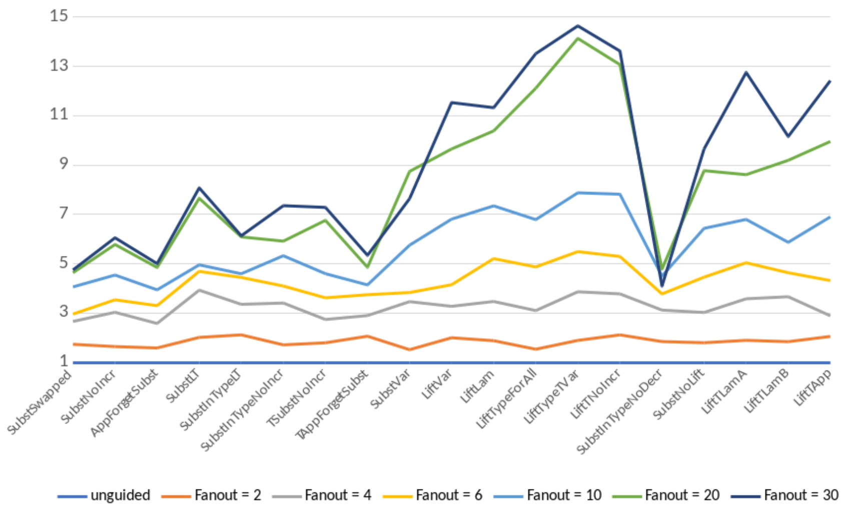


Fig. 1. Top: System F MTTF, $\log_{10}$ scale, plotted in order of MTTF for un-thinned random tests, $t = 2$.
Bottom: System F MTTF ratio of MTTF for un-thinned random tests to MTTF for *QuickCover*, $t = 2$.

values of fan-out tended to be more effective on average. In our best experiment, a fan-out of 30 found a bug in an average of 15× fewer tests than the baseline; on average it was about 10× better. Figure 2 shows the total MTTF improvement across 19 bugs, compared to the maximum theoretical improvement. If our algorithm were able to perfectly pick the best test input every time, the improvement would be proportional to the fan-out (i.e. it is impossible for our algorithm be more than 10× better with a fan-out of 10). On the other hand, if combinatorial coverage were irrelevant to test failure, then we would expect the

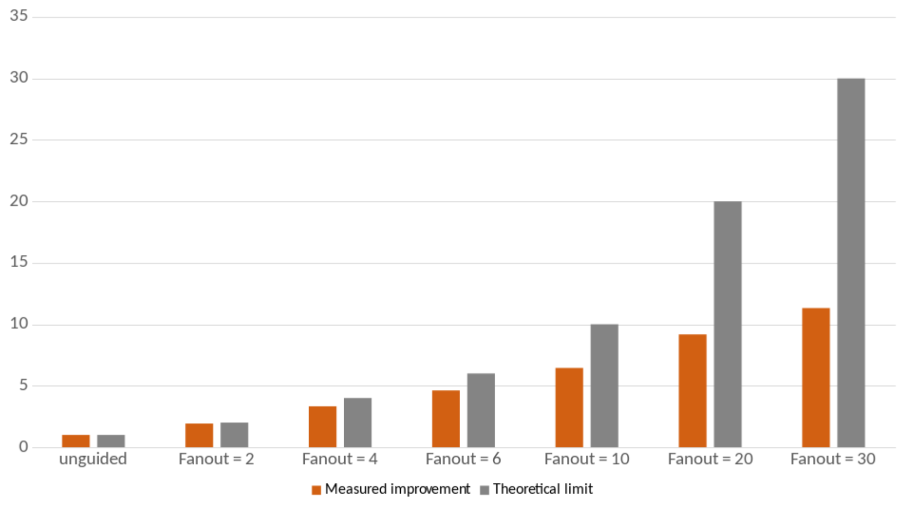


Fig. 2. System F, proportional reduction in total number of tests needed to find all bugs.

QuickCover test suites to have the *same* MTTF as *QuickCheck*. It is clear from the figure that *QuickCover* is really quite effective: for small fan-outs, it is very close to the theoretical optimum, and with a fan-out of 30 it achieves about $\frac{1}{3}$ of the potential improvement—three *QuickCover* test cases are more likely to provoke a bug than thirty *QuickCheck* ones.

6.2 Case Study: Strictness Analysis Bugs in GHC

To evaluate how our approach scales, and to answer whether *QuickCover* can be used not only to reduce the number of tests required but also to speed up bug-finding, we replicated the case study of Pałka et al. [25], which found bugs in the strictness analyzer of GHC 6.12 using a hand-crafted generator for well-typed lambda terms; we replicated their experimental setup, but used *QuickCover* to thin their generator and produce better tests. Two attributes of this case study make it an excellent test of the capabilities of our combinatorial thinning approach. First, it is a case study that found bugs in a real compiler by generating random well typed lambda terms, and therefore we can evaluate whether the reduction in number of tests observed in the System F case study scales to a production setting. Second, running a test involves invoking the GHC compiler, a heavyweight external process. As a result, reducing the number of tests required to provoke a failure should (and does) lead to an observable improvement in terms of wall-clock performance.

Concretely, Pałka et al. [25] generate a list of functions that manipulate lists of integers (with type `[Int] -> [Int]`) and compare the behavior of these functions on partial lists (lists with undefined elements or tails) when compiled with and without optimizations, another example of differential testing. They uncover errors in the strictness analyzer component of GHC's optimizer that introduce semantic inconsistencies where the non-optimized version correctly fails with an error while the optimized version prints something to the screen before failing:

Input	$-O0$ Output	$-O2$ Output
`[undefined]`	`Exception`	`[Exception]`
`[1,undefined]`	`Exception`	`[1,Exception]`
`[1,2,undefined]`	`Exception`	`[1,2,Exception]`

Finally, to balance the costly compiler invocation with the similarly costly smart generation process, Pałka et al. [25] group 1000 generated functions together in a single module to be compiled; this number was chosen to strike a precise 50-50 balance between generation time and compilation/execution time for each generated module. Since our thinning approach itself introduced approximately a 25% overhead in generation time, we increased the number of tests per module to 1250 to maintain the same balance.

We ran our experiments in a Virtual Box running Ubuntu 12.04 (a version old enough to allow for executing GHC 6.12.1), with 4GB RAM in a host machine running i7-8700 @ 3.2GHz. We performed 100 runs of the original case study and 100 runs of our variant that adds combinatorial thinning, using a fan-out of 2 and a strength of 2. We found that our approach reduces the mean number of *tests* required from 21268 ± 1349 to 14895 ± 1056 for a (42% reduction) and also reduces the mean *time* to failure, from 193 ± 13 seconds to 149 ± 12, a 30% improvement.

7 Related Work

The combinatorial testing literature is vast; a detailed survey can be found in [24]. Here we discuss just the most closely related work, in particular, other attempts to generalize combinatorial testing to structured and infinite domains. We also discuss other approaches to property based testing with similar goals to to ours, such as adaptive random testing and (branch) coverage-guided fuzzing.

7.1 Generalizations of Combinatorial Testing

There have already been some attempts to apply combinatorial testing to more complex and even infinite types.

Salecker and Glesner [29] extend combinatorial testing to sets of terms generated by a context-free grammar. Their approach cleverly maps *derivations* up to some length k to sets of parameter choices and then uses standard full-coverage test suite generation algorithms to pick a subset of derivations to test. The main limitation of this approach is the depth parameter k. By limiting the derivation length, this approach only defines coverage over a finite subset of the input type. By contrast, our definition of coverage is based on description size rather than term size and provides more flexibility for “packing” multiple descriptions into a single test.

Another approach to combinatorial testing of context-free inputs is due to Lämmel and Schulte [15]. Their system also uses a depth bound, but it provides the user finer-grained control. At each node in the grammar, the user is free to limit the coverage requirements and prune unnecessary tests. This is an elegant solution for situations where the desired interactions are know *a priori*. Unfortunately, this approach does not work well for the range of types we are interested in, and it requires a fair amount of manual input.

Finally, Kuhn et al. [12] present a notion of *sequence covering arrays* to describe combinatorial coverage of sequences of events. We believe that t-way sequence covering arrays in their system are equivalent to $(2t-1)$-way full-coverage test suites of the appropriate list type in ours. They also have a reasonably efficient algorithm for generating covering arrays in this specialized case.

Our idea to use regular tree expressions for coverage is partly inspired by Usaola et al. [32] and Mariani et al. [21]. These works focus on covering regular expressions themselves, rather than using regular expressions to cover ADTs, but they do explore the idea of combinatorial coverage in the world of formal languages.

In Appendix A we discuss some details around our implementation of *QuickCover* in Haskell. The idea of applying regular tree expressions to Haskell terms has also been addressed by Serrano and Hage [31], who developed a library called `t-regex` for matching Haskell terms with regular tree expressions.

7.2 Similar Approaches in Property-Based Testing

There are also several existing systems that attempt to improve on standard property-based testing by changing the way inputs are selected.

The *SmallCheck* [28] library generates test suites that contain all "small" test cases, which means that they are likely to have high combinatorial coverage for some small t. In contrast, *QuickCover* provides higher-strength combinatorial coverage without worrying about the size of each individual test. There are pros and cons for each approach: *QuickCover* guides generation to combinatorially interesting tests, but it might miss some small test cases; *SmallCheck* is extremely thorough with its coverage of small tests, but it requires a large volume of tests to be effective.

Coverage guided fuzzing tools like AFL [18] use a similar approach to ours, but they use code coverage (rather than combinatorial coverage) as a feedback mechanism for finding more interesting tests. There has already been successful work bringing these methods to functional programming [17, 10], but it is still far from perfect. One major downside is that measuring code coverage requires a *grey-box* approach and must be done on-line. Combinatorial coverage, on the other hand, can be computed without knowledge of the code itself, and therefore provides a *black-box* alternative (which is valuable when the same test suite is to be used for many versions of the code).

Chen et al.'s *adaptive random testing* (ART) [3] uses an algorithm that is related to the one we use in *QuickCover*, in that they generate a set of random tests and select the most interesting to run. Rather than using combinatorial coverage, ART requires a *distance metric* on test cases—at each step, the candidate which is farthest from the already-run tests is selected. Chen et al. show that this approach finds bugs after fewer tests on average, in the programs they study. ART was first proposed for programs with numerical inputs, but Ciupa et al. [4] showed how to define a suitable metric on objects in an object-oriented language and used it to obtain a reduction of up to two orders of magnitude in the number of tests needed to find a bug. Like combinatorial testing, ART is a black-box approach that depends only on the test cases themselves, not on the code under test. However, Arcuri and Briand [1] question its value in practice, because of the (quadratic) number of distance computations it requires, from each new test to every previously executed test; in a large empirical study, they found that the cost of these computations made ART uncompetitive with ordinary random testing. While our approach also has significant computational overhead, the time and space complexities grow with the number of possible descriptions (derived from the data type definition and the choice of strength) and *not* with the total number of tests run; this means that testing will not slow down significantly over time. In addition, our approach works in situations where a distance metric between inputs does not make sense.

8 Conclusion and Future Work

In sum, this paper presents a generalized definition of combinatorial coverage and an effective way to use that definition for property-based testing. We expand the definition of combinatorial coverage to work in the realm of algebraic data types with the help of regular tree expressions. Our sparse test descriptions provide a robust way to look at combinatorial testing, which specializes to the classical approach. We use these sparse descriptions as a basis for *QuickCover*—a tool that thins a random generator with a bias towards increased combinatorial coverage. In two case studies, we show that *QuickCover* is useful in practice: it finds bugs using an average of 10× fewer tests.

Moving forward, we see a number of potential directions for further research.

8.1 Variations

Sparse test descriptions are *a great way* to define combinatorial coverage for algebraic data types, but not as *the only way*. Here we discuss some variations on our approach and why they might be interesting to explore.

Representative Samples of Large Types Perhaps it is possible to do combinatorial testing with ADTs by having humans decide exactly which trees to cover. This approach is already widely used in combinatorial testing to deal with types like integers, which (though their machine representations are technically finite) are much too large for testing to efficiently cover all "constructors." For example, if the tester knows (by reading the code, or because they wrote it) that the code has an if-statement examining `x < 5`, they might choose to cover

$$\texttt{x} \in \{-2147483648,\ 0,\ 4,\ 5,\ 6,\ 2147483647\}.$$

The tester covers values around 5 because those are important to the specific use case and boundary values and 0 to check for common edge-cases. Concretely, this practice means that instead of trying to cover $\mathsf{tuple}_3(\textsc{Int},\ \mathsf{true}{+}\mathsf{false},\ \mathsf{true}{+}\mathsf{false})$, the tester covers the specification

$$\mathsf{tuple}_3(-2147483648 + 0 + 4 + 5 + 6 + 2147483647,\ \mathsf{true} + \mathsf{false},\ \mathsf{true} + \mathsf{false}).$$

In our setting, this practice might mean choosing a representative set of *constructor trees* to cover, and then treating them like a finite set. In much the same way as with integers, rather than cover

$$\mathsf{tuple}_3(\tau_{\mathsf{list(bool)}},\ \mathsf{true} + \mathsf{false},\ \mathsf{true} + \mathsf{false}),$$

we could treat a selection of lists as atomic constructors, and cover the specification

$$\mathsf{tuple}_3(\boxed{[]} + \boxed{[\mathsf{true},\ \mathsf{false}]} + \boxed{[\mathsf{false},\ \mathsf{false},\ \mathsf{false}]},\ \mathsf{true} + \mathsf{false},\ \mathsf{true} + \mathsf{false})$$

which has 2-way descriptions like

$$\mathsf{tuple}_3(\boxed{[]},\ \top,\ \mathsf{false}) \quad \text{and} \quad \mathsf{tuple}_3(\boxed{[\mathsf{true},\ \mathsf{false}]},\ \mathsf{true},\ \top).$$

Just as testers choose representative sets of integers, testers could choose sets of trees that they think are interesting and only cover those trees. Of course, the set of all trees for a type is usually much larger and more complex than the set of integers, so this approach is likely not practical. Still, it is possible that small amounts of human intervention could be beneficial to the process of determining the right descriptions to cover.

Type-Tagged Constructors Another variation to our approach changes the way that ADTs are translated into constructor trees. In Appendix A we show a simple example of a `Translation` for lists of Booleans, but an interesting problem arises if we consider *lists of* lists of Booleans. The most basic approach would be to use the same constructors (`LCNil` and `LCCons`) for both "levels" of list. For example, `[[True]]` would become (with a small abuse of notation)

```
LCCons (LCCons LCTrue LCNil) LCNil.
```

Depending on the application, it might actually make more sense to have different constructors for the different list types (`[Bool]` vs. `[[Bool]]`). `[[True]]` could instead be translated as

```
LCOuterCons (LCInnerCons LCTrue LCInnerNil) LCInnerNil
```

(with a slight abuse of notation), allowing for a broader range of potential test descriptions. Both of these are valid translations, in the sense that both define a set of constructor trees that mimic the structure of lists of lists of Booleans.

This observation can be generalized to any polymorphic ADT—any time a single constructor is used at multiple types, it is likely beneficial to differentiate between the two. With this in mind, we might explore type-tagged constructors and trees, where any use of a constructor would be tagged with the monomorphized type that it belongs to.

Eventual Descriptions A third potential variation is a modification to make test descriptions a bit less sparse. Recall that sparse test descriptions are defined as

$$d \triangleq \top \mid \diamond C(d_1,\ \ldots,\ d_n).$$

What if we chose this instead?

$$\begin{aligned} d &\triangleq \diamond d' \\ d' &\triangleq C(d'_1,\ \ldots,\ d'_n) \end{aligned}$$

In the former case, every relationship is "eventual": there is never a requirement that a particular constructor appear *directly* beneath another. In the latter case, the descriptions enforce a direct parent-child relationship, and we simply allow the expression to match anywhere in the term. We might call this class "eventual" test descriptions.

We chose sparse descriptions because putting eventually before every constructor leaves more opportunities for different descriptions to be "interleaved" within a term. This leads to smaller test suites, in general. We ran some small experiments and found that this alternative proposal seemed to perform similarly across the board but worse in a few cases. Still, we think that experimenting with the use of eventually in descriptions may lead to interesting new opportunities.

8.2 Combinatorial Coverage of More Types

Our sparse tree description definition of combinatorial coverage is focused on inductive algebraic types. While these encompass a wide range of the types that functional programmers use, it is far from everything. The most promising next step is an extension of descriptions that generalizes to co-inductive types. We actually think that the current definition might almost suffice—regular tree expressions can denote infinite structures, so this generalization would likely only affect our algorithms and the implementation of *QuickCover*. We also should be able to include GADTs without too much hassle. Our biggest open question is function types: these seem to require something more powerful than regular tree expressions to describe, and it is not clear that combinatorial testing even makes sense for inputs that are functions.

8.3 Regular Tree Expressions for Directed Generation

As we have shown, regular tree expressions are a powerful language for picking out subsets of types. In this paper, we mostly focused on automatically generating small descriptions, but it might be possible to apply this idea more broadly for specifying sets of tests. One straightforward extension would be to use the same machinery that we use for *QuickCover*, but, instead of covering an automatically generated set of descriptions, the generator might ensure that some manual set of expressions is covered. For example, we could use a modified version of our algorithm to generate a test set where

$$\mathsf{nil},\ \mathsf{cons}(\top,\ \mathsf{nil}),\ \text{and}\ \mu X.\ \mathsf{cons}(\mathsf{true},\ X) + \mathsf{nil}$$

are all covered. (Concretely, that would be a test suite containing, at a minimum, the empty list, a singleton list, and a list containing only true.) This might be useful for cases where the testers know *a priori* that certain cases are important to test, but they still want to focus on random testing primarily.

A different approach would be to create a tool that synthesizes *QuickCheck* generators that only generate terms matching a particular regular tree expression. This idea, related to work on adapting *branching processes* to control test distributions [23], would make it easy to write highly customized generators that have meticulous control over the resulting test suites.

Acknowledgments

Thank you to Calvin Beck, Filip Niksic, and Irene Yoon for help with the early development of these ideas. Thanks to Kostis Sagonas, Alexandra Silva, and Andrew Hirsch for feedback along the way. We're grateful to Vetenskapsrådet in Sweden for funding under the SyTeC project (grant number 2016-06204).

This work was supported by NSF awards #1421243, *Random Testing for Language Design* and #1521523, *Expeditions in Computing: The Science of Deep Specification*, by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-18-C-0011, and by the United States Air Force and DARPA under Contract No FA8750-16-C-0022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Vetenskapsrådet, the NSF, the United States Air Force, or DARPA.

Bibliography

[1] Andrea Arcuri and Lionel C. Briand. 2011. Adaptive random testing: an illusion of effectiveness?. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 265–275. `https://doi.org/10.1145/2001420.2001452`

[2] Kera Z Bell and Mladen A Vouk. 2005. On effectiveness of pairwise methodology for testing network-centric software. In *2005 International Conference on Information and Communication Technology*. IEEE, 221–235.

[3] Tsong Yueh Chen, Hing Leung, and I. K. Mak. 2004. Adaptive Random Testing. In *Advances in Computer Science - ASIAN 2004, Higher-Level Decision Making, 9th Asian Computing Science Conference, Dedicated to Jean-Louis Lassez on the Occasion of His 5th Cycle Birthday, Chiang Mai, Thailand, December 8-10, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3321)*, Michael J. Maher (Ed.). Springer, 320–329. `https://doi.org/10.1007/978-3-540-30502-6\_23`

[4] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. 2008. ARTOO: adaptive random testing for object-oriented software. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 71–80. `https://doi.org/10.1145/1368088.1368099`

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

[6] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. `https://doi.org/10.1145/351240.351266`

[7] Charles J. Colbourn, Myra B. Cohen, and Renée Turban. 2004. A deterministic density algorithm for pairwise interaction coverage. In *IASTED International Conference on Software Engineering, part of the 22nd Multi-Conference on Applied Informatics, Innsbruck, Austria, February 17-19, 2004*, M. H. Hamza (Ed.). IASTED/ACTA Press, 345–352.

[8] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007.

[9] Bruno Courcelle. 1983. Fundamental Properties of Infinite Trees. *Theor. Comput. Sci.* 25 (1983), 95–169. `https://doi.org/10.1016/0304-3975(83)90059-2`

[10] Andy Gill and Colin Runciman. 2007. Haskell program coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, Gabriele Keller (Ed.). ACM, 1–12. `https://doi.org/10.1145/1291201.1291203`

[11] Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation. Éditeur inconnu.

[12] D. Richard Kuhn, James M. Higdon, James Lawrence, Raghu Kacker, and Yu Lei. 2012. Combinatorial Methods for Event Sequence 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, 601–609. `https://doi.org/10.1109/ICST.2012.147`

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

[14] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. 2004. Software fault interactions and implications for software testing. *IEEE transactions on software engineering* 30, 6 (2004), 418–421.

[15] 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`

[16] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 114–129. `http://dl.acm.org/citation.cfm?id=3009868`

[17] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *PACMPL* 3, OOPSLA (2019), 181:1–181:29. `https://doi.org/10.1145/3360607`

[18] lcamtuf. 2018. AFL quick start guide. `http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt`.

[19] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07).* IEEE, 549–556.

[20] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 643–653.

[21] Leonardo Mariani, Mauro Pezzè, and David Willmor. 2004. Generation of Integration Tests for Self-Testing Components. In *Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops*

The FormEMC, EPEW, ITM, Toledo, Spain, October 1-2, 2004 (Lecture Notes in Computer Science, Vol. 3236), Manuel Núñez, Zakaria Maamar, Fernando L. Pelayo, Key Pousttchi, and Fernando Rubio (Eds.). Springer, 337–350. `https://doi.org/10.1007/978-3-540-30233-9\_25`

[22] Simon Marlow and Simon Peyton-Jones. 2012. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications*, Amy Brown and Greg Wilson (Eds.). Vol. II. Available online under the Creative Commons Attribution 3.0 Unported license. `http://www.aosabook.org/en/ghc.html`

[23] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for QuickCheck generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 1–13. `https://doi.org/10.1145/3242744.3242747`

[24] 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`

[25] 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`

[26] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012.* 335–346. `https://doi.org/10.1145/2254064.2254104`

[27] John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974 (Lecture Notes in Computer Science, Vol. 19)*, Bernard Robinet (Ed.). Springer, 408–423. `https://doi.org/10.1007/3-540-06859-7\_148`

[28] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, Andy Gill (Ed.). ACM, 37–48. `https://doi.org/10.1145/1411286.1411292`

[29] 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`

[30] 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`

[31] Alejandro Serrano and Jurriaan Hage. 2016. Generic Matching of Tree Regular Expressions over Haskell Data Types. In *Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9585)*, Marco Gavanelli and John H. Reppy (Eds.). Springer, 83–98. https://doi.org/10.1007/978-3-319-28228-2_6

[32] Macario Polo Usaola, Francisco Ruiz Romero, Rosana Rodriguez-Bobada Aranda, and Ignacio García Rodríguez de Guzmán. 2017. Test Case Generation with Regular Expressions and Combinatorial Techniques. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 189–198. https://doi.org/10.1109/ICSTW.2017.38

[33] 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

Appendices

A QuickCover in Detail

Our evaluation is based on a concrete implementation of the previously mentioned algorithm called *QuickCover*. The tool is written in Haskell and relies heavily on machinery already provided by *QuickCheck*. The easiest way to use *QuickCover* is via the function

```
quickCover ::
  Translation a t -> Strength -> (a -> Bool) -> IO ().
```

For comparison, the vanilla `quickCheck` function is roughly of type

```
quickCheck :: Gen a -> (a -> Bool) -> IO ().
```

Given a *translation* (defined below) and a strength, this can be used in the same way as the standard `quickCheck` function. The difference is that instead of running the property with every generated input, *QuickCover* implements the algorithm from Section 5.1, generating a few tests at a time and only running the one which improves t-way combinatorial coverage the best.

The translation has three fields; each field can be tuned, allowing the user to adjust the way that coverage is computed.

```
data Translation a t = Translation
  { haskTy :: TypeRepr t
  , toTermRepr :: a -> TermRepr t
  , gen :: Gen a
  }
```

The `haskTy` and `toTermRepr` fields define an explicit structure that mirrors the data type declarations of the input type. The `haskTy` field can be thought of as one of the specifications that are shown throughout the paper, although using a syntax closer to Haskell's type declarations for convenience, and the `toTermRepr` field is a translation from the actual Haskell terms to simple trees of constructors. The `gen` field is a *QuickCheck* generator for Haskell terms of the input type. Often it is possible to let *QuickCheck* automatically derive generators for a type, but users can supply their own generators if they want. This feature is important because many testing applications rely on custom generators that respect certain invariants.

Figure 3 shows an example of a translation for a property of type `[Bool] -> Bool`. In the example `haskTy` defines the two types we care about (lists and Booleans), the `toTermRepr` function recursively turns a real list of Booleans into a constructor tree of type `TermRepr LC`, and the generator comes from *QuickCheck*'s `Arbitrary` type class.

The current *QuickCover* prototype does not generate these translations automatically; we expect it is not too hard to do so using one of Haskell's many reflection mechanisms. However, there is often some freedom in exactly how to define a translation. For example, it can sometimes be useful to treat some subtrees as opaque for the purposes of coverage. In our System F case study, (see

```
-- | The type of Boolean list constructors
data LC = LCNil | LCCons | LCTrue | LCFalse
  deriving (Eq, Ord)

-- | A mapping from concrete lists to a generic repr.
listToTermRepr :: [Bool] -> TermRepr LC
listToTermRepr [] = CNode LCNil []
listToTermRepr (True  : xs) =
  CNode LCCons [ CNode LCTrue []
               , listToTermRepr xs
               ]
listToTermRepr (False : xs) =
  CNode LCCons [ CNode LCFalse []
               , listToTermRepr xs
               ]

-- | All info. needed to test Boolean lists
listBoolTranslation :: Translation [Bool] LC
listBoolTranslation = Translation
  { haskTy = Map.fromList
      [ ("List", [(LCCons, ["Bool", "List"]), (LCNil, [])])
      , ("Bool", [(LCTrue, []), (LCFalse, [])])
      ]
  , toTermRepr = listToTermRepr
  , gen = arbitrary
  }
```

Fig. 3. An example of a translation for a property of type `[Bool] -> Bool`. At least for me right now, there is too little blank space under this caption.

Section 6.1) we choose to make the de Bruijn indices used to represent variables opaque: instead of writing the variable constructor as `(LCVar, ["Int"])`, we simply wrote `(LCVar, [])` and defined our `toTermRepr` function to map variables accordingly. When computing coverage, our algorithm treats all variables (`Var 0`, `Var 1`, etc) as the same constructor in the tree, cutting down on the number of descriptions that need to be covered, potentially in exchange for some testing effectiveness.

B Case Study 1: Glossary of Bugs

Bug	Description
SubstSwapped	Reverses the substitution direction when evaluating an application.
SubstNoIncr	Fails to increment variables during substitution.
AppForgetSubst	Fails to substitute terms when evaluating an application.
SubstLT	Flips a $>$ check to $<$ during substitution.
SubstInTypeLT	Flips a $>$ check to $<$ during type-level (forall) substitution.
SubstInTypeNoIncr	Fails to increment variables during type-level (forall) substitution.
TSubstNoIncr	Fails to increment variables during type substitution.
TAppForgetSubst	Fails to substitute types when evaluating an application.
SubstVar	Incorrectly substitutes variables (erroneous decrement).
LiftVar	Incorrectly lifts variables during substitution.
LiftLam	Fails to lift lambdas during substitution.
LiftTypeForAll	Incorrectly lifts variables during type-level (forall) substitution.
LiftTypeTVar	Fails to lift variables during type substitution.
LiftTNoIncr	Fails to lift lambdas during type substitution.
SubstInTypeNoDecr	Fails to decrement an index during type-level substitution.
SubstNoLift	Fails to lift variables during substitution.
LiftTLamA	Variation A on incorrect type variable lifting.
LiftTLamB	Variation B on incorrect type variable lifting.
LiftTApp	Incorrectly lift during type application.

C Case Study 1: `Translation` for System F Terms

Note that the constructors for both `Var` and `TVar` are opaque—we do not model the variable indices.

```
data SFC = ...

typeT :: Translation Type SFC
typeT =
  Translation
    { haskTy =
        Map.fromList
          [ ( "Type",
              [ (SFBase, []),
                (SFTBool, []),
                (SFFunc, ["Type", "Type"]),
                (SFTVar, []),
                (SFForAll, ["Type"])
              ]
            )
          ],
      toTermRepr = ...,
      gen = ...
    }

exprT :: Translation Expr SFC
exprT =
  Translation
    { haskTy =
        Map.fromList
          [ ( "Expr",
              [ (SFCon, []),
                (SFVar, []),
                (SFLam, ["Type", "Expr"]),
                (SFApp, ["Expr", "Expr"]),
                (SFCond, ["Expr", "Expr", "Expr"]),
                (SFBTrue, []),
                (SFBFalse, []),
                (SFTLam, ["Expr"]),
                (SFTApp, ["Expr", "Type"])
              ]
            )
          ]
          `Map.union` haskTy typeT,
      toTermRepr = ...,
      gen = ...
    }
```