Algebraic Combinatorial Testing

HARRISON GOLDSTEIN, University of Pennsylvania, USA
JOHN HUGHES, Chalmers University of Technology
LEONIDAS LAMPROPOULOS*, University of Maryland, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

Combinatorial testing is a powerful and widely studied testing methodology. Its 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. But, in its usual form, this metric only applies to programs whose inputs have 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. We apply our new definition in two case studies and show that our implementation, which augments random testing using combinatorial coverage, catches certain classes of bugs with significantly fewer tests.

Additional Key Words and Phrases: Combinatorial testing, Combinatorial coverage, QuickCheck, Property-based testing, Regular tree expressions, Algebraic data types

ACM Reference Format:
Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2020. Algebraic Combinatorial Testing. 1, 1 (July 2020), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Combinatorial testing is a popular approach to software testing that advocates running tests to maximize *t-way coverage* of a program's input space—i.e., that exercise all possible choices of values for every combination of t distinct input parameters. For example, suppose a program p has four Boolean parameters `w`, `x`, `y`, and `z`, and suppose we want to test that p behaves well for for every possible 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
```

*Also with University of Pennsylvania, Computing and Information Science.

Authors' addresses: Harrison Goldstein, hgo@seas.upenn.edu, University of Pennsylvania, Computing and Information Science, 3330 Walnut St, Philadelphia, PA, 19104, USA; John Hughes, , Chalmers University of Technology; Leonidas Lampropoulos, leonidaslamp@hotmail.com, University of Maryland, USA; Benjamin C. Pierce, bcpierce@cis.upenn.edu, University of Pennsylvania, Computing and Information Science, 3330 Walnut St, Philadelphia, PA, 19104, USA.

XXXX-XXXX/2020/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

You can check for yourself: for any two parameters, every combination of values for those 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. In a scenario where each test is expensive to run, even this difference is a huge win, but things get even better as the number of parameters increases: with 40 Boolean parameters (an input space of just over a trillion possible inputs), 2-way coverage can be achieved with only 11 tests.

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 possible 2-way parameter interactions can catch up to 93% of bugs, and the same survey concluded that testing all 6-way interactions might be sufficient to catch essentially all bugs in practice [Kuhn et al. 2010].

To date, however, combinatorial testing has mostly been applied in a rather limited domain—specifically, programs taking a finite number of parameters, where each parameter is drawn from a small finite set. Could we take it further? In particular, could we use ideas from combinatorial testing for testing functional programs whose the inputs are drawn from structured, potentially infinite data types like lists and trees?

Our main contribution is showing how the key idea of combinatorial testing, *combinatorial coverage*, can be generalized 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 aim to cover sets of *test descriptions*—concise representations of sets of tests, encoding potentially interesting interactions between data constructors. For example, the test description

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

describes the set of Boolean lists that have true as their first element, followed by at least one false somewhere in the tail.

We have applied our generalized notion of combinatorial coverage in the setting of *property-based testing*. Popularized in functional programming by tools like Haskell's *QuickCheck* [Claessen and Hughes 2000], property-based testing works (essentially) with properties of the form

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

The test harness checks $P(x,\ f(x))$ for random values of x, with the hopes of either uncovering a counter-example—an x for which $\neg P(x,\ f(x))$), indicating a bug—or else providing confidence that f is correct with respect to P.

We have implemented a tool called *QuickACT* ("QuickCheck for Algebraic Combinatorial Testing") that uses our generalized notion of combinatorial coverage to "thin" an existing generator for random data structures. In our case studies, *QuickACT*'s high-coverage test suites uncovered most of the bugs that purely random test suites did, in some cases with 9× fewer tests.

In summary, we offer these main contributions (following a brief review of "classical" combinatorial testing in Section 2):

- We generalize the notion of combinatorial coverage to work over a set of *test descriptions* and show how this new definition can generalize to algebraic data types, with the help of regular tree expressions (Section 3).
- We identify a subset of regular tree expressions that we call *sparse test descriptions*, which provide an intuitive way to specify complex properties of test inputs (Section 4).
- We propose a simple algorithm for "thinning" the test distribution of a random generator, and demonstrate, with two case studies, that thinning test sets with coverage of sparse

test descriptions as a metric leads to test suites that are useful in practice (Section 5). Our implementation succeeds in finding almost all of the bugs that standard *QuickCheck* does, and in the best cases it can do so with significantly fewer tests.

We conclude with a discussion of some potential variations on our approach (Section 6), an overview of related work (Section 7), and ideas for future work (Section 8).

2 CLASSICAL COMBINATORIAL TESTING

Combinatorial testing measures the combinatorial coverage of test suites, with the goal either of finding more interesting tests to run or of generating a small test suite with high bug-finding potential. Standard presentations of combinatorial testing are phrased in terms of "interactions" among a number of separate input parameters; here, for notational consistency with what follows, we will assume that a program takes a single input consisting of a tuple of values.

Assume some finite set C of *constructors*, and consider the set of n-tuples over C:

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

(The tuple_k constructors are not strictly needed in this section, but they make 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 the notation

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

which restricts the set of valid tests to only those that have valid browsers in the first position, valid databases in the second, and so on. Specifications are a lot like types—they pick out a subset 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 also need a way to talk about *partial* tuples—tuples where some elements are left indeterminate. We call these partial tuples *descriptions* and define them as as tuples in which some positions may be filled by $\top$, for example

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

We say a description is *compatible* with a specification if the constructors that the description fixes are valid in positions where they appear. The above description is compatible with our web-app configuration specification, while the following 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:

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

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

Now, suppose we have some system that takes such a tuple as input. Assuming that we have some property P that we want to check, a test for the system is simply a particular tuple, and a test *suite* is a set of tuples. We can now formally define the notion of *combinatorial coverage*:

Definition 2.1. 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.

As we saw in the introduction, a test suite with 100% 2-way coverage for our web app configurations can be quite small:

$$\begin{aligned}&\mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{Postgres},\ \mathsf{Admin},\ \mathsf{English})\\&\mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{MySQL},\ \mathsf{NotAdmin},\ \mathsf{French})\\&\mathsf{tuple}_4(\mathsf{Safari},\ \mathsf{Postgres},\ \mathsf{NotAdmin},\ \mathsf{French})\\&\mathsf{tuple}_4(\mathsf{Safari},\ \mathsf{MySQL},\ \mathsf{Admin},\ \mathsf{French})\\&\mathsf{tuple}_4(\mathsf{Safari},\ \mathsf{MySQL},\ \mathsf{NotAdmin},\ \mathsf{English})\end{aligned}$$

Here we achieve 100% coverage with just five tests. We can unpack that a bit by looking at the two-way descriptions that are covered by one particular test:

$$\mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{Postgres},\ \mathsf{Admin},\ \mathsf{English}) \xrightarrow{\text{covers}} \begin{aligned}&\mathsf{tuple}_4(\mathsf{Chrome},\ \mathsf{Postgres},\ \top,\ \top)\\&\mathsf{tuple}_4(\top,\ \mathsf{Postgres},\ \mathsf{Admin},\ \top)\\&\mathsf{tuple}_4(\top,\ \top,\ \mathsf{Admin},\ \mathsf{English})\\&\mathsf{tuple}_4(\mathsf{Chrome},\ \top,\ \mathsf{Admin},\ \top)\\&\mathsf{tuple}_4(\top,\ \mathsf{Postgres},\ \top,\ \mathsf{English})\\&\mathsf{tuple}_4(\mathsf{Chrome},\ \top,\ \top,\ \mathsf{English})\end{aligned}$$

The fact that a single test covers six different descriptions is the bread and butter of what makes combinatorial testing work: while the number of descriptions that must be covered is combinatorially large (for an n-tuple, there are $\binom{n}{t}$ ways to choose t parameters; this must then be multiplied by the number of distinct values each parameter can take on), a single test can cover combinatorially many descriptions. This combinatorial advantage makes combinatorial testing especially attractive in scenarios where running each test is expensive.

3 GENERALIZING COVERAGE

The inputs to a functional program are usually more interesting than a tuple of enumerated values. In this section, we generalize tuples to *constructor trees* and tuple specifications to *regular tree expressions*. We then generalize the definition of test descriptions, the penultimate step towards defining combinatorial coverage over algebraic data types.

A finite *ranked alphabet* Σ is a 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 used in 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

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

Regular tree expressions, generated by the syntax

$$\begin{array}{rcl} e & \triangleq & \top \\ & | & e_1 + e_2 \\ & | & \mu X.\, e \\ & | & X \\ & | & C(e_1,\, \ldots,\, e_n) \text{ for } (C,\, n) \in \Sigma \end{array}$$

are a compact and powerful tool for specifying sets of trees [Comon et al. 2007; Courcelle 1983]. The denotation function $[\![\cdot]\!]$ is the least function from regular tree expressions to sets of trees that satisfies these equations:

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

Each of the operations on regular tree expressions has an analog in standard regular expressions: $+$ corresponds to disjunction of regular expressions, μ corresponds to iteration, and the parent-child relationship corresponds to concatenation.

Regular tree expressions subsume standard first-order *algebraic data type* definitions, so we will use them here as types, to lighten the notation. For example, the Haskell definition

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

is equivalent to the regular tree expression

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

while

```
data Expr = Add Expr Expr
          | Mul Expr Expr
          | One | Two | Three
```

can be represented by

$$\mu X.\, \mathsf{add}(X,\, X) + \mathsf{mul}(X,\, X) + 1 + 2 + 3.$$

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 talk about non-local structure—structure like "true appears at some point in the list"—as evidenced by the expression

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

We can use this machinery to generalize the ideas from Section 3: tuples are just a special form of trees, while specifications and test descriptions can be captured with regular tree expressions. With this perspective, we can start to form a definition of combinatorial coverage for algebraic data types. Recall the definition of t-way combinatorial coverage: "the proportion of (1) t-way descriptions, (2) compatible with a given specification, that (3) are covered by some test in the suite." What does

this mean in the context of regular tree expressions and trees? (3) is easy: a test (i.e., a tree) t covers a test description (i.e., 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 standard regular expressions, this can be efficiently checked. 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 present 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 define the *size* of a regular tree expression as the number of operators (C, $+$, μ) in it and then define a t-way description to be any regular tree expression of size t. Unfortunately, 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 appear to be "4-way", even though it is covered by every single test. Also, many interesting and useful descriptions are very large using this definition—the description of any list where true is followed by false can be written most simply as

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

which has size $t = 14$! (The outermost μ matches the front of the list before the first true, the second matches the part between true and false, and the innermost matches the tail of the list.) Generating descriptions of this size would be totally intractable—there are over 2.5 million such trees that can be made using just + and nil—even with the combinatorial advantage of high-coverage tests, the resulting test suite would be far too large to generate or use.

We would like a definition of coverage that straightforwardly specializes to the tuples-of-constructors case and that captures more interesting structure in smaller descriptions. Our proposed solution takes inspiration from temporal logic: we 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."

4.1 Encoding Eventually

The eventually operator can actually be encoded as syntactic sugar for regular tree expressions without adding any formal power. First, we 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

$$C([],\ \top,\ \ldots,\ \top) \text{ and } C(\top,\ [],\ \top,\ \ldots,\ \top) \text{ all the way to } C(\top,\ \ldots,\ \top,\ []),$$

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 can 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.[1] This definition of "next" says that $\circ e$ 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 e describes and by unrolling the μ we can see that it also captures $\circ e$, $\circ\circ e$, and so on. This is exactly the semantics we want, capturing the "somewhere in the tree" intuition from the example above.

The eventually operator can be used to describe complex and interesting sets of trees—in particular sets of trees that would make interesting tests. For example, consider a system that reacts to a stream of events. Tests for this system can be represented as lists of events, for example

$$[\mathsf{read},\ \mathsf{write},\ \mathsf{close}] \text{ or, more explicitly, } \mathsf{cons}(\mathsf{read},\ \mathsf{cons}(\mathsf{write},\ \mathsf{cons}(\mathsf{close},\ \mathsf{nil}))).$$

It might be important to check how the program behaves when a read happens immediately before a write. That condition can be expressed with the description

$$\diamond\mathsf{cons}(\mathsf{read},\ \mathsf{cons}(\mathsf{write},\ \top)).$$

Likewise, if the test suite covers the description

$$\mathsf{cons}(\mathsf{write},\ \diamond\mathsf{close}),$$

we can be sure that it tests the case of a write followed eventually by a close.

What if, instead, we have a system that operates on terms of the untyped lambda calculus over a finite set of variables? All of the terms will match the specification

$$\mu X.\ \mathsf{abs}(\textsc{Var},\ X) + \mathsf{app}(X,\ X) + \textsc{Var}$$

where Var represents some finite set of variable symbols x, y, z, etc. In order to make sure that the test suite covers different nestings of app and abs, we can use the the descriptions

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

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. We discuss several possibilities in Section 6; here we present one simple alternative that we call *sparse test descriptions*.

The set of sparse test descriptions is defined as

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

We call these descriptions "sparse" because they match certain specific constructors without any restriction on the constructors in-between (due to the "eventually" before each constructor).

Note that μ and + are both omitted. We've replaced μ with $\diamond$, but what about disjunctions? It turns out that they do not add anything: any test that covers

$$C(d_1, \ldots, d_n) \text{ or } D(d_1, \ldots, d_m)$$

would also necessarily cover

$$C(d_1, \ldots, d_n) + D(d_1, \ldots, d_m),$$

[1] 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.

so adding disjunction would not make our descriptions more interesting.

Next we need a notion of size. To a first approximation, it might make sense define a t-way description as one with t data constructors, but on closer examination this is not exactly what we want. The problem is with tuples that we saw in Section 3; these are purely "structural" and do not represent any kind of choice in the test. Thus, we should not count tuple constructors towards the size of the term. With this adjustment, t-way sparse test description coverage will correspond directly to t-way parameter interaction coverage.

Sparse test descriptions work nicely for signatures like

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

the type of Boolean lists that we saw above. The set of all 2-way descriptions that are compatible with $\tau_{\mathsf{list(bool)}}$ (that is, the set of all two-constructor sparse descriptions that have a non-empty intersection with $\tau_{\mathsf{list(bool)}}$) is:

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

Crucially, sparse descriptions work as expected for types like

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

where the definition of coverage specializes to the classical one from Section 2. Even though we have extra uses of eventually, for example in

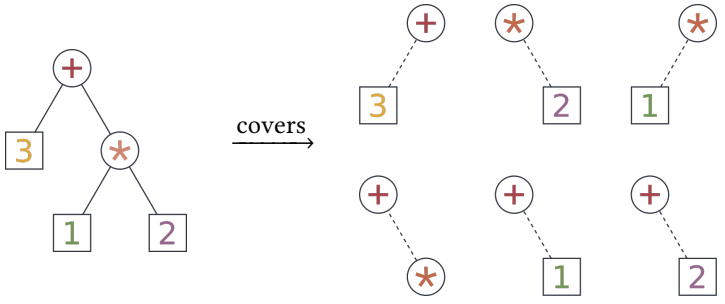
$$\diamond\mathsf{tuple}_4(\diamond\mathsf{Chrome},\ \diamond\mathsf{MySQL},\ \top,\ \top),$$

the descriptions still describe the same sets of tests.

How might we use these sparse test descriptions in practice? Consider a system that takes arithmetic expressions as input. We can write a specification like

$$\tau_{\mathsf{expr}} \triangleq \mu X.\ \mathsf{add}(X,\ X) + \mathsf{mul}(X,\ X) + 1 + 2 + 3,$$

to represent such expressions. (Obviously a real expression type would include integers beyond just 1, 2, and 3, but trying to cover the full range of integers would cause problems, as there are far too many integers to expect to cover all combinations of them in a test suite. In this case, as is typically done in practice, we simply choose a subset of integers to cover. We discuss this issue further in Section 6.) The diagram below shows the set of descriptions covered by one particular tree:



Note that this looks a lot like the set of descriptions covered by our configuration parameters in Section 3! A single test again covers six different descriptions. Written out formally, the test is

$$\mathsf{add}(3,\ \mathsf{mul}(1,\ 2)),$$

and it covers these descriptions:

$$\begin{array}{ccc} \diamond\mathsf{add}(\diamond 3,\ \top) & \diamond\mathsf{mul}(\top,\ \diamond 2) & \diamond\mathsf{mul}(\diamond 1,\ \top) \\ \diamond\mathsf{add}(\top,\ \diamond\mathsf{mul}(\top,\ \top)) & \diamond\mathsf{add}(\top,\ \diamond 1) & \diamond\mathsf{add}(\top,\ \diamond 2) \end{array}$$

5 CASE STUDIES

In this section, we describe *QuickACT*, a tool that applies our new definition of combinatorial coverage in the context of *QuickCheck* and property-based testing. We begin with some high level information about *QuickACT* and how it is implemented. Next, we describe the simple, offline algorithm that our tool uses to generate test suites. Finally, we discuss two case studies that use *QuickACT* in practice.

5.1 QuickACT

We have implemented an extension to *QuickCheck* called *QuickACT* that uses sparse test descriptions as a combinatorial coverage metric. *QuickACT* adapts *QuickCheck* generators to produce test suites with 100% combinatorial coverage. For users of the tool, the most important function, `quickACT`, has the following type:

```
quickACT :: Translation a -> Strength -> (a -> Bool) -> IO ()
```

For comparison, the vanilla `quickCheck` function has type `(a -> Bool) -> IO ()`. (Technically, `quickCheck` has a slightly more complicated type that includes the `Testable` type-class and allows properties of types other than `a -> Bool`, but for our purposes this presentation is more clear.)

Given a *translation* (defined below) and a strength (the integer t that specifies t-way combinatorial coverage), this can be used in the same way as the standard `quickCheck` function. Rather than run a set number of random tests, `quickACT` generates and runs a test suite with 100% t-way combinatorial coverage.

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

```
data Translation a = Translation
  { haskTy :: TypeExpr
  , toTerm :: a -> ConsTree
  , gen :: Gen a
  }
```

The `haskTy` and `toTerm` 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 `toTerm` 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. However, this flexibility does cause some issues, as we will see in the next section.

Figure 1 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 `toTerm` function recursively turns a real list of Booleans into a constructor tree of type `ConsTree`, and the generator comes from *QuickCheck*'s `Arbitrary` type class.

```
listToTerm :: [Bool] -> ConsTree
listToTerm [] = CNode "Nil" []
listToTerm (True  : xs) = CNode "Cons" [ CNode "True" []
                                       , listToTerm xs
                                       ]
listToTerm (False : xs) = CNode "Cons" [ CNode "False" []
                                       , listToTerm xs
                                       ]

Translation
{ haskTy = Map.fromList
    [ ("List", [("Cons", ["Bool", "List"]), ("Nil", [])])
    , ("Bool", [("True", []), ("False", [])])
    ]
, toTerm = listToTerm
, gen = arbitrary
}
```

Fig. 1. An example of a translation for a property of type `[Bool] -> Bool`.

The current *QuickACT* 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 sub-trees as opaque for the purposes of coverage. In our System F case study, (see Section 5.3) we choose to make the DeBruijn indices used to represent variables opaque: instead of writing the variable constructor as

```
("Var", ["Int"]),
```

we simply wrote

```
("Var", [])
```

and defined our `toTerm` 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.

5.2 An Offline Algorithm

A standard *QuickCheck* generator embodies a probability distribution over program inputs. This distribution is often highly tuned and customized to produce tests that are more likely to find bugs. Given one of these generators (which is provided via the translation and denoted `GenerateTerm` in the algorithm below), *QuickACT* modifies the distribution, "thinning" it to remove tests which do not improve combinatorial coverage and enforcing that the final test suite cover all possible descriptions. This process can be done completely offline, meaning that the test suites can be generated ahead of time and used multiple times as the program evolves. The following pseudo-code describes the high-level algorithm that *QuickACT* uses:

```
GenerateCover(τ, strength):
  tests ← ∅
  uncovered ← AllSparseDescriptions(τ, strength)
  while (uncovered != ∅):
    test ← GenerateTerm(τ)
    uncovered' ← uncovered - {d ∈ uncovered | test ∈ ⟦d⟧}
    if (uncovered != uncovered'):
      uncovered ← uncovered'
      tests ← tests ∪ {test}
  return tests
```

This algorithm relies on an externally defined function, `AllSparseDescriptions`, which takes a type τ, and a strength t, and generates t-way sparse descriptions that are compatible with τ. We could do this by generating all possible t-way descriptions and then filtering to only those descriptions d where

$$\llbracket \tau \rrbracket \cap \llbracket d \rrbracket \neq \emptyset,$$

but in practice we implement a slightly more efficient algorithm that uses τ directly to generate the compatible descriptions.

At a high level, the algorithm draws candidate tests randomly from the generator and keeps only those tests that improve combinatorial coverage. When the algorithm keeps a test, it also removes any newly covered descriptions from the `uncovered` set. This process repeats until all descriptions are covered; thus, the final suite will have 100% coverage by construction.

While there are a few improvements that could be made to this algorithm, is is impossible to get around the central inefficiency in thinning distributions with combinatorial coverage: as more and more descriptions are covered, it gets less and less likely that the next random test will be able to cover the remaining descriptions. That being said, when this approach does eventually terminate with a test suite, the suite tends to be small. This is due in part to the close relationship between random and combinatorial testing. In many cases, random testing has been shown to do a good job of generating small test sets with high combinatorial coverage [Majumdar and Niksic 2018]. In addition, most of the best known algorithms for generating test suites with high coverage use random generation, and their size bounds are derived via the probabilistic method [Sarkar and Colbourn 2017]. This means that even though generating test suites for our case studies took a long time, the results were comparable to test suites that a more specialized algorithm might produce.

The algorithm above works for many natural test distributions, but some can cause problems. An issue occurs when the generator cannot produce terms to cover a particular description.[2] For example, consider a generator that only generates valid binary search trees, where we call a binary search tree valid when for every node in the tree, (l, x, r), all values in l are less than x, and all values in r are greater than x. Using this generator in the above algorithm would cause an infinite loop, since it would be impossible to cover

$$\diamond\mathsf{branch}(\diamond 3,\ \top,\ \diamond 1)$$

or any other description that is incompatible with the invariant.

If the user knows that their generator might have this property, *QuickACT* provides a modified version of the algorithm that uses a termination heuristic. In that case, the user sets a cutoff value

[2] A similar problem occurs in standard combinatorial testing, when certain configuration interactions are disallowed. The field of *constrained combinatorial testing* proposes promising solutions to this problem with this problem [Wu et al. 2019], and it would be interesting to see if any can generalize to the functional setting.

k, and if the algorithm runs for k iterations without increasing its coverage, it terminates and assumes that the remaining descriptions cannot be covered. Note that while the original algorithm is guaranteed to produce a 100% coverage test set, the heuristic version—even with an appropriate choice of k—does so only with high probability, since there always is a chance that k iterations go by without covering a description that should be coverable.

5.3 Case Study: System F Terms

Our first case study examines the effects of adding combinatorial testing to a highly tuned and optimized test generator for System F terms [Reynolds 1974]. We found that thinning this generator's distribution produces a test suite that can find bugs substantially faster in some cases.

Generation of well-behaved 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 [Claessen et al. 2015; Lampropoulos et al. 2017; Pałka et al. 2011], while generating C programs without undefined behavior has been used to find bugs in C compilers [Regehr et al. 2012; Yang et al. 2011]. 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 two different evaluation strategies for System F, comparing the behavior of buggy versions to a reference implementation.

Since neither standard *QuickCheck* nor *QuickACT* attempts to run tests in any particular order, we measured the *expected* number of tests required to find a bug. Consider running a suite T of tests, and assume that $B \subseteq T$ are the tests that find a bug in the system. The probability that a uniformly drawn test finds a bug is

$$\mathbf{P}_{t \in T}[t \in B] = \frac{|B|}{|T|}.$$

If we run tests without replacement until a bug is found, the expected number of tests required to find a bug is

$$\frac{|T| + 1}{|B| + 1}.$$

We call this value *mean tests to failure* (MTTF).

For the case study, we built 16 buggy versions of each evaluation relation, by injecting errors in the substitution and lifting functions. To get a feeling of the kind of errors injected, consider the standard syntax for System F with $\mathcal{U}$:

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

Let $y[x/n]$ stand for substituting in x for variable n in y, and $x \uparrow_n$ for lifting all variables above n in x. Then, for example, the standard rule for substituting a type τ for variable n inside a type abstraction $\Lambda.\, e$ would require lifting τ and incrementing the DeBruijn index of the variable being substituted by one:

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

There are two obvious 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:

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

Both of these bugs, along with 14 others, appear in our MTTF table, and a more thorough description of all 16 bugs can be found in Appendix A.

	1	2	3	4	5	6	7	8
Big Step								
Random	173.6	173.6		23.19		26.40	396.0	448.4
Combinatorial (t = 2)	-	-		-		-	-	-
Combinatorial (t = 3)	-	-		5.06		7.72	-	-
Combinatorial (t = 4)	26.34	26.34		5.06		7.17	43.99	47.94
Parallel								
Random	1,492	1,492	97.52	21.40	732.6	28.68		
Combinatorial (t = 2)	-	-	-	4.43	-	-		
Combinatorial (t = 3)	-	-	12.51	4.46	-	8.57		
Combinatorial (t = 4)	-	-	13.05	4.37	83.67	8.35		

	9	10	11	12	13	14	15	16
Big Step								
Random	58.50	19.89	92.18	5.80	17.05		37.53	590.0
Combinatorial (t = 2)	-	4.22	-	1.96	3.97		-	-
Combinatorial (t = 3)	12.65	5.44	17.84	1.85	4.32		10.07	-
Combinatorial (t = 4)	14.92	5.09	22.11	1.69	4.06		9.59	61.84
Parallel								
Random	61.34	17.26	127.2	5.36	19.52	1,117	49.00	
Combinatorial (t = 2)	-	3.97	-	1.75	3.97	-	-	
Combinatorial (t = 3)	15.11	4.27	-	1.54	4.95	-	13.95	
Combinatorial (t = 4)	16.22	4.08	38.44	1.41	4.79	-	15.46	

Table 1. MTTF for three properties across 16 buggy implementations of System F. Each cell is either the average MTTF for the property/bug combination, or "-" if the test suites do not reliably catch the bug.

For each evaluation strategy we tested whether the buggy variant produced the same results as a clean reference implementation. As a baseline, we generated 20 sets of 10,000 random test inputs each. We kept track of the number of tests in each test set that found a bug, for each each pair of a property and an implementation, and computed the mean tests to failure in the cases where the property uncovered a bug. We then generated 60 100% coverage test suites: 20 each with 2-way, 3-way, and 4-way coverage. We show some examples of the descriptions that we covered in Appendix C. We used the heuristic version of the algorithm, since some descriptions cannot be covered by the generator. Again, we computed the expected number of tests to find a bug under each strength.

Table 1 shows the MTTF results. Note that the MTTF numbers are per test suite. If more than one (5%) of the test suites did not catch the bug, we report an "-" in the table; we chose this cutoff because requiring that every test suite catch the bug would be too strict. We rely on coverage to lead us to interesting tests, but in practice it is very hard to guarantee that a bug will always be caught at a certain t. There is a short glossary of bugs in Appendix A that describes each of the 16 bugs in the table.

Our thinned test suites worked well in practice, especially the ones with complete 4-way coverage. Those test suites needed between 3× and 9× fewer tests to reveal bugs and only failed to find a bug in one case. This big performance boost makes sense: forcing tests to use a large variety of different arrangements of syntactic constructors should help reveal bugs quickly in an interpreter.

5.4 Case Study: Binary Search Trees

In our second case study, we examined another highly optimized testing setup, courtesy of a paper by John Hughes called *How to Specify it!* [Hughes 2019], and showed that *QuickACT*'s thinned test suites are competitive with random ones in this setting (though, this time, not significantly better). This increases our confidence that thinning well-tuned test generators using our combinatorial coverage metric does not damage their bug-finding power.

We examined 24 properties of binary search tree operations, which are grouped into four categories: invariant, post-condition, metamorphic, and model-based properties. Again, we used the heuristic version of the algorithm when working with binary search trees, since the generators only generate valid trees. We tested eight buggy versions of the code (labeled BST1 to BST8 in the table), with bugs ranging from test generators that generate invalid trees (and thus bugs in the generators themselves), to subtle errors in the more complex functions.

Our methodology was very similar to that of the System F example: we had $20 \times 10{,}000$ random tests and 20 full-coverage suites for each strength, and measured MTTF for each property and each bug. There were two small differences. First, some of the properties differed in type, and thus required different test suites. For example, properties testing insertion might have input type `(Int, Int, Tree)` while others might simply take a `Tree` as an input. Second, rather than making integers opaque (which would make no sense for this example), we chose to cover descriptions containing the integers zero to five.

The full MTTF tables can be found in Appendix B; we will just discuss some high-level takeaways here. We found that the majority of properties that caught a bug with pure random testing did so with *QuickACT*'s 3-way coverage test suites as well, further validating the idea that combinatorial testing does in fact help guide us to interesting inputs in some domains.

We also observed some conditions under which thinning distributions using combinatorial coverage tends to fail. One such failure in our case studies occurred with an example that had a bug in test harness itself, not the system under test. The test generator was able to generate invalid binary search trees, and thus *QuickACT* was thinning a distribution that was inappropriate for the test properties. In this case, *QuickACT* struggled to find bugs. Another issue was due to a property that checked

```
forall k t t'. union (delete k t) (delete k t') == delete k (union t t').
```

The *QuickACT* algorithm works hard to make sure that both `t` and `t'` have appropriate combinatorial coverage, and then the property computes `union t t'` whose coverage may be quite different. These sorts of properties essentially modify the test distribution after generation, so the impact of combinatorial coverage is diminished.

6 VARIATIONS

As the above case studies show, sparse tests descriptions are an attractive way to define combinatorial coverage and using that definition to generate full-coverage test suites can be very effective in practice; but there are a number of other obvious directions that seem well worth exploring. We discuss some of them here.

6.1 Representative Samples of Large Types

Perhaps it would have been possible to do combinatorial testing with ADTs by having humans decide exactly which trees to cover. A version of 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 checks

$$\texttt{if (x < 5) then ... else ...},$$

then they might choose to cover

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

That is, 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 a type like

$$\mathsf{tuple}_3(\textsc{Int},\ \mathsf{true} + \mathsf{false},\ \mathsf{true} + \mathsf{false}),$$

we instead cover the specification

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

which has 2-way descriptions like:

$$\mathsf{tuple}_3(0,\ \top,\ \mathsf{false})$$

$$\mathsf{tuple}_3(2147483647,\ \mathsf{true},\ \top)$$

In our setting, this practice might mean choosing a representative set of 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

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

which has 2-way descriptions like:

$$\mathsf{tuple}_3(\ [\,]\ ,\ \top,\ \mathsf{false})$$

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

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.

6.2 Type-Tagged Constructors

Another variation to our approach changes the way that ADTs are translated into constructor trees, making those translations easier to automatically derive. In Section 5 we talk about translations that are used primarily to translate between Haskell terms and explicit trees of constructors. Consider the two options for a translation of a list of lists of Booleans that are shown in Figure 2. The difference is subtle, but important—in the first, `Cons` and `Nil` are used for both the inner and outer lists, whereas in the second, `OuterCons` and `OuterNil` contrast `InnerCons` and `InnerNil`. 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. However, it is likely that the latter approach will produce better test sets, since more constructors means more variety of interactions to test.

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.

```
Translation
{ haskTy = M.fromList
    [ ("ListList", [ ("Cons", ["List", "ListList"])
                   , ("Nil", [])
                   ])
    , ("List", [ ("Cons", ["Bool", "List"])
               , ("Nil", [])
               ])
    , ("Bool", [("True", []), ("False", [])])
    ]
, ...
}
```

versus

```
Translation
{ haskTy = M.fromList
    [ ("ListList", [ ("OuterCons", ["List", "ListList"])
                   , ("OuterNil", [])
                   ])
    , ("List", [ ("InnerCons", ["Bool", "List"])
               , ("InnerNil", [])
               ])
    , ("Bool", [("True", []), ("False", [])])
    ]
, ...
}
```

Fig. 2. Two ways to translate of a list of lists of Booleans.

6.3 Eventual Descriptions

Another 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?

$$d \triangleq \diamond d'$$
$$d' \triangleq C(d'_1, \ldots, d'_n)$$

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 ideas.

6.4 Partial Coverage

One final variation we might consider is a different perspective on the distribution thinning process that *QuickACT* uses. Up until now, we have taken for granted the idea that reaching 100% combinatorial coverage is a useful goal. While this is the usual assumption in the combinatorial testing literature, there may be good reasons to be less strict. For example, one of the more controversial papers in the combinatorial testing literature, *Pairwise testing: A best practice that isn't* [Bach and Schroeder 2004], warns that t-way testing might create a false sense of security. Even with 100% t-way coverage *and* a reason to believe that t is an appropriate strength, it points out, there is still no guarantee that the test suite will find every bug. Thus, every testing setup has uncertainty (due to human error, choice of properties, etc.) and placing too much faith in a test suite because it gives full coverage according to some measure is naïve.

We think this cloud has a silver lining—specifically, if 100% coverage does not provide strong guarantees (which we agree is often the case), it might make sense to aim for a slightly—or much—lower level of coverage. This could significantly reduce the time it takes to generate test suites, since the major inefficiency in *QuickACT*'s algorithm comes from the long-tail probability of covering the last few descriptions. More formally, if the probability of a test covering any particular description is p, the number of tests it would take to cover p with confidence γ is

$$\frac{\log(1-\gamma)}{\log(1-p)}.$$

As γ approaches 1 and our coverage goal approaches 100%, this quantity grows without bound. All this suggests that it might make sense to make a simple modification to *QuickACT*: reduce the coverage goal from 100% to something lower if test-suite generation times are important.

7 RELATED WORK

The combinatorial testing literature is vast both in breadth and depth; we refer the reader to a survey for a detailed view of the area [Nie and Leung 2011]. 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 that achieve similar results to ours.

7.1 Generalizations of Combinatorial Testing

There has already been some work done to apply combinatorial testing to more complex and even infinite types. Salecker and Glesner [2012] extend combinatorial testing to sets of terms that are generated by a context-free grammar. Their clever approach 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 actually only defines coverage over a finite subset of the input type. By basing coverage on description size rather than term size, our approach 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 [2006]. 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. The idea of user control of coverage is interesting, but again, we prefer our system because it deals with truly infinite types.

Finally, Kuhn et al. [2012] 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. Their generation algorithm is far more efficient than ours for this specialized case.

Our idea to use regular tree expressions for coverage is due in part to Usaola et al. [2017] and Mariani et al. [2004]. The goals of these projects are very different to ours, but they do explore the idea of combinatorial coverage of regular expressions.

The idea of applying regular tree expressions to Haskell terms has also been addressed by Serrano and Hage [2016], 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 some systems outside the realm of combinatorial testing that nevertheless achieve similar results to ours.

The *SmallCheck* [Runciman et al. 2008] 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, *QuickACT* provides higher-strength combinatorial coverage without worrying about the size of each individual test. There are pros and cons for each approach: *QuickACT* can generate test suites with high-strength coverage, but it might miss some interesting small test cases; *SmallCheck* is extremely thorough with its coverage of small tests, but requires a much larger test suite in general (although *Lazy SmallCheck* ameliorates this considerably in some cases).

Rather than view combinatorial coverage as way of knowing when to stop generating tests, we could view it as a way of determining how novel a new test is. Vanilla *QuickCheck* draws each test independently from the same distribution. However, after running some test A, it might be that some future test B is better to run than a different test C. Intuitively, this is because C might re-check some cases that A has already gone over, while B covers more new ground. Coverage guided fuzzing tools like AFL capitalize on this observation, using *code coverage* as a proxy for the novelty of each new test. There has already been successful work bringing these methods to functional programming [Gill and Runciman 2007; Lampropoulos et al. 2019], but it is 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 offline, and therefore might provide a *black-box* alternative for approaches that maximize the novelty of tests relative to one-another.

Chen et al.'s *adaptive random testing* (ART) [Chen et al. 2004] aims takes the idea of comparing tests for novelty a step further. It attempts to distribute test cases more evenly over the input space, by generating a set of *candidate* random tests, and repeatedly choosing to run the test case that is farthest from any previously-run test case. Chen et al. showed that this approach finds bugs after fewer tests on average, in the programs they studied. ART does require a metric on test cases, and was first proposed for programs with numerical inputs, but Ciupa et al. [Ciupa et al. 2008] 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 [Arcuri and Briand 2011] 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. Our approach also increases the diversity of test cases, but has the advantage over ART that it does not require a metric on test cases. , or a quadratic number of comparisons when generating a test suite.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we first generalized the definition of combinatorial coverage to work with the more general idea of *test descriptions* based on regular tree expressions. Second, we presented one natural class of descriptions, *sparse test descriptions*, and showed that it can be used to find bugs in practice. And finally, we discussed some variations on this approach that we think would be fruitful.

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

Algorithms for Generating Test Suites. A large part of the combinatorial testing literature is dedicated to the theory and practice of generating test suites with 100% combinatorial coverage—in combinatorial testing parlance, *covering arrays* [Sarkar and Colbourn 2017]. Though we have not seriously addressed algorithmic issues, *QuickACT* does employ a simple algorithm for generating covering arrays—namely, selecting a suite of randomly generated tests that has full combinatorial coverage. As we mentioned, relying on random generation in this way is inefficient because it gets harder and harder to cover new descriptions as the algorithm proceeds. In general, the problem of generating covering arrays is NP-hard, but there is still extensive literature that explores algorithms for special cases like 2-way coverage [Colbourn et al. 2004] and heuristics for more general cases [Torres-Jimenez and Rodriguez-Tello 2012].

Moving forward, we plan learn from this extensive literature and find more efficient covering array algorithms for covering regular tree expression descriptions. Such algorithms would necessarily be based on heuristics, since covering ADTs is more general than covering finite parameter spaces. However, it would almost certainly be possible to find an algorithm that is significantly faster than the one that *QuickACT* currently employs.

Applicability. In our case studies, *QuickACT* proved useful for finding bugs in functional programs, and the idea of combinatorial testing in general has broad support. Still, there are a number of open questions around exactly what class of programs are amenable to testing with *QuickACT*. In the classical combinatorial testing community, there are relatively well-understood guidelines for when combinatorial testing might be expected to work well—in particular, programs that are known to use their parameters in certain restricted ways. Such guidelines are not yet clear for the extended form of combinatorial testing that *QuickACT* employs, and we would like to spend more time exploring the situations where *QuickACT* excels and those where it falls short.

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 *QuickACT*. We also may 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.

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 *QuickACT*, 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

$$\mathsf{nil}$$

$$\mathsf{cons}(\top,\ \mathsf{nil})$$

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

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 [Mista et al. 2018], would make it easy to write highly customized generators that have meticulous control over the resulting test suites.

REFERENCES

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

James Bach and Patrick J Schroeder. 2004. Pairwise testing: A best practice that isn't. In *Proceedings of 22nd Pacific Northwest Software Quality Conference*. Citeseer, 180–196.

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)*, Michael J. Maher (Ed.), Vol. 3321. Springer, 320–329. https://doi.org/10.1007/978-3-540-30502-6_23

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

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

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

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.

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.

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

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

John Hughes. 2019. How to Specify It! *20th International Symposium on Trends in Functional Programming* (2019).

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

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

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)*, M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko (Eds.), Vol. 3964. Springer, 19–38. https://doi.org/10.1007/11754008_2

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

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

Rupak Majumdar and Filip Niksic. 2018. Why is random testing effective for partition tolerance bugs? *PACMPL* 2, POPL (2018), 46:1–46:24. https://doi.org/10.1145/3158134

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)*, Manuel Núñez, Zakaria Maamar, Fernando L. Pelayo, Key Pousttchi, and Fernando Rubio (Eds.), Vol. 3236. Springer, 337–350. https://doi.org/10.1007/978-3-540-30233-9_25

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

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

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 (AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

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

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)*, Bernard Robinet (Ed.), Vol. 19. Springer, 408–423. https://doi.org/10.1007/3-540-06859-7_148

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

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

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

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)*, Marco Gavanelli and John H. Reppy (Eds.), Vol. 9585. Springer, 83–98. https://doi.org/10.1007/978-3-319-28228-2_6

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

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

Huayao Wu, Changhai Nie, Justyna Petke, Yue Jia, and Mark Harman. 2019. A Survey of Constrained Combinatorial Testing. arXiv:cs.SE/1908.02480

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 CASE STUDY 1: GLOSSARY OF BUGS

Bug #	Description
1	Incorrectly lifts variables during substitution.
2	Fails to lift lambdas during substitution.
3	Fails to decrement variables during substitution.
4	Flips a > check to < during substitution.
5	Fails to decrement an index during type-level substitution.
6	Flips a > check to < during type-level substitution.
7	Incorrectly lifts variables during type-level substitution.
8	Fails to lift lambdas during type-level substitution.
9	Fails to decrement variables during type-level substitution.
10	Fails to substitute terms when evaluating an application.
11	Fails to substitute types when evaluating an application.
12	Reverses the substitution direction when evaluating an application.
13	Fails to increment variables during substitution.
14	Fails to lift variables during substitution.
15	Fails to increment variables during type-level substitution.
16	Fails to lift variables during type-level substitution.

B CASE STUDY 2: MTTF DATA FOR BINARY SEARCH TREES

These tables contain the raw *mean tests to failure* data for our binary search tree case study.

Invariant	BST1	BST2	BST3	BST4	BST5	BST6	BST7	BST8
Arbitrary Valid								
Random		1.51						
Comb. (t = 2)		1.18						
Comb. (t = 3)		1.18						
Insert Valid								
Random		1.44						
Comb. (t = 2)		1.50						
Comb. (t = 3)		1.51						
Delete Valid								
Random		1.52						
Comb. (t = 2)		1.46						
Comb. (t = 3)		1.53						
Union Valid								
Random		1.14				1.08	1.09	
Comb. (t = 2)		1.15				1.11	1.12	
Comb. (t = 3)		1.18				1.15	1.17	

Post-condition	BST1	BST2	BST3	BST4	BST5	BST6	BST7	BST8
Insert Post								
Random	64.36	265.6	285.7					
Comb. (t = 2)	-	-	-					
Comb. (t = 3)	35.77	-	-					
Delete Post								
Random		1,613		5.06	300.7			
Comb. (t = 2)		-		4.09	-			
Comb. (t = 3)		-		5.08	-			
Find Post Present								
Random		4.79	4.81					
Comb. (t = 2)		4.80	4.73					
Comb. (t = 3)		4.66	4.78					
Find Post Absent								
Random		31.74			5.12			
Comb. (t = 2)		-			4.15			
Comb. (t = 3)		23.86			5.13			
Insert Delete Compl.								
Random		4.75		1.32				
Comb. (t = 2)		4.12		1.33				
Comb. (t = 3)		4.39		1.30				
Union Post								
Random						5.98	6.62	44.66
Comb. (t = 2)						5.19	6.36	-
Comb. (t = 3)						6.28	7.14	53.40

Metamorphic	BST1	BST2	BST3	BST4	BST5	BST6	BST7	BST8
Insert Insert								
Random	1.02	61.62	78.26					
Comb. (t = 2)	1.03	29.92	30.55					
Comb. (t = 3)	1.02	43.79	56.73					
Insert Delete								
Random		263.53	285.74	1.07				
Comb. (t = 2)		-	-	1.09				
Comb. (t = 3)		-	-	1.07				
Insert Union								
Random	1.05	6.03	6.12			2.62	2.10	8.76
Comb. (t = 2)	1.13	6.72	5.63			2.56	2.06	7.37
Comb. (t = 3)	1.12	6.33	6.36			2.64	2.14	9.13
Delete Insert								
Random	63.18	263.5		1.08	63.30			
Comb. (t = 2)	-	-		1.10	-			
Comb. (t = 3)	32.57	-		1.09	46.07			
Delete Delete								
Random				77.47	1,492			
Comb. (t = 2)				-	-			
Comb. (t = 3)				-	-			
Delete Union								
Random		163.2		4.37	64.23	5.09	4.97	36.49
Comb. (t = 2)		-		4.34	-	4.37	5.12	-
Comb. (t = 3)		-		4.71	38.68	5.40	5.09	36.30
Union Insert								
Random	1.05	6.03	6.12			2.62	2.10	8.76
Comb. (t = 2)	1.13	6.72	5.63			2.56	2.06	7.37
Comb. (t = 3)	1.12	6.33	6.36			2.64	2.14	9.13
Union Delete Insert								
Random	1.05	4.75	4.80	1.05	5.22	1.47	1.80	68.48
Comb. (t = 2)	1.13	4.94	4.76	1.09	4.59	1.54	1.78	-
Comb. (t = 3)	1.11	4.67	5.10	1.09	5.37	1.50	1.86	71.79
Union Delete Delete								
Random		163.2		4.37	64.23	5.09	4.97	36.49
Comb. (t = 2)		-		4.34	-	4.37	5.12	-
Comb. (t = 3)		-		4.71	38.68	5.40	5.09	36.30
Union Union Assoc								
Random						1.10	1.10	1.78
Comb. (t = 2)						1.16	1.17	1.78
Comb. (t = 3)						1.31	1.31	2.27
Insert Compl. for Union								
Random	1.09					1.08	1.09	
Comb. (t = 2)	1.13					1.11	1.12	
Comb. (t = 3)	1.26					1.15	1.17	

Model-Based	BST1	BST2	BST3	BST4	BST5	BST6	BST7	BST8
Insert Model								
Random	1.05	4.72	4.81					
Comb. (t = 2)	1.15	4.70	4.73					
Comb. (t = 3)	1.12	4.62	4.78					
Delete Model								
Random		31.74		1.05	5.12			
Comb. (t = 2)		-		1.08	4.15			
Comb. (t = 3)		23.86		1.06	5.13			
Union Model								
Random		1.21				1.08	1.09	1.71
Comb. (t = 2)		1.23				1.11	1.12	1.79
Comb. (t = 3)		1.26				1.15	1.17	1.91

C EXAMPLES OF 2-WAY DESCRIPTIONS FOR SYSTEM F TERMS

```
◇Con
◇Var
◇Lam(⊤, ⊤)
◇Lam(⊤, ◇Con)
◇Lam(⊤, ◇Var)
◇Lam(⊤, ◇Lam(⊤, ⊤))
◇Lam(⊤, ◇App(⊤, ⊤))
◇Lam(⊤, ◇Cond(⊤, ⊤, ⊤))
◇Lam(⊤, ◇BTrue)
◇Lam(⊤, ◇BFalse)
◇Lam(⊤, ◇TLam(⊤))
◇Lam(⊤, ◇TApp(⊤, ⊤))
◇Lam(⊤, ◇Base)
◇Lam(⊤, ◇TBool)
◇Lam(⊤, ◇Func(⊤, ⊤))
◇Lam(⊤, ◇TVar)
◇Lam(⊤, ◇Forall(⊤))
◇Lam(◇Base, ⊤)
◇Lam(◇TBool, ⊤)
◇Lam(◇Func(⊤, ⊤), ⊤)
◇Lam(◇TVar, ⊤)
◇Lam(◇Forall(⊤), ⊤)
◇App(⊤, ⊤)
◇App(⊤, ◇Con)
◇App(⊤, ◇Var)
◇App(⊤, ◇Lam(⊤, ⊤))
◇App(⊤, ◇App(⊤, ⊤))
◇App(⊤, ◇Cond(⊤, ⊤, ⊤))
◇App(⊤, ◇BTrue)
◇App(⊤, ◇BFalse)
◇App(⊤, ◇TLam(⊤))
◇App(⊤, ◇TApp(⊤, ⊤))
◇App(⊤, ◇Base)
◇App(⊤, ◇TBool)
◇App(⊤, ◇Func(⊤, ⊤))
◇App(⊤, ◇TVar)
◇App(⊤, ◇Forall(⊤))
◇App(◇Con, ⊤)
◇App(◇Var, ⊤)
◇App(◇Lam(⊤, ⊤), ⊤)
◇App(◇App(⊤, ⊤), ⊤)
◇App(◇Cond(⊤, ⊤, ⊤), ⊤)
◇App(◇BTrue, ⊤)
◇App(◇BFalse, ⊤)
◇App(◇TLam(⊤), ⊤)
◇App(◇TApp(⊤, ⊤), ⊤)
◇App(◇Base, ⊤)
◇App(◇TBool, ⊤)
◇App(◇Func(⊤, ⊤), ⊤)
◇App(◇TVar, ⊤)
```