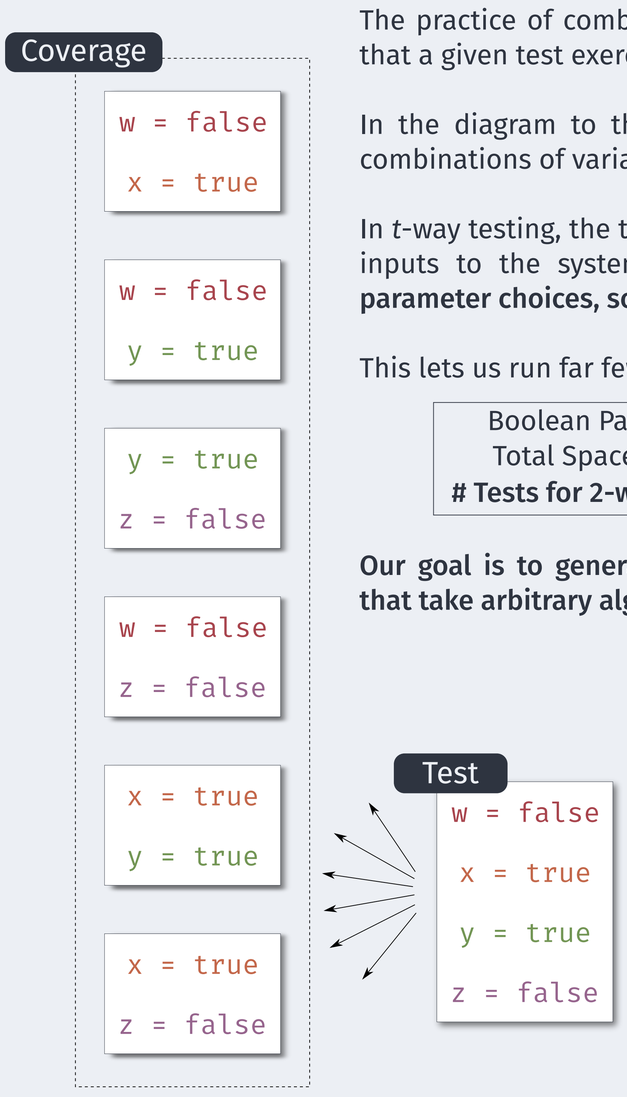
Combinatorial Testing for Algebraic Data Types

Harrison Goldstein
University of Pennsylvania

Combinatorial or "*t*-way" testing is a powerful tool for gaining confidence that a system is correct. While this approach is well studied when tests are simple sets of input parameters, there has been little exploration of its use for structured data. We propose a definition of *t*-way testing that is generalized to algebraic data types. We define *combinatorial coverage* using *regular tree expressions* as descriptions of tests, and we provide a sketch of an algorithm for generating covering test sets that are guaranteed to catch certain classes of bugs. This poster presents work in progress.

Combinatorial Testing



The practice of combinatorial testing is based on the observation that a given test exercises a **combinatorial explosion** of behaviors.

In the diagram to the left, one test exercises 6 different 2-way combinations of variable choices.

In *t*-way testing, the tester provides a carefully chosen collection of inputs to the system such that **for any *t*-way combination of parameter choices, some test exercises those choices together.**

This lets us run far fewer tests! If we want 2-way coverage:

Boolean Parameters	4	35
Total Space of Tests	16	34 billion
# Tests for 2-way Coverage	**5**	**11**

Our goal is to generalize these definitions to work for programs that take arbitrary algebraic data types as input.

Regular Tree Expressions

Instead of *t*-way choices of input parameters, we define coverage of algebraic data types in terms of **regular tree expressions**.

Regular tree expressions are like standard regular expressions, but they denote sets of constructor trees

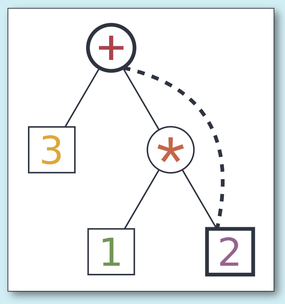
$$e \triangleq \top \mid C(e_1, \ldots, e_n) \mid e_1 + e_2 \mid \mu X.\, e \mid X$$

As an example, consider the following program type and regular tree expression

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

$$\mu X.\, \mathrm{Add}(X, X) + \mathrm{Mul}(X, X) + 1 + 2 + 3 + \ldots$$

Describing Sub-Terms



Our definition of coverage relies on the ability to describe constructors that might not be at the root of a term.

We can encode an **"eventually"** operator (written with a diamond), used as follows

$$\Diamond\mathrm{Add}(\top, \Diamond 2)$$

This tree expression matches terms like the one to the left; anywhere a diamond appears, we look **arbitrarily far down the tree** for a match.

A Generalized Definition of Coverage

We define coverage of algebraic data types using a specific subset of regular tree expressions that we call, simply, *descriptions*.

Descriptions are trees where nodes are normal constructors and edges specify potentially distant ancestor nodes.

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

We define coverage as follows: set of terms (tests) has *t*-way coverage if it covers all size-*t* *descriptions* that are compatible with the input type. (See right for more detail.)

This definition correctly specializes to the standard definition. In particular, we can generalize the example above by covering

```
data FourBools = FB Bool Bool Bool Bool
```

Size

We define the ***size*** of a description to be roughly the number of constructors used. The one exception is for types like

```
data Pair a b = Pair a b
```

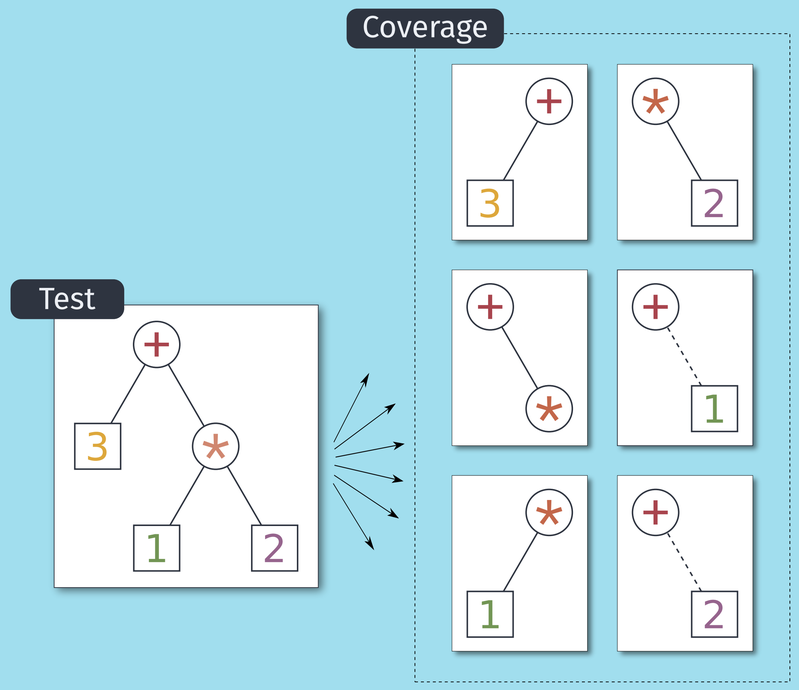
If there is only one constructor for a type, we do not count those constructors.

Compatibility

We can express a particular algebraic data type, τ, as a regular tree expression.

A description is ***compatible*** with τ if

$$\tau \cap d \neq \varnothing$$



Generating Covering Test Sets

To the right is a **greedy randomized algorithm** for generating test sets.

Though not very fast, his algorithm finds an **asymptotically optimal** covering test set with high probability. *(Not yet proven.)*

We plan to integrate a version of this algorithm with *QuickCheck*, making combinatorial testing a viable approach for testing many Haskell programs.

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

Many thanks to my advisor
Benjamin Pierce
and to
John Hughes,
Leonidas Lampropoulos,
and **Irene Yoon**
for ideas and guidance

For more about me,
please visit my website:
https://harrisongoldste.in