Thesis for the Degree of Doctor of Philosophy

Automating Black-Box Property Based Testing

Jonas Duregård

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2016

Automating Black-Box Property Based Testing
Jonas Duregård

Printed at Chalmers, Göteborg, Sweden 2016

ISBN 978-91-7597-431-6

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 4112
ISSN 0346-718X

Technical Report 132D
Department of Computer Science and Engineering
Functional Programming Research Group

Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 10 28

Abstract

Black-box property based testing tools like QuickCheck allow developers to write elegant logical specifications of their programs, while still permitting unrestricted use of the same language features and libraries that simplify writing the programs themselves. This is an improvement over unit testing because a single property can replace a large collection of test cases, and over more heavy-weight white-box testing frameworks that impose restrictions on how properties and tested code are written. In most cases the developer only needs to write a function returning a boolean, something any developer is capable of without additional training.

This thesis aims to further lower the threshold for using property based testing by automating some problematic areas, most notably generating test data for user defined data types. Writing procedures for random test data generation by hand is time consuming and error prone, and most fully automatic algorithms give very poor random distributions for practical cases.

Several fully automatic algorithms for generating test data are presented in this thesis, along with implementations as Haskell libraries. These algorithms all fit nicely within a framework called sized functors, allowing re-usable generator definitions to be constructed automatically or by hand using a few simple combinators.

Test quality is another difficulty with property based testing. When a property fails to find a counterexample there is always some uncertainty in the strength of the property as a specification. To address this problem we introduce a black-box variant of mutation testing. Usually mutation testing involves automatically introducing errors (mutations) in the source code of a tested program to see if a test suite can detect it. Using higher order functions, we mutate functions without accessing their source code. The result is a very light-weight mutation testing procedure that automatically estimates property strength for QuickCheck properties.

Contents

Acknowledgements

There are many who deserve acknowledgement for their assistance in making this thesis a reality: My supervisor Patrik Jansson, for helping me on every step of the way to making this thesis. My co-supervisor and co-author Meng Wang. My other co-authors Michał Pałka and Koen Claessen for countless productive discussions (and some less productive but very entertaining ones). My other colleagues at the department for making Chalmers a great place to work. My wife Amanda for all her encouragement and her interest in my work. Last and (physically) least, my daughter Mod for making every part of all my days better.

Introduction

Verifying the correctness of software is difficult. With software becoming ubiquitous and growing in complexity, the cost of finding and fixing bugs in commercial software has increased dramatically, to the point that it often exceeds the cost of programming the software in the first place (Tassey, 2002; Beizer, 1990; Myers and Sandler, 2004). With this in mind, automating this procedure as far as possible is highly desirable, and the focus of this thesis.

The first part of this chapter introduces relevant concepts and background, gradually zooming in on the subject of the thesis. The second part is a broad-stroke explanation of the contributions of the author to the subject.

1 Functional programming

Most of the research in this thesis relates in one way or another to functional programming. Most noticeably it tends to talk about computation as evaluation of mathematical functions. In code throughout the thesis, the functional language Haskell is used. Much of the work can be transferred to other languages and other programming paradigms, but some knowledge of Haskell is certainly helpful for readers. Readers who have experience using Haskell can skip this section.

Two features that are essential for understanding the algorithms in this thesis are algebraic data types, and lazy evaluation.

Algebraic data types Algebraic data types are defined by a name and a number of constructors. Each constructor has a name and a number of other data types it contains. Values in a type are built by applying one of the constructors to values of all the types it contains.

A very simple algebraic data type is Boolean values. It has two constructors "False" and "True", and neither constructors contain any data types so each constructor is a value in its own. In Haskell, Booleans are defined by:

```
data Bool = False | True
```

A data type of pairs of Boolean values can be defined by a data type with a single constructor containing two Bool values:

```
data BoolPair = BP Bool Bool
```

A pair of Booleans can be thus be constructed by applying the constructor BP to any two Boolean values e.g. BP True False. The BoolPair type demonstrates the algebraic nature of ADTs: Complex types are built by combining simpler ones. The algebra becomes clearer if we consider the *sum of product* view of data types: Adding a constructor to a data type corresponds to addition, extending a constructor with an additional contained type corresponds to multiplication. Thus Bool is expressed as $\mathsf{False} + \mathsf{True}$ and if we disregard the label for BoolPair it is expressed as: $(\mathsf{False} + \mathsf{True}) * (\mathsf{False} + \mathsf{True})$. Expanding this using the distributive property (same as in arithmetic) we get:

$$\mathsf{False} * \mathsf{False} + \mathsf{False} * \mathsf{True} + \mathsf{True} * \mathsf{False} + \mathsf{True} * \mathsf{True}$$

This sum corresponds directly to each of the four existing pairs of Boolean values. Quite frequently constructor names are abstracted away altogether and constructors containing no values are simply expressed as 1 giving $\mathsf{BoolPair} = (1 + 1) * (1 + 1)$.

The example types so far contain only a finite number of values. Most interesting data types are recursive, meaning some constructor directly or indirectly contains its own type. A simple example is the set of Peano coded natural numbers. Each number is either zero or the successor of another number. As a recursive Haskell type:

```
data Nat = Zero
         | Succ Nat
```

The introduction of recursion makes the algebraic view of data types slightly more complicated. It is highly desirable to have a closed form algebraic expression without recursion for data types such as Nat. This is usually expressed by extending the algebra with a least fixed point operator μ such that $\mathsf{Nat} = \mu\ \mathsf{n}.\ 1 + \mathsf{n}$. The fixed point operator can be extended to enable mutual recursion, although representations of data types used in libraries often do not support this.

Using algebraic data types Pattern matching is used to define functions on algebraic data types, by breaking a function definition into cases for each constructor and binding the contained values of constructors to variables. For instance addition of natural numbers:

```
add Zero     m = m
add (Succ n) m = Succ (add n m)
```

This mirrors the standard mathematical definition of addition, as a recursive function with a base case for zero and a recursive case for successors. In the recursive case the number contained in the Succ constructor is bound to the variable n, so it can be recursively added to m.

Datatype-generic programming Datatype-generic programming is an umbrella term for techniques that allow programmers to write functions that work on all data types, or on families of similar data types rather than on specific data types (Gibbons, 2007). A simple example could be a function that counts the number of constructors in a value or a function that generates a random value of any data type.

Type constructors and regular types A *type constructor*, not to be confused with the *data* constructors like True and Succ above, is a data type definition with variables that when substituted for specific ADTs forms a new ADT. An example is the tuple type (a, b), where a and b are variables. This is a generalization of BoolPair where $\mathsf{BoolPair} = (\mathsf{Bool}, \mathsf{Bool})$. Another example is a type of binary tree with data in each leaf:

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
```

Type constructors with at least one type variable are called polymorphic, as opposed to monomorphic types like Bool and Nat. Applying the type constructor Tree to the Nat type to yields the monomorphic type Tree Nat of trees with natural numbers in leaves. Similarly Tree (Tree Bool) is trees with trees containing trees of Booleans. In the Tree example, simple syntactic substitution of a by a monomorphic type t gives a definition of a new monomorphic type equivalent to Tree t. This means that a preprocessor could replace the type constructor Tree by a finite set of monomorphic types. This is not always the case, for instance consider the type of trees of natural numbers.

```
data Complete a = BLeaf a
                | BBranch (Complete (a, a))
```

Here Complete Nat would expand to contain Complete (Nat, Nat) which in turns contains Complete ((Nat, Nat), (Nat, Nat)) and so on, resulting in an infinite number of different applications of Complete and an infinite set of monomorphic types.

Data types like Complete are referred to as $\mathsf{non-regular}$, or nested (Bird and Meertens, 1998). Generic programming libraries often have limited support for non-regular types (Rodriguez et al., 2008).

Another example of a non-regular data type is this representation of closed lambda terms (Barendregt, 1984), with constructors for lambda abstraction, function application and De Bruijn-indexed variables (De Bruijn, 1972):

```
data Extend s = This
              | Other s
data Term s = Lam (Term (Extend s))
            | App (Term s) (Term s)
            | Var s
data Void
type Closed = Term Void
```

Here `Term s` is the type of terms with the variable scope `s`, meaning every value in `s` is allowed as a free variable. The `Extend` type constructor takes a data type and extends it with one additional value (`This`). In the `Lam` constructor, `Extend` is used to express that the body of a lambda abstraction has an additional variable in scope compared to the surrounding expression.

The type of closed expressions is `Term Void`, where `Void` is an empty data type used to signify that there are no free variables in closed expressions. Algebraically, `Void` is 0 and the expected algebraic identities hold[1], for instance the tuple type `(Void, t)` is also 0 for any type `t`.

Lazy evaluation Haskell is a lazy language. Intuitively, this means it avoids computing values that are not used. This can have performance benefits but it also gives a more expressive language, particularly it allows defining infinite values. Consider this example:

```
inf = Succ inf

isZero Zero = True
isZero (Succ n) = False
```

Here `inf` is an infinitely large number, but computing `isZero inf` terminates with `False`, because laziness prevents `inf` from being computed other than determining that it is a successor of something.

For every function, lazy evaluation introduces an equivalence relation between values: Two values are equivalent with respect to a function `f` if the parts of the values that `f` evaluate are identical. For instance `Succ Zero` and `inf` are equivalent w.r.t. `isZero`, because only the first `Succ` constructor is evaluated in both cases. A function always yields the same result for equivalent values, but values can yield the same result without being equivalent, for instance consider this example:

```
small Zero     = True
small (Succ n) = isZero n
```

Here `small` gives `True` for `Zero` and `Succ Zero` but the values are not equivalent because the evaluated parts differ.

[1] At least with a bit of "Fast and loose reasoning" (Danielsson et al., 2006).

2 Software Testing

This section gives a general introduction to the topic of software testing, focusing on QuickCheck-style property based testing. Readers who have experience using QuickCheck can skip this section.

Specification The first step in any verification effort is specification. When one claims a program is correct, it is always with respect to some specification of the desired behaviour. If the program deviates from this behaviour, there is a bug. If there is no specification, or the specification is very imprecise, the question of correctness is moot. Indeed there is often disagreement on whether a reported behaviour is a bug, a user error (using the software in unintended ways) or just a misinterpretation of the intended behaviour (Herzig, Just, and Zeller, 2013).

To avoid this, the specification must be precise enough that the correctness of a particular behaviour can be resolved without ambiguity.

In formal methods, programs are specified with techniques taken directly from mathematics and logic. Often the programs themselves are written in a similar formalism so they can be proven correct with respect to a specification, or even generated directly from the specification.

Formal methods have seen relatively limited deployment in commercial software. Often this is attributed to being time consuming (and thus expensive) and requiring highly specialized skills, although such claims are disputed by researchers in the area (Hinchey and Bowen, 2012; Knight et al., 1997).

Testing Testing is the dominant technique to establish software correctness. The general idea is that the program is executed in various concrete cases (test cases) and the behaviour is observed and evaluated based on the specification. The result is then extrapolated from correctness in the specific cases to correctness in all cases. Naturally, this extrapolation is not always sound and testing can generally not definitively exclude the presence of bugs.

The most simplistic form of testing is done manually by running the program and keeping an eye out for defects. Although this technique is often employed before software is released (alpha and beta testing), it is highly time consuming and it lacks the systematic approach that engineers tend to appreciate.

To remedy both these issues, tests are constructed as executable programs that automatically test various aspects of a software component.

Unit testing In unit testing, software is tested by a manually constructed set of executable test cases, called a test suite. Each test case consists of two parts:

- *Test data*: Settings, parameters for functions and everything else needed to run the tested software under the particular circumstances covered by this test case.
- An expected behaviour, manually constructed based on the test data and the specification.

The software is tested by running all test cases. And the programmer is alerted of any deviations from the predicted behaviour.

An example of a unit test case for a sorting function is a program that applies the tested function to $[3, 2, 1]$ (the test data) and checks that the result is $[1, 2, 3]$ (the expected behaviour). The test data can be much more complex than just the input to a function, for instance simulating user interaction.

A major advantage compared to completely manual testing is that once constructed, test cases can be executed each time a program is modified to ensure that the modification does not introduce any bugs. This technique is called regression testing (Myers and Sandler, 2004; Huizinga and Kolawa, 2007).

Another advantage is that as a separate software artefact, the test suite can be analysed to estimate and hopefully improve its bug finding capacity. For the latter purpose, adding more test cases is a good, but time consuming, start. However, test suites with many similar test cases are generally inferior to test suites with a larger spread, and there are better metrics than number of tests for estimating the quality of a test suite. One is code coverage, checking how much of the tested code is executed by the test suite. If a piece of code is not executed at all, the test suite can hardly be used to argue the correctness of that code, so low coverage is an indication of poor test suite quality. Another technique is mutation testing, that evaluates a test suite by deliberately introducing bugs in the software and checking how often the test suite detects those bugs (Offutt, 1994; DeMillo, Lipton, and Sayward, 1978).

Property Based Testing Property based testing automates unit testing by automatically building a test suite. Automatically constructing a test case requires two things:

- A way to automatically generate test data.
- An oracle that predicts (part of) the behaviour for any test data.

In property based testing, oracles are executable logical predicates, called properties. In this respect it somewhat bridges the gap between formal methods and testing. If a property is false for any generated test data, it means there is a bug, or the specification is incorrect.

To test a sorting function using property based testing, one would write a property stating that the output of the function is an ordered permutation of the input. The property is then tested by automatically generating input lists and checking that the output satisfies the property. The unit test case described above, testing that sorting $[3,2,1]$ gives $[1,2,3]$, is one of the possible test cases generated. For more complicated test data, like user interaction, both generators and properties can be much more complicated.

An advantage of property based testing is that properties are often useful as specifications, providing a short, precise and general definition of the expected behaviour.

One kind of properties is comparison to reference implementations. In these we have a functionally equivalent (but perhaps less efficient) implementation of the tested software. The property is that for any test data, the tested software yields the same result as the reference implementation. For instance an implementation of the merge-sort algorithm can be tested against the slower but simpler insertion sort algorithm (the reference implementation). In this case the reference implementation acts as specification of the tested function.

A reference implementation gives a complete specification, but weaker properties can also be used for meaningful testing. For instance a general property can be stated that a function does not crash for any input, providing a kind of fuzz-testing (Takanen, Demott, and Miller, 2008). As a specification, this is clearly incomplete, but it requires no effort to write and is useful as a minimal requirement. Another example of a useful but incomplete property is that the result of a sorting function is ordered.

Black-box Property Based Testing Black-box tools analyse software without directly accessing its source code. Tools that do access the source code are called white-box. The software is (figuratively) considered a box accepting inputs and providing output through a certain interface. In black-box tools what happens inside the box cannot be observed. A tool that applies a function to certain inputs and analyses the output is an example of a black-box tool.

For white-box tools, the inner working of the box are known (typically from access to the source code). This means white-box tools can do more powerful analysis, including simulating execution or transforming the program in various ways. But it also makes the tools more complex compared to black-box tools, and white-box tools often impose limitations on the structure and language features of analysed programs.

In black-box property based testing tools, properties themselves are black-box, typically in the form of Boolean functions. As such, the tool has no knowledge at all of the software being tested, not even its interface. As an example, a property in Haskell could be as follows:

```
prop_insert :: [Int] → Int → Bool
```

This type signature is all the testing tool knows of the property, and its task is simply to find a counterexample (a list and a number for which the property is false). Presumably the property tests some Haskell function, but even this assumption may be false since black-boxing allows executing code compiled from other languages.

From the perspective of the developer implementing prop_insert, this black-boxing gives a property language that is powerful, flexible and familiar to the programmer, overcoming many of the problems associated with formal methods. Reference implementations, logical connectives and other means of specification can be mixed seamlessly in properties. From the perspective of the testing framework the property is just a black box where test data goes in and a true/false result comes out.

This approach is well suited for languages with higher order functions, where properties can be written as functions and passed to a testing driver that deals with generating test data and presenting the results to the user. Different test frameworks for black-box Property Based Testing differ mainly in how test data is generated.

Generating random test data The most well known testing framework for functional programming is QuickCheck, described by Claessen and Hughes, (2000). One of the foremost merits of QuickCheck is the ease with which properties are defined and the short step from a high level specification to an executable test suite. The simplest predicates are just functions from input data to Booleans. For instance to test the relation between the reverse function on strings and string concatenation we define the following Haskell function:

```
prop_RevApp :: String → String → Bool
prop_RevApp xs ys = reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Both parameters of the function are implicitly universally quantified. In other words, we expect the property to be true for any two strings we throw at it. To test the property we pass it to the QuickCheck test driver (here using the GHCi Haskell interpreter):

```
Main> quickCheck prop_RevApp
OK! passed 100 tests.
```

As the output suggests, QuickCheck generated 100 test cases by applying prop_RevApp to 100 pairs of strings, and the property held in all cases.

The strings, like all QuickCheck test data, were generated randomly. Data types are associated with default random generator using a type class (called Arbitrary), and the library includes combinators to build generators for user defined types.

While writing properties for QuickCheck usually does not require skills beyond what can be expected of any programmer, this can sadly not be said about writing test data generators. Generators are mostly compositional: To generate a pair of values, first generate a random left component then a random right component. If there are multiple choices, assign each choice a probability and choose one based on those. But most interesting data types are recursive, which complicates things. When writing generators for such types, the user must ensure termination and reasonable size of generated values. The library provides several tools for making this easier. This makes generator definitions quite complicated, and every choice made in designing them impacts the distribution of generated values in ways that are hard to predict.

In the end, this means that when a property passes it is difficult to verify that it is not due to some flaw in the random generator masking a bug by rarely or never generating the test data required to find it. This uncertainty can be mitigated somewhat by running more tests, but if there is a serious flaw in the generator additional tests will solve it. The QuickCheck library also provides some tools for manually inspecting the generated test data, but that is time consuming and unreliable for detecting flaws.

The small scope hypothesis A common observation in software testing is that if a program fails to meet its specification, there is typically a small input that exhibits the failure (by some definition of small). The small scope hypothesis states that it is at least as effective to exhaustively test a class of smaller values (the small scope) as it is to randomly or manually select test cases from a much larger scope. The Haskell libraries SmallCheck and Lazy SmallCheck (Runciman, Naylor, and Lindblad, 2008) applies the small scope hypothesis to Haskell programs, and argues that most bugs can be found by exhaustively testing all values below a certain depth limit. The depth of a value is the largest number of nested constructor applications required to construct it. So in a value like `Cons False (Cons True Nil)` the depth is 2 because `True` and `Nil` are nested inside `Cons`, which in turn is nested inside another `Cons`. Exhaustive testing by depth has at least two advantages over random generation:

- Generators are mechanically defined. There is usually no manual work involved in writing the enumeration procedure for a data type; they tend to mirror the definition of the type itself.
- When a property succeeds, the testing driver gives a concise and meaningful description of coverage: The depth limit to which it was

able to exhaustively test.

The disadvantage is that the number of values can grow extremely fast and exhaustively testing even to a low depth may not be feasible. Typically the number of values is doubly exponential in the depth. The SmallCheck library provides combinators to mitigate this by manually changing the depth cost of selected constructors e.g. certain constructors can increase the "depth" of values by two instead of one. Unfortunately this procedure partly eliminates both the advantages described above: Generator definition is no longer mechanical and it is no longer easy to understand the inclusion criteria of a test run.

3 Contributions

The main contribution of this thesis is a set of algorithms for black-box property based testing of functional programs. Specifically for automatic test data generation based on definitions of algebraic data types. The algorithms differ in their basic approaches: QuickCheck-style random selection or SmallCheck-style bounded exhaustive enumeration. The other important divider is if they can detect (and avoid) equivalent test cases. The algorithms are:

- FEAT: Efficient random access enumeration of values in a data type. Combines exhaustive and random enumeration (but does not detect equivalent values).
- NEAT: Efficient bounded exhaustive enumeration of non-equivalent inputs to a lazy predicate.
- Uniform selection from non-equivalent values of a lazy predicate[2].

Each algorithm is covered in its own chapter of the thesis. As a secondary contribution we present black-box mutation testing, a technique to automate another aspect of property based testing: measuring test suite quality.

Size based algebraic enumerations Each algorithm uses its own representation of enumerable sets, but all three algorithms provide the same basic operations for defining the enumerations.

The most basic operations are union and products (corresponding to sums and products in data types) and a unary operator called `pay` to represent the application of (any) constructor by increasing the size ("cost") of all values in a given enumeration. This direct correspondence to algebraic

[2] As of yet, the uniform selection algorithm does not have a catchy name like `FEAT` and `NEAT`.

data types means enumerations can be constructed automatically from type definitions.

An important feature of these operations is support for recursively defined enumerations without using a least fixed point operator. The only requirement is that any cyclic definition must contain at least one application of pay. With pay used to represent constructor application, this requirement is automatically respected for all enumerations derived from Haskell data type definitions. As a consequence, mutually recursive and non-regular types (such as the data types for closed lambda terms presented earlier) can be enumerated without restrictions.

Paper I:
FEAT: Functional Enumeration of Algebraic Types The first chapter covers FEAT: An algorithm for efficient functional enumerations and a library based on this algorithm. Initially FEAT was intended to overcome the difficulty of writing random generators for large systems of algebraic types such as syntax trees in compilers (but it is useful for smaller data types as well). We identified two problems with using existing tools (QuickCheck and SmallCheck) on these types:

- Writing random generators by hand for large systems of types is painstaking, and so is verifying their statistical soundness.
- The small scope hypothesis does not apply directly to large ADTs.

The second issue is demonstrated in the paper. Applying SmallCheck to properties that quantify over a large AST, in our case that of Haskell itself with some extensions, proved insufficient for the purpose of finding bugs. The reason is the extreme growth of the search space as depth increases, which prevents SmallCheck from reaching deep enough to find bugs.

To overcome these problems we provide *functional enumerations*. We consider an enumeration as a sequence of values. In serial enumerations like SmallCheck, this sequence is an infinite list starting with small elements and moving to progressively larger ones. For example the enumeration of the values of the closed lambda terms are:

```
Lam (Var This)
Lam (Lam (Var This))
Lam (Lam (Lam (Var This)))
Lam (Lam (Var (Other This)))
Lam (App (Var This) (Var This))
[...]
```

A functional enumeration is instead characterized by an efficient indexing function that computes the value at a specified index of the sequence, es-

sentially providing random access to enumerated values. The difference is best demonstrated by an example:

```
Main> index (10^100) :: Closed
Lam (App (Lam (Lam (Lam (App (Lam (Lam (App (Lam (Var This))
[...]
(Lam (Lam (Var This))))
```

This computes the value at position 10^{100} in the enumeration of the Closed type (with [...] replacing around 20 lines of additional output). Clearly accessing this position in a serial enumeration is not practical.

This "random access" allows Functional enumerations to be used both for SmallCheck-style exhaustive testing and QuickCheck-style random testing. In the latter case it guarantees uniform distribution over values of a given size.

We show in a case study that this flexibility helps discover bugs that cannot practically be reached by the serial enumeration provided by SmallCheck.

Motivating example An area where FEAT really shines is properties that do not have complex preconditions on test data. This includes syntactic properties of languages for instance (quantifying over all syntactically correct programs) but usually not semantic properties (quantifying over all type correct programs). For instance, suppose we have a pretty printer and parser for closed lambda terms. We can test the property that parsing a printed term gives gives the original term:

$$\begin{aligned}&\mathsf{parse} :: \mathsf{String} \to \mathsf{Maybe\ Closed}\\&\mathsf{print} :: \mathsf{Closed} \to \mathsf{String}\\&\mathsf{prop\_cycle} :: \mathsf{Closed} \to \mathsf{Bool}\\&\mathsf{prop\_cycle\ t} = \mathsf{parse}\ (\mathsf{print\ t}) \equiv \mathsf{Just\ t}\end{aligned}$$

A default enumeration for Closed can be derived automatically by FEAT (or defined manually). FEAT can then test prop_cycle both exhaustively for inputs up to a given size and for random inputs of larger sizes.

For instance one could instruct FEAT to test at most 100000 values of each size. If there are fewer values of any given size it tests it exhaustively, if there are more it can pick values randomly or evenly over the sequence of values.

FEAT is also en example of an "embarrassingly parallel" algorithm: N parallel processes can exhaustively search for a counterexample simply by selecting every Nth value from the enumeration (starting on a unique number). This requires no communication between the processes (other than a unique initial number) and work can be distributed over different machines without any significant overhead.

Paper II:
NEAT: Non-strict Enumeration of Algebraic Data Types As mentioned, FEAT works best for properties without preconditions. Implications like $\mathsf{p}\ \mathsf{x} \Rightarrow \mathsf{q}\ \mathsf{x}$, where p is a false for almost all values are sometimes problematic because FEAT spends almost all its time testing the precondition and very rarely tests q which is the actual property. This is especially true for preconditions that recursively check a condition for every part of x. For instance checking that every node in a binary tree satisfies the heap invariant or type checking a lambda term. In these cases the probability of $\mathsf{p}\ \mathsf{x}$ for a random x decreases dramatically with the size of x, since each constructor in x is a potential point of failure.

This means that large randomly generated values have a very small chance of satisfying p, and as such they are not useful to test the implication property. Exhaustively enumerating small values eventually finds values that satisfy the condition, but the search space can be too large.

For this kind of predicates, $\mathsf{p}\ \mathsf{x}$ tends to terminate with a false result directly upon finding a single point in x that falsifies the predicate. In a language with lazy evaluation, large parts of x may not have been evaluated. In such cases there is a large set of values equivalent to x (all values that differs from x only in the un-evaluated parts). FEAT cannot detect these equivalences, and tends to needlessly test several equivalent values.

A simple example is a property that takes an ordered list and an element and checks that inserting the element in the list gives an ordered list:

$$
\begin{aligned}
&\mathsf{insert} :: \mathsf{Int} \to [\mathsf{Int}] \to [\mathsf{Int}] \\
&\mathsf{ordered} :: [\mathsf{Int}] \to \mathsf{Bool} \\
&\mathsf{prop\_insert} :: ([\mathsf{Int}], \mathsf{Int}) \to \mathsf{Bool} \\
&\mathsf{prop\_insert}\ (\mathsf{xs}, \mathsf{x}) = \mathsf{ordered}\ \mathsf{xs} \Rightarrow \mathsf{ordered}\ (\mathsf{insert}\ \mathsf{x}\ \mathsf{xs})
\end{aligned}
$$

The predicate ordered yields false on the first out of order element in the list. So executing $\mathsf{ordered}\ [1,2,1,0]$ and $\mathsf{ordered}\ [1,2,1,1]$ is the exact same procedure; the inputs are equivalent with respect to ordered. Unlike FEAT, NEAT never applies a predicate to more than one value in each equivalence class.

NEAT is inspired by Lazy SmallCheck (Runciman, Naylor, and Lindblad, 2008), a variant of SmallCheck that also uses laziness to avoid testing equivalent values. Here is a summary of how NEAT relates to FEAT and Lazy SmallCheck:

- NEAT is size based like FEAT and unlike Lazy SmallCheck (Lazy SmallCheck is based on depth).

- NEAT provides bounded exhaustive search like Lazy SmallCheck and FEAT, but no random access like FEAT does.

- NEAT avoids testing equivalent values like Lazy SmallCheck and unlike FEAT.

- NEAT is more efficient than Lazy SmallCheck, with a worst case complexity linear in the number of total non-equivalent values within the size bound (Lazy SmallCheck is linear in the number of partial values, a strictly greater set).

In the worst case, when the predicate is fully eager so each value has its own equivalence class, the number of executions of the predicate is the same for NEAT as it is for FEAT (but NEAT lacks the possibility of random selection). In many cases NEAT is a lot faster, and in some cases the number of equivalence classes is logarithmic or even constant in the total number of values.

The paper also discusses several algorithms called *conjunction strategies*. These are based on the observation that for logical connectives (not just conjunction) the predicates $p \wedge q$ differ in laziness from $q \wedge p$ although they are logically equivalent. Conjunction strategies are intended to increase the laziness of predicates, thus reducing the search space, by strategically flipping the order in which operands of conjunctions are evaluated.

Motivating example One could argue that in the example of sorted lists, it is easy to circumvent the problem by generating sorted lists directly, or by sorting the list before using it. An example where this is much harder is generating type correct closed lambda terms (as defined earlier), for instance to test a normalization function as such:

```
typeCheck :: Closed → Bool
isNormal  :: Closed → Bool
normalize :: Closed → Closed

prop_evaluates :: Closed → Bool
prop_evaluates c = typeCheck c ⇒ isNormal (normalize c)
```

Generating well typed terms is very difficult (Pałka, 2012). Type checking is also an example of a lazy predicate, likely to fail early and with large classes of equivalent terms.

This means that NEAT outperforms FEAT in exhaustive search, and is capable of verifying the predicate for larger sizes using fewer tests of the predicate. Direct comparison to Lazy SmallCheck is difficult because it uses depth instead of size, but preliminary experiments and theoretical analysis both indicate that NEAT is more capable of finding counterexamples.

Paper III:
Generating Constrained Random Data with Uniform Distribution With FEAT offering exhaustive enumeration and random sampling without detecting equivalent values, and NEAT offering exhaustive enumeration of non-equivalent values, this paper addresses the final piece of the puzzle: Random sampling of non-equivalent values.

The algorithm uses the same counting algorithm as FEAT to select a value of a given size uniformly at random. If it does not satisfy the predicate it is excluded from future selection along with all equivalent values. Then a new value is sampled until a satisfying value is found (or the search space is exhausted).

The algorithm does not offer a functional enumeration of the satisfying values (we cannot find the n:th satisfying value), but when a value is found it is guaranteed to have been uniformly selected from the set of satisfying values.

The foremost problem with the algorithm is memory usage. The algorithm starts with a very compact representation of all values (like a representation of an algebraic data type). This representation tends to grow in memory usage as values are removed from it (because of decreased sharing). For eager predicates this quickly exhausts the memory of the machine, but for a sufficiently lazy predicates it can find randomly selected values far beyond what FEAT can find.

Motivating example Although NEAT can be used to find all type correct lambda terms up to a given size, it relies on the small scope hypothesis for finding counterexamples. But experimentation with FEAT indicates that exhaustively searching a small scope is not always sufficient to find a counterexample.

The algorithm we present complements NEAT in these cases by generating random type correct values of larger size.

Concretely, one could use NEAT to exhaustively test to the largest size possible in a given amount of time, then select e.g. 2000 values of each size beyond that until a second time-out is reached (or a memory limit is reached).

Paper IV:
Black-box Mutation Testing This paper concerns automating another aspect of property based testing, namely evaluating test suite quality. Specifically it measures the strength of a property as specification of a tested function. The intended application is finding weaknesses in property suites and increasing confidence in strong property suite.

The basic idea is that all valid properties of a function f can be placed on an approximate scale from tautologies or near tautologies (like f x ≡ f x) to complete specifications (e.g. comparing to a reference implementation f x ≡ ref x). In between these extremes we have properties that say *something*, but not everything about the behaviour of f.

The problem we address is that after testing a property p, even using all the clever algorithms in this thesis to generate test data, if no counterexample is found there is no direct way of knowing where on this spectrum p is. In fact, QuickCheck gives identical output for the tautological property and the reference implementation property.

The question we ask to measure the specification strength of a property is "How many functions other than f does this property hold for". For the tautological property, the answer is "all other functions", and for the reference implementation it is "no other functions". For properties between these two on the spectrum the answer is "some other functions".

Since most properties tend to be somewhere between the two extremes, we need a more fine grained measure than just complete/tautological/neither. We want to test the property on a carefully chosen set of other functions, and report how many of the functions pass the test (lower number means higher strength). For most properties a completely random function is unlikely to satisfy it, so functions in the set should be similar but not identical to f.

The idea of evaluating the strength of a test suite by running it on modified versions of the tested functions is not a new one, it is called mutation testing (and the modified functions are called mutants). The likelihood that a mutant is "killed" by a test suite is called a mutation score. Traditionally, mutation testing is an inherently white-box procedure, with mutants generated by modifying the source code of the function.

In this paper, we toy with the idea of black-box mutation testing. In a functional language, functions can be modified much like any other values (for instance by composing them with other functions).

This is a promising technique, with some unique challenges and advantages compared to traditional white-box mutation testing. In some ways our approach is to traditional mutation testing what QuickCheck is to theorem provers: A very light weight alternative providing a less rigorous solution at a fraction of the resource expenditure.

Most importantly our approach has the general advantage of black-boxing: It supports all language features and extensions. If a function can be compiled it can be mutated. Developing a white-box mutation testing tool to this standard for a language like Haskell would require a massive engineering effort as well as substantial research on how to mutate all the individual language constructs (preserving type correctness).

As a proof of concept, we implement a mutation testing framework for

QuickCheck in less than a hundred lines of code and show that it is capable of providing useful measurements of property quality.

Paper I

FEAT: Functional Enumeration of Algebraic Types

This chapter is an adapted version of a paper originally published in the proceedings of the 2012 Haskell Symposium under the same title.

FEAT: Functional Enumeration of Algebraic Types

Jonas Duregård, Patrik Jansson, Meng Wang

Abstract

In mathematics, an enumeration of a set S is a bijective function from (an initial segment of) the natural numbers to S. We define "functional enumerations" as efficiently computable such bijections. This paper describes a theory of functional enumeration and provides an algebra of enumerations closed under sums, products, guarded recursion and bijections. We partition each enumerated set into numbered, finite subsets.

We provide a generic enumeration such that the number of each part corresponds to the size of its values (measured in the number of constructors). We implement our ideas in a Haskell library called `testing-feat`, and make the source code freely available. Feat provides efficient "random access" to enumerated values. The primary application is property-based testing, where it is used to define both random sampling (for example QuickCheck generators) and exhaustive enumeration (in the style of SmallCheck). We claim that functional enumeration is the best option for automatically generating test cases from large groups of mutually recursive syntax tree types. As a case study we use Feat to test the pretty-printer of the Template Haskell library (uncovering several bugs).

1 Introduction

Enumeration is used to mean many different things in different contexts. Looking only at the Enum class of Haskell we can see two distinct views: The list view and the function view. In the list view succ and pred let us move forward or backward in a list of the form $[\mathsf{start}\,..\,\mathsf{end}]$. In the function view we have a bijective function $\mathsf{toEnum} :: \mathsf{Int} \to \mathsf{a}$ that allows direct access to any value of the enumeration. The Enum class is intended for enumeration types (types whose constructors have no fields), and some of the methods (fromEnum in particular) of the class make it difficult to implement efficient instances for more complex types.

The list view can be generalised to arbitrary types. Two examples of such generalisations for Haskell are SmallCheck (Runciman, Naylor, and Lindblad, 2008) and the less well-known enumerable package. SmallCheck implements a kind of $\mathsf{enumToSize} :: \mathbb{N} \to [\mathsf{a}]$ function that provides a finite list of all values bounded by a size limit. Enumerable instead provides only a lazy $[\mathsf{a}]$ of all values.

Our proposal, implemented in a library called Feat, is based on the function view. We focus on an efficient bijective function $\mathsf{index}_{\mathsf{a}} :: \mathbb{N} \to \mathsf{a}$, much

like toEnum in the Enum class. This enables a wider set of operations to explore the enumerated set. For instance we can efficiently implement $\mathsf{enumFrom} :: \mathbb{N} \to [\mathsf{a}]$ that jumps directly to a given starting point in the enumeration and proceeds to enumerate all values from that point. Seeing it in the light of property based testing, this flexibility allows us to generate test cases that are beyond the reach of the other tools.

As an example usage, imagine we are enumerating the values of an abstract syntax tree for Haskell (this example is from the Template Haskell library). Both Feat *and* SmallCheck can easily calculate the value at position 10^5 of their respective enumerations:

```
*Main> index (10^5) :: Exp
AppE (LitE (StringL "")) (CondE (ListE []) (ListE [])
                                           (LitE (IntegerL 1)))
```

But in Feat we can also do this:

```
*Main> index (10^100) :: Exp
ArithSeqE (FromR (AppE (AppE (ArithSeqE (FromR (ListE [])))
... -- and 20 more lines!
```

Computing this value takes less than a second on a desktop computer. The complexity of indexing is (worst case) quadratic in the size of the selected value. Clearly any simple list-based enumeration would never reach this far into the enumeration.

On the other hand QuickCheck (Claessen and Hughes, 2000), in theory, has no problem with generating large values. However, it is well known that reasonable QuickCheck generators are really difficult to write for mutually recursive data types (such as syntax trees). Sometimes the generator grows as complex as the code to be tested! SmallCheck generators are easier to write, but fail to falsify some properties that Feat can.

We argue that functional enumeration is the only available option for automatically generating useful test cases from large groups of mutually recursive syntax tree types. Since compilers are a very common application of Haskell, Feat fills an important gap left by existing tools.

For enumerating the set of values of typeF we partition a into numbered, finite subsets (which we call *parts*). The number associated with each part is the size of the values it contains (measured in the number of constructors). We can define a function for computing the cardinality for each part i.e. $\mathsf{card_a} :: \mathsf{Part} \to \mathbb{N}$. We can also define $\mathsf{select_a} :: \mathsf{Part} \to \mathbb{N} \to \mathsf{a}$ that maps a part number p and an index i within that part to a value of type a and size p. Using these functions we define the bijection that characterises our enumerations: $\mathsf{index_a} :: \mathbb{N} \to \mathsf{a}$.

We describe (in §2) a simple theory of functional enumeration and provide an algebra of enumerations closed under sums, products, guarded

recursion and bijections. These operations make defining enumerations for Haskell data types (even mutually recursive ones) completely mechanical. We present an efficient Haskell implementation (in §3).

The efficiency of Feat relies on memoising (of meta information, not values) and thus on *sharing*, which is illustrated in detail in §3 and §4.

We discuss (in §5) the enumeration of data types with invariants, and show (in §6) how to define random sampling (QuickCheck generators) and exhaustive enumeration (in the style of SmallCheck) and combinations of these. In §7 we show results from a case study using Feat to test the pretty printer of the Template Haskell library and some associated tools.

2 Functional enumeration

For the type E of functional enumerations, the goal of Feat is an efficient indexing function $\mathsf{index} :: \mathsf{E}\ \mathsf{a} \to \mathbb{N} \to \mathsf{a}$. For the purpose of property based testing it is useful with a generalisation of index that selects values by giving size and (sub-)index. Inspired by this fact, we represent the enumeration of a (typically infinite) set S as a *partition* of S, where each part is a numbered finite subset of S representing values of a certain size. Our theory of functional enumerations is a simple algebra of such partitions.

Definition 1 (Functional Enumeration)**.** A functional enumeration of the set S is a partition of S that is

- *Bijective*, each value in S is in exactly one part (this is implied by the mathematical definition of a partition).
- *Part-Finite*, every part is finite and ordered.
- *Countable*, the set of parts is *countable*.

□

The countability requirement means that each part has a number. This number is (slightly simplified) the size of the values in the part. In this section we show that this algebra is closed under disjoint union, Cartesian product, bijective function application and guarded recursion. In Table 1.1 there is a comprehensive overview of these operations expressed as a set of combinators, and some important properties that the operations guarantee (albeit not a complete specification).

To specify the operations we make a tiny proof of concept implementation that does not consider efficiency. In §3 and §4 we show an efficient implementation that adheres to this specification.

Enumeration combinators:

$$
\begin{array}{ll}
\mathsf{empty} & :: \mathsf{E\ a} \\
\mathsf{singleton} & :: \mathsf{a} \to \mathsf{E\ a} \\
(\oplus) & :: E\ \mathsf{a} \to E\ \mathsf{b} \to E\ (\mathsf{Either\ a\ b}) \\
(\otimes) & :: E\ \mathsf{a} \to E\ \mathsf{b} \to E\ (\mathsf{a}, \mathsf{b}) \\
\mathsf{biMap} & :: (\mathsf{a} \to \mathsf{b}) \to E\ \mathsf{a} \to E\ \mathsf{b} \\
\mathsf{pay} & :: \mathsf{E\ a} \to \mathsf{E\ a}
\end{array}
$$

Properties:

$$
\begin{array}{ll}
\mathsf{index}\ (\mathsf{pay\ e})\ \mathsf{i} & \equiv \mathsf{index\ e\ i} \\
(\mathsf{index\ e\ i}_1 \equiv \mathsf{index\ e\ i}_2) & \equiv (\mathsf{i}_1 \equiv \mathsf{i}_2) \\
\mathsf{pay}\ (\mathsf{e}_1 \oplus \mathsf{e}_2) & \equiv \mathsf{pay\ e}_1 \oplus \mathsf{pay\ e}_2 \\
\mathsf{pay}\ (\mathsf{e}_1 \otimes \mathsf{e}_2) & \equiv \mathsf{pay\ e}_1 \otimes \mathsf{e}_2 \\
 & \equiv \mathsf{e}_1 \otimes \mathsf{pay\ e}_2 \\
\mathsf{fix\ pay} & \equiv \mathsf{empty} \\
\mathsf{biMap\ f}\ (\mathsf{biMap\ g\ e}) & \equiv \mathsf{biMap}\ (\mathsf{f} \circ \mathsf{g})\ \mathsf{e} \\
\mathsf{singleton\ a} \otimes \mathsf{e} & \equiv \mathsf{biMap}\ (\mathsf{a},)\ \mathsf{e} \\
\mathsf{e} \otimes \mathsf{singleton\ b} & \equiv \mathsf{biMap}\ (, \mathsf{b})\ \mathsf{e} \\
\mathsf{empty} \oplus \mathsf{e} & \equiv \mathsf{biMap\ Right\ e} \\
\mathsf{e} \oplus \mathsf{empty} & \equiv \mathsf{biMap\ Left\ e}
\end{array}
$$

Table 1.1: Operations on enumerations and selected properties

Representing parts The parts of the partition are finite ordered sets. We first specify a data type Finite a that represents such sets and a minimal set of operations that we require. The data type is isomorphic to finite lists, with the additional requirement of unique elements. It has two consumer functions: computing the cardinality of the set and indexing to retrieve a value.

$$\begin{aligned}&\mathsf{card}_F :: \mathsf{Finite\ a} \to \mathbb{N}\\ &(!!_F) :: \mathsf{Finite\ a} \to \mathbb{N} \to \mathsf{a}\end{aligned}$$

As can be expected, $\mathsf{f} !!_F \mathsf{i}$ is defined only for $\mathsf{i} < \mathsf{card}_F\ \mathsf{f}$. We can convert the finite set into a list:

$$\begin{aligned}&\mathsf{values}_F :: \mathsf{Finite\ a} \to [\mathsf{a}]\\ &\mathsf{values}_F\ \mathsf{f} = \mathsf{map}\ (\mathsf{f}!!_F)\ [0\,..\,\mathsf{card}_F\ \mathsf{f} - 1]\end{aligned}$$

The translation satisfies these properties:

$$\begin{aligned}&\mathsf{card}_F\ \mathsf{f} \equiv \mathsf{length}\ (\mathsf{values}_F\ \mathsf{f})\\ &\mathsf{f} !!_F \mathsf{i} \quad \equiv (\mathsf{values}_F\ \mathsf{f}) !!\ \mathsf{i}\end{aligned}$$

For constructing Finite sets, we have disjoint union, product and bijective function application. The complete interface for building sets is as follows:

$$\begin{aligned}&\mathsf{empty}_F &&:: \mathsf{Finite\ a}\\ &\mathsf{singleton}_F &&:: \mathsf{a} \to \mathsf{Finite\ a}\\ &(\oplus_F) &&:: \mathsf{Finite\ a} \to \mathsf{Finite\ b} \to \mathsf{Finite\ (Either\ a\ b)}\\ &(\otimes_F) &&:: \mathsf{Finite\ a} \to \mathsf{Finite\ b} \to \mathsf{Finite\ (a, b)}\\ &\mathsf{biMap}_F &&:: (\mathsf{a} \to \mathsf{b}) \to \mathsf{Finite\ a} \to \mathsf{Finite\ b}\end{aligned}$$

The operations are specified by the following simple laws:

$$\begin{aligned}&\mathsf{values}_F\ \mathsf{empty}_F &&\equiv [\,]\\ &\mathsf{values}_F\ (\mathsf{singleton}_F\ \mathsf{a}) &&\equiv [\mathsf{a}]\\ &\mathsf{values}_F\ (\mathsf{f}_1 \oplus_F \mathsf{f}_2) &&\equiv \mathsf{map\ Left}\ (\mathsf{values}_F\ \mathsf{f}_1) +\!\!+ \mathsf{map\ Right}\ (\mathsf{values}_F\ \mathsf{f}_2)\\ &\mathsf{values}_F\ (\mathsf{f}_1 \otimes_F \mathsf{f}_2) &&\equiv [(\mathsf{x}, \mathsf{y}) \mid \mathsf{x} \leftarrow \mathsf{values}_F\ \mathsf{f}_1, \mathsf{y} \leftarrow \mathsf{values}_F\ \mathsf{f}_2]\\ &\mathsf{values}_F\ (\mathsf{biMap}_F\ \mathsf{g}\ \mathsf{f}) &&\equiv \mathsf{map}\ \mathsf{g}\ (\mathsf{values}_F\ \mathsf{f})\end{aligned}$$

To preserve the uniqueness of elements, the operand of biMap_F must be bijective. Arguably the function only needs to be injective, it does not need to be surjective in the type b. It is surjective into the resulting set of values however, which is the image of the function g on f.

A type of functional enumerations Given the countability requirement, it is natural to define the partition of a set of type a as a function from $\mathbb{N}$ to Finite a. For numbers that do not correspond to a part, the function

returns the empty set (empty_F is technically not a part, a partition only has non-empty elements).

$$\begin{aligned}
&\textbf{type } \mathsf{Part} = \mathbb{N}\\
&\textbf{type } E\ \mathsf{a} = \mathsf{Part} \to \mathsf{Finite\ a}\\
&\mathsf{empty} :: E\ \mathsf{a}\\
&\mathsf{empty} = \mathsf{const}\ \mathsf{empty}_F\\
&\mathsf{singleton} :: \mathsf{a} \to E\ \mathsf{a}\\
&\mathsf{singleton}\ \mathsf{a}\ 0 = \mathsf{singleton}_F\ \mathsf{a}\\
&\mathsf{singleton}\ _\ _ = \mathsf{empty}_F
\end{aligned}$$

Indexing in an enumeration is a simple linear search:

$$\begin{aligned}
&\mathsf{index} :: E\ \mathsf{a} \to \mathbb{N} \to \mathsf{a}\\
&\mathsf{index}\ \mathsf{e}\ \mathsf{i}_0 = \mathsf{go}\ 0\ \mathsf{i}_0\ \textbf{where}\\
&\quad \mathsf{go}\ \mathsf{p}\ \mathsf{i} = \textbf{if}\ \mathsf{i} < \mathsf{card}_F\ (\mathsf{e}\ \mathsf{p})\\
&\qquad\qquad \textbf{then}\ \mathsf{e}\ \mathsf{p}\ !!_F\ \mathsf{i}\\
&\qquad\qquad \textbf{else}\ \ \mathsf{go}\ (\mathsf{p}+1)\ (\mathsf{i} - \mathsf{card}_F\ (\mathsf{e}\ \mathsf{p}))
\end{aligned}$$

This representation of enumerations always satisfies countability, but care is needed to ensure bijectivity and part-finiteness when we define the operations in Table 1.1.

The major drawback of this approach is that we cannot determine if an enumeration is finite, which means expressions such as index empty 0 fail to terminate. In our implementation (§3) we have a more sensible behaviour (an error message) when the index is out of bounds.

Bijective-function application We can map a bijective function over an enumeration.

$$\mathsf{biMap}\ \mathsf{f}\ \mathsf{e} = \mathsf{biMap}_F\ \mathsf{f} \circ \mathsf{e}$$

Part-finiteness and bijectivity are preserved by biMap (as long as it is always used only with bijective functions). The inverse of biMap f is biMap f^{-1}.

Disjoint union Disjoint union of enumerations is the pointwise union of the parts.

$$\mathsf{e}_1 \oplus \mathsf{e}_2 = \lambda \mathsf{p} \to \mathsf{e}_1\ \mathsf{p} \oplus_F \mathsf{e}_2\ \mathsf{p}$$

It is again not hard to verify that bijectivity and part-finiteness are preserved. We can also define an "unsafe" version using biMap where the user must ensure that the enumerations are disjoint:

$$\begin{aligned}
&\mathsf{union} :: E\ \mathsf{a} \to E\ \mathsf{a} \to E\ \mathsf{a}\\
&\mathsf{union}\ \mathsf{e}_1\ \mathsf{e}_2 = \mathsf{biMap}\ (\mathsf{either\ id\ id})\ (\mathsf{e}_1 \oplus \mathsf{e}_2)
\end{aligned}$$

Guarded recursion and costs Arbitrary recursion may create infinite parts. For example in the following enumeration of natural numbers:

```
data N = Z | S N deriving Show
natEnum :: E N
natEnum = union (singleton Z) (biMap S natEnum)
```

All natural numbers are placed in the same part, which breaks part-finiteness. To avoid this we place a *guard* called pay on (at least) all recursive enumerations, which pays a "cost" each time it is executed. The cost of a value in an enumeration is simply the part-number associated with the part in which it resides. Another way to put this is that pay increases the cost of all values in an enumeration:

```
pay e 0 = empty_F
pay e p = e (p − 1)
```

This definition gives fix pay ≡ empty. The cost of a value can be specified given that we know the enumeration from which it was selected.

```
cost :: E t → t → ℕ
cost (singleton _) _         ≡ 0
cost (a ⊕ b)       (Left x)  ≡ cost a x
cost (a ⊕ b)       (Right y) ≡ cost b y
cost (a ⊗ b)       (x, y)    ≡ cost a x + cost b y
cost (biMap f e)   x         ≡ cost e (f^−1 x)
cost (pay e)       x         ≡ 1 + cost e x
```

We modify natEnum by adding an application of pay around the entire body of the function:

```
natEnum = pay (union (singleton Z) (biMap S natEnum))
```

Now because we pay for each recursive call, each natural number is assigned to a separate part:

```
*Main> map values_F (map natEnum [0..3])
[[],[Z],[S Z],[S (S Z)]]
```

Cartesian product Product is slightly more complicated to define. The specification of cost allows a more formal definition of part:

Definition 2 (Part)**.** Given an enumeration e, the part for cost p (denoted as P_e^p) is the finite set of values in e such that

$$(v \in P_e^p) \Leftrightarrow (cost_e\ v \equiv p)$$

□

The specification of cost says that the cost of a product is the sum of the costs of the operands. Thus we can specify the set of values in each part of a product: $\mathsf{P}^{\mathsf{p}}_{\mathsf{a}\otimes\mathsf{b}} = \bigcup_{k=0}^{\mathsf{p}} \mathsf{P}^{\mathsf{k}}_{\mathsf{a}} \times \mathsf{P}^{\mathsf{p}-\mathsf{k}}_{\mathsf{b}}$. For our functional representation this gives the following definition:

$$
\begin{aligned}
&\mathsf{e}_1 \otimes \mathsf{e}_2 = \mathsf{pairs}\ \textbf{where}\\
&\quad \mathsf{pairs\ p} = \mathsf{concat}_F\ (\mathsf{conv}\ (\otimes_F)\ \mathsf{e}_1\ \mathsf{e}_2\ \mathsf{p})\\
&\mathsf{concat}_F :: [\mathsf{Finite\ a}] \to \mathsf{Finite\ a}\\
&\mathsf{concat}_F = \mathsf{foldl\ union}_F\ \mathsf{empty}_F\\
&\mathsf{conv} :: (\mathsf{a} \to \mathsf{b} \to \mathsf{c}) \to (\mathbb{N} \to \mathsf{a}) \to (\mathbb{N} \to \mathsf{b}) \to (\mathbb{N} \to [\mathsf{c}])\\
&\mathsf{conv}\ (\oslash)\ \mathsf{fx\ fy\ p} = [\mathsf{fx\ k} \oslash \mathsf{fy}\ (\mathsf{p} - \mathsf{k}) \mid \mathsf{k} \leftarrow [0\,..\,\mathsf{p}]]
\end{aligned}
$$

For each part we define pairs p as the set of pairs with a combined cost of p, which is the equivalent of $\mathsf{P}^{\mathsf{p}}_{\mathsf{e}_1\otimes\mathsf{e}_2}$. Because the sets of values "cheaper" than p in both e_1 and e_2 are finite, pairs p is finite for all p. For surjectivity: Any pair of values (a, b) have costs $\mathsf{ca} = \mathsf{cost}_{\mathsf{e}_1}\ \mathsf{a}$ and $\mathsf{cb} = \mathsf{cost}_{\mathsf{e}_2}\ \mathsf{b}$. This gives $(\mathsf{a}, \mathsf{b}) \in (\mathsf{e}_1\ \mathsf{ca} \otimes_F \mathsf{e}_2\ \mathsf{cb})$. This product is an element of $\mathsf{conv}\ (\otimes_F)\ \mathsf{e}_1\ \mathsf{e}_2\ (\mathsf{ca} + \mathsf{cb})$ and as such $(\mathsf{a}, \mathsf{b}) \in (\mathsf{e}_1 \otimes \mathsf{e}_2)\ (\mathsf{ca} + \mathsf{cb})$. For injectivity, it is enough to prove that pairs p1 is disjoint from pairs p2 for $\mathsf{p1} \not\equiv \mathsf{p2}$ and that (a, b) appears once in pairs $(\mathsf{ca} + \mathsf{cb})$. Both these properties follow from the bijectivity of e_1 and e_2.

3 Implementation

The implementation in the previous section is thoroughly inefficient; the complexity is exponential in the cost of the input. The cause is the computation of the cardinalities of parts. These are recomputed on each indexing (even multiple times for each indexing). In Feat we tackle this issue with *memoisation*, ensuring that the cardinality of each part is computed at most once for any enumeration.

Finite sets First we implement the Finite type as specified in the previous section. Finite is implemented directly by its consumers: A cardinality and an indexing function.

```
type Index    = Integer
data Finite a = Finite { card_F :: Index
                       , (!!_F) :: Index → a
                       }
```

Since there is no standard type for infinite precision *natural* numbers in Haskell, we use Integer for the indices. All combinators follow naturally from the correspondence to finite lists (specified in §2). Like lists, Finite is a monoid under append (i.e. union):

```
(⊕F) :: Finite a → Finite a → Finite a
f1 ⊕F f2 = Finite car ix where
  car = cardF f1 + cardF f2
  ix i = if i < cardF f1
         then f1 !!F i
         else f2 !!F (i − cardF f1)
emptyF = Finite 0 (λi → error "Empty")
instance Monoid (Finite a) where
  mempty = emptyF
  mappend = (⊕F)
```

It is also an applicative functor under product, again just like lists:

```
(⊗F) :: Finite a → Finite b → Finite (a, b)
(⊗F) f1 f2 = Finite car sel where
  car = cardF f1 ∗ cardF f2
  sel i = let (q, r) = (i `divMod` cardF f2)
          in (f1 !!F q, f2 !!F r)
singletonF :: a → Finite a
singletonF a = Finite 1 one where
  one 0 = a
  one _ = error "Index out of bounds"
instance Functor Finite where
  fmap f fin = fin { (!!F) = f ∘ (fin!!F) }
instance Applicative Finite where
  pure    = singletonF
  f ⟨∗⟩ a = fmap (uncurry ($)) (f ⊗F a)
```

For indexing we split the index $i < c_1 * c_2$ into two components by dividing either by c_1 or c_2. For an ordering which is consistent with lists (s.t. $\text{values}_F\ (f \langle * \rangle a) \equiv \text{values}_F\ f \langle * \rangle \text{values}_F\ a$) we divide by the cardinality of the second operand. Bijective map is already covered by the Functor instance, i.e. we require that the argument of fmap is a bijective function.

Enumerate As we hinted earlier, memoisation of cardinalities (i.e. of Finite values) is the key to efficient indexing. The remainder of this section is about this topic and implementing efficient versions of the operations specified in the previous section. A simple solution is to explicitly memoise the function from part numbers to part sets. Depending on where you apply such memoisation this gives different memory/speed tradeoffs (discussed later in this section).

In order to avoid having explicit memoisation we use a different approach: We replace the outer function with a list. This may seem like a regression to the list view of enumerations, but the complexity of indexing is not adversely affected since it already does a linear search on an initial segment

of the set of parts. Also the interface in the previous section can be recovered by just applying (!!) to the list. We define a data type Enumerate a for enumerations containing values of type a.

```
data Enumerate a = Enumerate { parts :: [Finite a] }
```

In the previous section we simplified by supporting only infinite enumerations. Allowing finite enumerations is practically useful and gives an algorithmic speedups for many common applications. This gives the following simple definitions of empty and singleton enumerations:

```
empty :: Enumerate a
empty = Enumerate []

singleton :: a → Enumerate a
singleton a = Enumerate [singleton_F a]
```

Now we define an indexing function with bounds-checking:

```
index :: Enumerate a → Integer → a
index = index' ∘ parts where
  index' []       i = error "index out of bounds"
  index' (f : rest) i
    | i < card_F f  = f !!_F i
    | otherwise     = index' rest (i − card_F f)
```

This type is more useful for a propery-based testing driver (see §6) because it can detect with certainty if it has tested all values of the type.

Disjoint union Our enumeration type is a monoid under disjoint union. We use the infix operator (◊) = mappend (from the library Data.Monoid) for both the Finite and the Enumerate union.

```
instance Monoid (Enumerate a) where
  mempty  = empty
  mappend = union

union :: Enumerate a → Enumerate a → Enumerate a
union a b = Enumerate $ zipPlus (◊) (parts a) (parts b)
  where
    zipPlus :: (a → a → a) → [a] → [a] → [a]
    zipPlus f  (x : xs) (y : ys) = f x y : zipPlus f xs ys
    zipPlus _ xs        ys       = xs ++ ys   -- one of them is empty
```

It is up to the user to ensure that the operands are really disjoint. If they are not then the resulting enumeration may contain repeated values. For example pure True ◊ pure True type checks and runs but it is probably not what the programmer intended. If we replace one of the Trues with False we get a perfectly reasonable enumeration of Bool.

Cartesian product and bijective functions First we define a Functor instance for Enumerate in a straightforward fashion:

```
instance Functor Enumerate where
  fmap f e = Enumerate (fmap (fmap f) (parts e))
```

An important caveat is that the function mapped over the enumeration must be *bijective* in the same sense as for biMap, otherwise the resulting enumeration may contain duplicates.

Just as Finite, Enumerate is an applicative functor under product with singleton as the lifting operation.

```
instance Applicative Enumerate where
  pure  = singleton
  f ⟨*⟩ a = fmap (uncurry ($)) (prod f a)
```

Similar to fmap, the first operand of $\langle * \rangle$ must be an enumeration of bijective functions. Typically we get such an enumeration by lifting or partially applying a constructor function, e.g. if e has type Enumerate a then f = pure (,) $\langle * \rangle$ e has type Enumerate (b → (a, b)) and f $\langle * \rangle$ e has type Enumerate (a, a).

Two things complicate the computation of the product compared to its definition in §2. One is accounting for finite enumerations, the other is defining the convolution function on lists.

A first definition of conv (that computes the set of pairs of combined cost p) might look like this (with mconcat equivalent to foldr $(\oplus_F)$ empty_F):

```
badConv :: [Finite a] → [Finite b] → Int → Finite (a, b)
badConv xs ys p = mconcat (zipWith (⊗F) (take p xs)
                                        (reverse (take p ys)))
```

The problem with this implementation is memory. Specifically it needs to retain the result of all multiplications performed by $(\otimes_F)$ which yields quadratic memory use for each product in an enumeration.

Instead we perform the multiplications each time the indexing function is executed and just retain pointers to e_1 and e_2. The problem then is the reversal. With partitions as functions it is trivial to iterate an inital segment of the partition in reverse order, but with lists it is rather inefficient and we do not want to reverse a linearly sized list every time we index into a product. To avoid this we define a function that returns all reversals of a given list. We then define a product function that takes the parts of the first operand and all reversals of the parts of the second operand.

```
reversals :: [a] → [[a]]
reversals = go [] where
  go _   []       = []
  go rev (x : xs) = let rev′ = x : rev
                    in  rev′ : go rev′ xs
```

```
prod :: Enumerate a → Enumerate b → Enumerate (a, b)
prod e1 e2 = Enumerate $ prod′ (parts e1) (reversals (parts e2))
prod′ :: [Finite a] → [[Finite b]] → [Finite (a, b)]
```

In any sensible Haskell implementation evaluating an initial segment of reversals xs uses linear memory in the length of the segment, and constructing the lists is done in linear time.

We define a version of conv where the second operand is already reversed, so it is simply a concatenation of a zipWith.

```
conv :: [Finite a] → [Finite b] → Finite (a, b)
conv xs ys = Finite card index
  where card    = sum $ zipWith (∗) (map card_F xs) (map card_F ys)
        index i = mconcat (zipWith (⊗_F) xs ys) !!_F i
```

The worst case complexity of this function is the same as for the conv that reverses the list (linear in the list length). The best case complexity is constant however, since indexing into the result of mconcat is just a linear search. It might be tempting to move the mconcat out of the indexing function and use it directly to define the result of conv. This is semantically correct but the result of the multiplications are never garbage collected. Experiments show an increase in memory usage from a few megabytes to a few hundred megabytes in a realistic application.

For specifying prod′ we can revert to dealing with only infinite enumerations i.e. assume prod′ is only applied to "padded" lists:

```
parts = let rep = repeat empty_F in Enumerate $
  prod′ (parts e1 ++ rep) (reversals (parts e2 ++ rep))
```

Then we define prod′ as:

```
prod′ xs rys = map (conv xs) rys
```

Analysing the behaviour of prod we notice that if e_2 is finite then we eventually start applying conv xs on the reversal of parts e_2 with a increasing chunk of $empty_F$ prepended. Analysing conv reveals that each such $empty_F$ corresponds to just dropping an element from the first operand (xs), since the head of the list is multiplied with $empty_F$. This suggest a strategy of computing prod′ in two stages, the second used only if e_2 is finite:

```
prod′ xs@(_ : xs′) (ys : yss) = goY ys yss where
  goY ry rys = conv xs ry : case rys of
    []          → goX ry  xs′
    (ry′ : rys′) → goY ry′ rys′
  goX ry = map (flip conv ry) ∘ tails
prod′ _ _                     = []
```

If any of the enumerations are empty the result is empty, otherwise we map over the reversals (in goY) with the twist that if the list is depleted we pass the final element (the reversal of all parts of e_2) to a new map (goX) that applies conv to this reversal and every suffix of xs. With a bit of analysis it is clear that this is semantically equivalent to the padded version (except that it produces a finite list if both operands are finite), but it is much more efficient if one or both the operands are finite. For instance the complexity of computing the cardinality at part p of a product is typically linear in p, but if one of the operands is finite it is max p l where l is the length of the part list of the finite operand (which is typically very small). The same complexity argument holds for indexing.

Assigning costs So far we are not assigning any costs to our enumerations, and we need the guarded recursion operator to complete the implementation:

```
pay :: Enumerate a → Enumerate a
pay e = Enumerate (emptyF : parts e)
```

To verify its correctness, consider that parts (pay e) !! 0 ≡ empty_F and parts (pay e) !! (p + 1) ≡ parts e !! p. In other words, applying the list indexing function on the list of parts recovers the definition of pay in the previous section (except in the case of finite enumerations where padding is needed).

Examples Having defined all the building blocks we can start defining enumerations:

```
boolE :: Enumerate Bool
boolE = pay $ pure False ◊ pure True

blistE :: Enumerate [Bool]
blistE = pay $ pure [ ]
           ◊ ((:) ⟨$⟩ boolE ⟨∗⟩ blistE)
```

A simple example shows what we have at this stage:

```
∗Main> take 16 (map cardF $ parts blistE)
[0, 1, 0, 2, 0, 4, 0, 8, 0, 16, 0, 32, 0, 64, 0, 128]
∗Main> valuesF (parts blistE !! 5)
[[False, False], [False, True], [True, False], [True, True]]
```

We can also very efficiently access values at extremely large indices:

```
∗Main> length $ index blistE (10^1000)
3321
```

```
*Main> foldl1 xor $ index blistE (10^1000)
True
*Main> foldl1 xor $ index blistE (10^1001)
False
```

Computational complexity Analysing the complexity of indexing, we see that union adds a constant factor to the indexing function of each part, and it also adds one to the generic size of all values (since it can be considered an application of Left or Right). For product we choose between p different branches where p is the cost of the indexed value, and increase the generic size by one. This gives a pessimistic worst case complexity of $p * s$ where s is the generic size. If we do not apply pay directly to the result of another pay, then $p \leqslant s$ which gives s^2. This could be improved to $s \log p$ by using a binary search in the product case, but this also increases the memory consumption (see below).

The memory usage is (as always in a lazy language) difficult to measure exactly. Roughly speaking it is the product of the number of distinguished enumerations and the highest part to which these enumerations are evaluated. This number is equal to the sum of all constructor arities of the enumerated (monomorphic) types. For regular ADTs this is a constant, for non-regular ones it is bounded by a constant multiplied with the highest evaluated part.

Sharing As mentioned, Feat relies on memoisation and subsequently sharing for efficient indexing. To demonstrate this, we move to a more realistic implementation of the list enumerator which is parameterised over the underlying enumeration.

```
listE :: Enumerate a → Enumerate [a]
listE aS = pay $ pure []
             ♢ ((:) ⟨$⟩ aS ⟨*⟩ listE aS)
blistE2 :: Enumerate [Bool]
blistE2 = listE boolE
```

This simple change causes the performance of blistE2 to drop severely compared to blistE. The reason is that every evaluation of listE aS creates a separate enumeration, even though the argument to the function has been used previously. In the original we had blistE in the tail instead, which is a top level declaration. Any clever Haskell compiler evaluates such declarations at most once throughout the execution of a program (although it is technically not required by the Haskell language report). We can remedy the problem by manually sharing the result of the computation with a **let** binding (or equivalently by using a fix point combinator):

```
listE2 :: Enumerate a → Enumerate [a]
listE2 aS = let listE = pay $ pure []
                        ◊ ((:) ⟨$⟩ aS ⟨∗⟩ listE)
            in listE
blistE3 :: Enumerate [Bool]
blistE3 = listE2 boolE
```

This is efficient again but it has one major problem, it requires the user to explicitly mark recursion. This is especially painful for mutually recursive data types since all members of a system of such types must be defined in the same scope:

```
data Tree a = Leaf a | Branch (Forest a)
newtype Forest a = Forest [Tree a]

treeE   = fst ∘ treesAndForests
forestE = snd ∘ treesAndForests
treesAndForests :: Enumerate a → (Enumerate (Tree a)
                                 , Enumerate (Forest a))
treesAndForests eA =
  let eT = pay $ (Leaf ⟨$⟩ eA) ◊ (Branch ⟨$⟩ eF)
      eF = pay $ Forest ⟨$⟩ listE2 eT
  in (eT, eF)
```

Also there is still no sharing between different evaluations of treeS and forestS in other parts of the program. This forces everything into the same scope and crushes modularity. What we really want is a class of enumerable types with a single overloaded enumeration function.

```
class Enumerable a where
  enumerate :: Enumerate a

instance Enumerable Bool where
  enumerate = boolE

instance Enumerable a ⇒ Enumerable (Tree a) where
  enumerate = pay ((Leaf ⟨$⟩ enumerate) ◊ (Branch ⟨$⟩ enumerate))

instance Enumerable a ⇒ Enumerable [a] where
  enumerate = listE2 enumerate

instance Enumerable a ⇒ Enumerable (Forest a) where
  enumerate = pay (Forest ⟨$⟩ enumerate)
```

This solution performs well and it is modular. The only potential problem is that there is no guarantee of enumerate being evaluated at most once for each monomorphic type. We write potential problem because it is difficult to determine if this is a problem in practice. It is possible to provoke GHC into reevaluating instance members, and even if GHC mostly does what we want other compilers might not. In the next section we discuss a solution that guarantees sharing of instance members.

4 Instance sharing

Our implementation relies on memoisation for efficient calculation of cardinalities. This in turn relies on sharing; specifically we want to share the instance methods of a type class. For instance we may have:

```
instance Enumerable a ⇒ Enumerable [a] where
  enumerate = pay $ pure []
                    ♢ ((:) ⟨$⟩ enumerate ⟨∗⟩ enumerate)
```

The typical way of implementing Haskell type classes is using dictionaries, and this essentially translates the instance above into a function similar to enumerableList :: Enumerate a → Enumerate [a]. Determining exactly when GHC or other compilers recompute the result of this function requires significant insight into the workings of the compiler and its runtime system. Suffice it to say that when re-evaluation does occur it has a significant negative impact on the performance of Feat. In this section we present a practical solution to this problem.

A monad for type-based sharing The general formulation of this problem is that we have a value x :: C a ⇒ f a, and for each monomorphic type T we want x :: f T to be shared, i.e. to be evaluated at most once. The most direct solution to this problem seems to be a map from types to values i.e. Bool is mapped to x :: f Bool and () to x :: f (). The map can then either be threaded through a computation using a state monad and updated as new types are discovered or updated with unsafe IO operations (with careful consideration of safety). We have chosen the former approach here.

The map must be dynamic, i.e. capable of storing values of different types (but we still want a type safe interface). We also need representations of Haskell types that can be used as keys. Both these features are provided by the Typeable class.

We define a data structure we call a dynamic map as an (abstract) data type providing type safe insertion and lookup. The type signatures of dynInsert and dynLookup are the significant part of the code, but the full implementation is provided for completeness.

```
import Data.Dynamic (Dynamic, fromDynamic, toDyn)
import Data.Typeable (Typeable, TypeRep, typeOf)
import Data.Map as M

newtype DynMap = DynMap (M.Map TypeRep Dynamic)
  deriving Show

dynEmpty :: DynMap
dynEmpty = DynMap M.empty
```

```
dynInsert :: Typeable a ⇒ a → DynMap → DynMap
dynInsert a (DynMap m) =
  DynMap (M.insert (typeOf a) (toDyn a) m)
```

To associate a value with a type we just map its type representation to the dynamic (type casted) value.

```
dynLookup :: Typeable a ⇒ DynMap → Maybe a
dynLookup (DynMap m) = hlp run ⊥ where
  hlp :: Typeable a ⇒ (TypeRep → Maybe a) → a → Maybe a
  hlp f a = f (typeOf a)
  run tr  = M.lookup tr m >>= fromDynamic
```

Lookup is also easily defined. The dynamic library provides a function fromDynamic :: Dynamic → Maybe a. In our case the M.lookup function has already matched the type representation against a type stored in the map, so fromDynamic is guaranteed to succeed (as long as values are only added using the insert function).

Using this map type we define a sharing monad with a function share that binds a value to its type.

```
type Sharing a = State DynMap a

runSharing :: Sharing a → a
runSharing m = evalState m dynEmpty

share :: Typeable a ⇒ Sharing a → Sharing a
share m = do
  mx ← gets dynLookup
  case mx of
    Just e    → return e
    Nothing → mfix $ λe → do
      modify (dynInsert e)
      m
```

Note that we require a monadic fixpoint combinator to ensure that recursive computations are shared. If it had not been used (i.e. if the Nothing case had been m >>= modify ∘ dynInsert) then any recursively defined m would eventually evaluate share m and enter the Nothing case. Using the fix point combinator ensures that a reference to the result of m is added to the map *before* m is computed. This makes any evaluations of share m inside m end up in the Just case which creates a cyclic reference in the value (exactly what we want for a recursive m). For example in x = share (liftM pay x) the fixpoint combinator ensures that we get runSharing x ≡ fix pay instead of ⊥.

Self-optimising enumerations Now we have a monad for sharing and one way to proceed is to replace Enumerate a with Sharing (Enumerate a)

and re-implement all the combinators for that type. We don't want to lose the simplicity of our current type though and it seems a very high price to pay for guaranteeing sharing which we are used to getting for free.

Our solution extends the enumeration type with a self-optimising routine, i.e. all enumerations have the same functionality as before but with the addition of an optimiser record field:

```
data Enumerate a = Enumerate
  { parts     :: [Finite a]
  , optimiser :: Sharing (Enumerate a)
  } deriving Typeable
```

The combinator for binding a type to an enumeration is called eShare.

```
eShare :: Typeable a ⇒ Enumerate a → Enumerate a
eShare e = e { optimiser = share (optimiser e) }
```

We can resolve the sharing using optimise.

```
optimise :: Enumerate a → Enumerate a
optimise e = let e' = runSharing (optimiser e) in
  e' { optimiser = return e' }
```

If eShare is used correctly, optimise is semantically equivalent to id but possibly with a higher degree of sharing. But using eShare directly is potentially harmful. It is possible to create "optimised" enumerations that differ semantically from the original. For instance λe $\rightarrow$ eShare t e yields the same enumerator when applied to two different enumerators of the same type. As a general rule the enumeration passed to eShare should be a closed expression to avoid such problems. Luckily users of Feat never have to use eShare, instead we provide a safe interface that uses it internally.

An implication of the semantic changes that eShare may introduce is the possibility to replace the Enumerable instances for any type throughout another enumerator by simply inserting a value in the dynamic map before computing the optimised version. This could give unintuitive results if such enumerations are later combined with other enumerations. In our library we provide a simplified version of this feature where instances can be replaced but the resulting enumeration is optimised, which makes the replacement completely local and guarantees that optimise still preserves the semantics.

The next step is to implement sharing in all the combinators. This is simply a matter of lifting the operation to the optimised enumeration. Here are some examples where ... is the original definitions of parts.

```
fmap f e = e { ...
  optimiser = fmap (fmap f) $ optimiser e }
```

```
f ⟨∗⟩ a = Enumerate {...
  optimiser = liftM2 (⟨∗⟩) (optimiser f) (optimiser a)}
pure a = Enumerate {...
  optimiser = return (pure a)}
```

The only noticeable cost of using eShare is the reliance on Typeable. Since almost every instance should use eShare and consequently require type parameters to be Typeable and since Typeable can be derived by GHC, we chose to have it as a superclass and implement a default sharing mechanism with eShare.

```
class Typeable a ⇒ Enumerable a where
  enumerate :: Enumerate a
shared :: Enumerable a ⇒ Enumerate a
shared = eShare enumerate
optimal :: Enumerable a ⇒ Enumerate a
optimal = optimise shared
```

The idiom is that enumerate is used to define instances and shared is used to combine them. Finally optimal is used by libraries to access the contents of the enumeration (see §6).

Non-regular enumerations The sharing monad works very well for enumerations of regular types, where there is a closed system of shared enumerations. For non-regular enumerations (where the number of enumerations is unbounded) the monadic computation may fail to terminate. In these (rare) cases the programmer must ensure termination.

Free pairs and boilerplate instances There are several ways to increase the sharing further, thus reducing memory consumption. Particularly we want to share the cardinality computation of every sequenced application ($\langle * \rangle$). To do this we introduce the FreePair data type which is just like a pair except constructing one carries no cost i.e. the cost of the pair is equal to the total costs of its components.

```
data FreePair a b = FreePair a b deriving (Show, Typeable)
instance (Enumerable a, Enumerable b) ⇒ Enumerable (FreePair a b)
  where enumerate = FreePair ⟨$⟩ shared ⟨∗⟩ shared
```

Since the size of FreePair a b is equal to the sum of the sizes of a and b, we know that for these functions:

```
f :: a → b → c
g :: FreePair a b → c
g (FreePair a b) = f a b
```

We have $(\mathsf{f}\ \langle\$\rangle\ \mathsf{shared}\ \langle*\rangle\ \mathsf{shared})$ isomorphic to $(\mathsf{g}\ \langle\$\rangle\ \mathsf{shared})$ but in the latter case the product of the enumerations for a and b are always shared with other enumerations that require it (because $\mathsf{shared} :: \mathsf{FreePair\ a\ b}$ is always shared. In other words *deep uncurrying* functions before applying them to shared often improve the performance of the resulting enumeration. For this purpose we define a function which is equivalent to uncurry from the Prelude but that operates on FreePair.

```
funcurry :: (a → b → c) → FreePair a b → c
funcurry f (FreePair a b) = f a b
```

Now in order to make an enumeration for a data constructor we need one more function:

```
unary :: Enumerable a ⇒ (a → b) → Enumerate b
unary f = f ⟨$⟩ shared
```

Together with pure for nullary constructors, unary and funcurry can be used to map any data constructor to an enumeration. For instance pure [] and unary (funcurry (:)) are enumerations for the constructors of [a]. In order to build a new instance we still need to combine the enumerations for all constructors and pay a suitable cost. Since pay is distributive over $\Diamond$, we can pay once for the whole type:

```
consts :: [Enumerate a] → Enumerate a
consts xs = pay $ foldl (◇) mempty xs
```

This gives the following instance for lists:

```
instance Enumerable a ⇒ Enumerable [a] where
  enumerate = consts [pure [], unary (funcurry (:))]
```

5 Invariants

Data type invariants are a major challenge in property based testing. An invariant is just a property on a data type, and one often wants to test that it holds for the result of a function. But we also want to test other properties only with input that is known to satisfy the invariant. In random testing this can sometimes be achieved by filtering: discarding the test cases that do not satisfy the invariant and generating new ones instead, but if the invariant is an arbitrary Boolean predicate finding test data that satisfies the invariant can be as difficult as finding a bug. For systematic testing (with SmallCheck or Feat) this method is slightly more feasible since we do not repeat values which guarantees progress, but filtering is still a brute force solution.

With QuickCheck programmers can manually define custom test data generators that guarantee any invariant, but it may require a significant programmer effort and analysing the resulting generator to ensure correctness and statistical coverage can be difficult. Introducing this kind of complexity into testing code is hazardous since complex usually means error prone.

In Feat the room for customised generators is smaller (corresponding to the difference between monads and applicative functors). In theory it is possible to express any invariant by providing a bijection from a Haskell data type to the set of values that satisfy the invariant (since functional enumerations are closed under bijective function application). In practice the performance of the bijection needs to be considered because it directly affects the performance of indexing.

A simple and very common example of an invariant is the non-empty list. The function uncurry $(:)$ is a bijection into non-empty lists of a from the type $(a, [a])$. The preferred way of dealing with these invariants in Feat is by defining a **newtype** for each restricted type, and a *smart constructor* which is the previously mentioned bijection and export it instead of the data constructor.

```
newtype NonEmpty a = MkNonEmpty {nonEmpty :: [a]}
  deriving Typeable

mkNonEmpty :: a → [a] → NonEmpty a
mkNonEmpty x xs = MkNonEmpty (x : xs)

instance Enumerable a ⇒ Enumerable (NonEmpty a) where
  enumerate = consts [unary (funcurry mkNonEmpty)]
```

To use this in an instance declaration, we only need the nonEmpty record function. In this example we look at the instance for the data type Type from the Template Haskell abstract syntax tree which describes the syntax of (extended) Haskell types. Consider the constructor for universal quantification:

```
ForallT :: [TyVarBndr] → Cxt → Type → Type
```

This constructor must not be applied to the empty list. We use nonEmpty to ensure this:

```
instance Enumerable Type where
  enumerate = consts [...
                      , funcurry $ funcurry $ ForallT ∘ nonEmpty]
```

Here ForallT ∘ nonEmpty has type:

```
NonEmpty TyVarBndr → Cxt → Type → Type
```

The only change from the unrestricted enumeration is post-composition with nonEmpty.

Enumerating Sets of natural numbers Another fairly common invariant is sorted lists of unique elements i.e. Sets. It is not obvious that sets can be built from our basic combinators. We can however define a bijection from lists of natural numbers to sets of natural numbers: scanl $(((+) \circ (1+))$. For example the list $[0,0,0]$ represents the set $[0,1,2]$, the list $[1,1,0]$ represents $[1,3,4]$ and so on. We can define an enumerator for natural numbers using a bijection from Integer.

```
newtype Nat = Nat {nat :: Integer}
  deriving (Show, Typeable, Eq, Ord)
mkNat :: Integer → Nat
mkNat a = Nat $ abs $ a * 2 − if a > 0 then 1 else 0
instance Enumerable Nat where
  enumerate = unary mkNat
```

Then we define sets of naturals:

```
newtype NatSet = MkNatSet {natSet :: [Integer]}
  deriving Typeable
mkNatSet :: [Nat] → NatSet
mkNatSet = MkNatSet ∘ scanl1 ((+) ∘ (1+)) ∘ map nat
```

Generalising to sets of arbitrary types Sets of naturals are useful but what we really want is a data type Set a = MkSet {set :: [a]} and a bijection to this type from something which we can already enumerate. Since we just defined an enumeration for sets of naturals, an efficient bijective mapping from natural numbers to a is all we need. Since this is the definition of a functional enumeration, we appear to be in luck.

```
mkSet :: Enumerate a → NatSet → Set a
mkSet e = MkSet ∘ map (index e) ∘ natSet
instance Enumerable a ⇒ Enumerable (Set a) where
  enumerate = unary (mkSet optimal)
```

This implementation works but it is slightly simplified, it doesn't use the cardinalities of a when determining the indices to use. This distorts the cost of our sets away from the actual size of the values.

6 Accessing enumerated values

This section discusses strategies for accessing the values of enumerations, especially for the purpose of property based testing. The simplest function values is simply all values in the enumeration partitioned by size. We include the cardinalities as well because this is often useful e.g. to report to

the user how many values are in a part before initiating testing on values. For this reason we give values type Enumerate a $\rightarrow [(\text{Integer}, [\text{a}])]$.

Given that Feat is intended to be used primarily with the Enumerable type class, we have implemented the library functions to use class members, but provide non-class versions of the functions that have the suffix With:

```
type EnumL a = [(Integer, [a])]
values :: Enumerable a ⇒ [(Integer, [a])]
values = valuesWith optimal
valuesWith :: Enumerate a → [(Integer, [a])]
valuesWith = map (λf → (card_F f, values_F f)) ∘ parts
```

Parallel enumeration A generalisation of values is possible since we can "skip" an arbitrary number of steps into the enumeration at any point. The function striped takes a starting index and a step size n and enumerates every n^{th} value after the initial index in the ordering. As a special case values = striped 0 0 1. One purpose of this function is to enumerate in parallel. If n processes execute uncurry striped k n where k is a process-unique id in the range $[0\,..\,\text{n}-1]$ then all values are eventually evaluated by some process and, even though the processes are not communicating, the work is evenly distributed in terms of number and size of test cases.

```
stripedWith :: Enumerate a → Index → Integer → EnumL a
stripedWith e o₀ step = stripedWith′ (parts e) o₀ where
  stripedWith′ (Finite crd ix : ps) o =
      (max 0 d, thisP) : stripedWith′ ps o′
    where
      o′     = if space ⩽ 0 then o − crd else step − m − 1
      thisP  = map ix (genericTake d $ iterate (+step) o)
      space  = crd − o
      (d, m) = divMod space step
```

Bounded enumeration Another feature afforded by random-access indexing is the ability to systematically select manageable portions of gigantic parts. Specifically we can devise a function bounded :: Integer → EnumL a such that each list in bounded n contains at most n elements. If there are more than n elements in a part we systematically sample n values that are evenly spaced across the part.

```
samplePart :: Integer → Finite a → (Integer, [a])
samplePart m (Finite crd ix) =
  let step   = crd % m
  in if crd ⩽ m
```

```
      then (crd, map ix [0 .. crd − 1])
      else (m,  map ix [ round (k ∗ step)
                       | k ← map toRational [0 .. m − 1]])
boundedWith :: Enumerate a → Integer → EnumL a
boundedWith e n = map (samplePart n) $ parts e
```

Random sampling A noticeable feature of Feat is that it provides random sampling with uniform distribution over a size-bounded subset of a type. This is not just nice for compatibility with QuickCheck, it is genuinely difficult to write a uniform generator even for simple recursive types with the tools provided by the QuickCheck library.

The function uniform :: Enumerable a ⇒ Part → Gen a generates values of the given size or smaller.

```
uniformWith :: Enumerate a → Int → Gen a
uniformWith = uni ∘ parts where
  uni :: [Finite a] → Int → Gen a
  uni [] _      = error "uniform: Empty enumeration"
  uni ps maxp = let (incl, rest) = splitAt maxp ps
                    fin          = mconcat incl
                in case card_F fin of
                     0 → uni rest 1
                     _ → do i ← choose (0, card_F fin − 1)
                            return (fin !!_F i)
```

Since we do not make any local random choices, performance is favourable compared to hand written generators. The typical usage is sized uniform, which generates values bounded by the QuickCheck size parameter. In Table 1.2 we present a typical output of applying the function sample from the QuickCheck library to the uniform generator for [[Bool]]. The function drafts values from the generator using increasing sizes from 0 to 20.

7 Case study: Enumerating the ASTs of Haskell

As a case study, we use the enumeration technique developed in this paper to generate values of Haskell ASTs, specifically the abstract syntax of Template Haskell, taken from the module Language.Haskell.TH.Syntax.

We use the generated ASTs to test the Template Haskell pretty printer. The background is that in working with *BNFC-meta* (Duregård and Jansson, 2011), which relies heavily on meta programming, we noticed that the TH pretty printer occasionally produced un-parseable output. BNFC-meta also relies on the more experimental package *haskell-src-meta* that

```
*Main> sample (sized uniform :: Gen [[Bool]])
[]
[[]]
[[],[]]
[[True]]
[[False],[],[]]
[[],[False,False,True]]
[[False,True,False,True,True]]
[[False],[],[],[]]
[[True],[True],[],[False,True]]
[[False],[False,True,False,False,True]]
```

Table 1.2: Randomly chosen values from the enumeration of [[Bool]]

```
data Exp    = VarE Name | CaseE Exp [Match] | ...   -- 18 Cons.
data Match  = Match Pat Body [Dec]
data Body   = GuardedB [(Guard, Exp)] | NormalB Exp
data Dec    = FunD Name [Clause] | ...              -- 14 Cons.
data Clause = Clause [Pat] Body [Dec]
data Pat    = LitP Lit | ViewP Exp Pat | ...        -- 14 Cons.
```

Table 1.3: Parts of the Template Haskell AST type. Note that all the types are mutually recursive. The comments indicate how many constructors there are in total of that type

forms a bridge between the *haskell-src-exts* parser and Template Haskell. We wanted to test this tool chain on a system-level.

The AST types We limited ourselves to testing expressions, but following dependencies and adding a few **newtype** wrappers this yielded a system of almost 30 data types with 80+ constructors. A small part is shown in Table 1.3.

We excluded a few non-standard extensions (e.g. bang patterns) because the specification for these are not as clear (especially the interactions between different Haskell extensions).

Comparison to existing test frameworks We wanted to compare Feat to existing test frameworks. For a set of mutual-recursive data types of this size, it is very difficult to write a sensible QuickCheck generator. We therefore excluded QuickCheck from the case study.

On the other hand, generators for SmallCheck and Feat are largely boilerplate code. To avoid having the results skewed by trying to generate the large set of strings for names (and to avoid using GHC-internal names which are not printable), we fix the name space and regard any name as having size 1. But we do generate characters and strings as literals (and found bugs in these).

Test case distribution The result shows some interesting differences between Feat and SmallCheck on the distribution of the generated values. We count the number of values of each part (depth for SmallCheck and size for Feat) of each generator.

Size	1	2	3	4	5	6	...	20
SmallCheck	1	9	951	×	×	×	...	×
Feat	0	1	5	11	20	49	...	65072965

Table 1.4: The number of test cases below a certain size

It is clear that for big data types such as ASTs, SmallCheck quickly hits a wall: The number of values below a fixed size grows aggressively, and we are not able to complete the enumeration of size 4 (given several hours of execution time). In the case of Feat, the growth in the number of values in each category is more controlled, due to its more refined definition of size.

We looked more closely into the values generated by SmallCheck by sampling the first 10000 values of the series on depth 4. A count revealed that the maximum size in this sample is 35, with more than 50% of the values having a size more than 20. Thus, contrary to the goal of generating small values, SmallCheck is actually generating pretty large values from early on.

Testing the TH pretty printer The generated AST values are used as test cases to find bugs in Template Haskell's pretty printer (Language.Haskell.TH.Ppr). We start with a simple property: A printed expression should be syntactically valid Haskell. We use *haskell-src-exts* as a test oracle:

```
prop_parses e =
  case parse $ pprint (e :: Exp) :: ParseResult Exp of
    ParseOk _       → True
    ParseFailed _ s → False
```

After a quick run, Feat reports numerous bugs, some of which are no doubt false positives. A small example of a confirmed bug is the expression `[Con..]`. The correct syntax has a space after the constructor name (i.e. `[Con ..]`). As we can see, this counterexample is rather small (having size 6 and depth 4). However, after hours of testing SmallCheck is not able to

find this bug even though many much larger (but not deeper) values are tested. Given a very large search space that is not exhaustible, SmallCheck tends to get stuck in a corner of the space and test large but similar values. The primary cause of SmallCheck's inability to deal with ASTs is that the definition of "small" as "shallowly nested" means that there are very many small values but many types can practically not be reached at all. For instance generating any `Exp` with a where-clause seems to require at least depth 8, which is far out of reach.

Comparatively, the behaviour of Feat is much better. It advances quickly to cover a wider range of small values, which maximises the chance of finding a bug. The guarantee "correct for all inputs with 15 constructors or less" is much stronger than "correct for all values of at most depth 3 and a few million of depth 4". When there is no bug reported, Feat reports a more meaningful portion of the search space that has been tested.

It is worth mentioning that SmallCheck has the facility of performing "depth-adjustment", that allows manual increment of the depth count of individual constructors to reduce the number of values in each category. For example, instead counting all constructors as 1, one may choose to count a binary constructor as having depth 2 to reflect the fact that it may create a larger value than a unary one (similar to our `pay` function). In our opinion, this adjustment is a step towards an imprecise approximation of size as used in our approach. Even if we put time into manually adjusting the depth it is unclear what kind of guarantee testing up to depth 8 implies, especially when the definition of depth has been altered away from generic depth.

Testing round trip properties We also tested an extension of this property that does not only test the syntactic correctness but also that the information in the AST is preserved when pretty printing. We tested this by making a round trip function that pretty prints the AST, parses it with *haskell-src-exts* and converts it back to Template Haskell AST with *haskell-src-meta*. This way we could test this tool chain on a system level, finding bugs in *haskell-src-meta* as well as the pretty printer. The minimal example of a pretty printer error found was `StringL "\n"` which is pretty printed to `""`, discarding the newline character. This error was not found by SmallCheck partly because it is too deep (at least depth 4 depending on the character generator), and partly because the default character generator of SmallCheck only tests alphabetical characters. Presumably an experienced SmallCheck tester would use a **newtype** to generate more sensible string literals.

Refuting the small scope hypothesis SmallCheck is based on the *small scope hypothesis* which states that it is sufficient to exhaustively test a small

part of the input set to find most bugs. SmallCheck in particular makes the assumption that it is sufficient to test all values bounded by some depth. As we have shown, this assumption does not hold for testing the Template Haskell pretty printer and other properties that quantify over ASTs: Although there where several bugs, none where found within feasible range of a depth-bounded search.

Although Feat is not limited to exhaustive search, we found that using Feat to implement size-bounded search is sufficient to find bugs in the Template Haskell example. In other words we did not have to rely on the random access our enumerators provide, the difference between partitioning by depth and size seemed sufficient to find bugs. This raises the question of whether the small scope hypothesis is valid for the scope of size-bounded values, and ultimately if there is any practical value in the ability to select larger values by random or systematic sampling.

To test this we made an additional experiment where we disabled individual constructors from being generated until we where not able to find any errors in the first few million values of our exhaustive search. This is an abbreviated output from our test run:

```
* Testing 0 values at size 0
* Testing 0 values at size 1
* Testing 1 values at size 2
...
* Testing 984968 values at size 16
```

In less than a minute we where able to exhaustively search to size 16 without finding any new bugs. We then tested a systematic sampler that selected at most 10000 values of each size up to size 100 and saw if it found any additional errors.

```
* Testing 0 values at size 0
...
* Testing 4583 values at size 11
* Testing 10000 values at size 12
...
* Testing 10000 values at size 24
Failure!
Conrete Syntax: \'\NUL' -> let var :: [forall var . []] in []
Abstract Syntax: LamE [LitP (CharL '\NUL')] (LetE [SigD var
  (AppT ListT (ForallT [PlainTV var] [] ListT))] (ListE []))
```

This appears to be an error in our reference parser, which expects brackets around the forall type even when it is inside a list constructor. Regardless of where the error lies, it is a counterexample that is not found by exhaustive testing but which is found using systematic sampling.

This method does not guarantee a minimal counterexample and indeed it is possible to remove parts of the example above and still get the same error. The smallest size of a program exhibiting this bug turned out to be 17. We where able to find this bug using exhaustive search by letting it run for a few more minutes. For this reason we are reluctant to consider the outcome as a proper refutation of the small scope hypothesis and we see this part of our experiment as inconclusive.

8 Related Work

SmallCheck, Lazy SmallCheck and QuickCheck Our work is heavily influenced by the property based testing frameworks QuickCheck (Claessen and Hughes, 2000) and SmallCheck (Runciman, Naylor, and Lindblad, 2008). The similarity is greatest with SmallCheck and we improve upon it in two distinct ways:

- (Almost) Random access times to enumerated values. This presents a number of possibilities that are not present in SmallCheck, including random or systematic sampling of large values (too large to exhaustively enumerate) and overhead-free parallelism.

- A definition of size which is closer to the actual size. Especially for testing abstract syntax tree types and other "wide" types this seems to be a very important feature (see §7).

Since our library provides random generation as an alternative or complement to exhaustive enumeration it can be considered a "best of two worlds" link between SmallCheck and QuickCheck. We provide a generator which should ease the reuse of existing QuickCheck properties.

SmallCheck systematically tests by enumerating all values bounded by depth of constructor nestings. In a sense this is also a partitioning by size. The major problem with SmallCheck is that the number of values in each partition grows too quickly, often hitting a wall after a few levels of depth. For AST's this is doubly true; the growth is proportional to the number of constructors in the type, and it is unlikely you can ever test beyond depth 4 or so. This means that most constructors in an AST are never touched.

Lazy SmallCheck can cut the number of tests on each depth level by using the inherent laziness of Haskell. It can detect if a part of the tested value is evaluated by the property and if it is not it refrains from refining this value further. In some cases this can lead to an exponential decrease of the number of required test cases. In the case of testing a pretty printer (as we do in §7) Lazy SmallCheck would offer no advantage since the property fully evaluates its argument every time.

After the submission of this paper, a package named *gencheck* was uploaded to Hackage (Uszkay and Carette, 2012). GenCheck is designed to generalise both QuickCheck and SmallCheck, which is similar to Feat in goal. This initial release has very limited documentation, which prevents a more comprehensive comparison at the moment.

EasyCheck In the functional logic programming language Curry (Hanus et al., 2006), one form of enumeration of values comes for free in the form of a search tree. As a result, testing tools such as EasyCheck (Christiansen and Fischer, 2008) only need to focus on the traversal strategy for test case generation. It is argued in Christiansen and Fischer, 2008 that this separation of the enumeration scheme and the test case generation algorithm is particularly beneficial in supporting flexible testing strategies.

Feat's functional enumeration, with its ability to exhaustively enumerate finite values, and to randomly sample very large values, lays an excellent groundwork for supporting various test case generation algorithms. One can easily select test cases of different sizes with a desired distribution.

AGATA AGATA (Duregård, 2009) is previous work by Jonas Duregård. Although it is based entirely on random testing it is a predecessor of Feat in the sense that it attempts to solve the problem of testing syntactic properties of abstract syntax trees. It is our opinion that Feat subsumes AGATA in this and every other aspect.

Generating (Typed) Lambda Terms To test more aspects of a compiler other than the libraries that perform syntax manipulation, it is more desirable to generate terms that are type correct.

In Yakushev and Jeuring, 2009, well-typed terms are enumerated according to their costs—a concept similar to our notion of size. Similar to SmallCheck, the enumeration in Yakushev and Jeuring, 2009 adopts the list view, which prohibits the sampling of large values. On the other hand, the special-purpose QuickCheck generator designed in Pałka et al., 2011, randomly generates well-typed terms. Unsurprisingly, it has no problem with constructing individual large terms, but falls short in systematicness.

It is shown (Wang, 2005) that well-scoped (but not necessarily well-typed) lambda terms can be uniformly generated. The technique used in Wang, 2005 is very similar to ours, in the sense that the number of possible terms for each syntactic constructs are counted (with memoization) to guide the random generation for a uniform distribution. This work can be seen as a special case of Feat, and Feat can indeed be straightforwardly instrumented to generate well-scoped lambda terms.

Feat is at present not able to express complicated invariants such as type correctness of the enumerated terms. One potential solution is to adopt

more advanced type systems as in Yakushev and Jeuring, 2009, so that the type of the enumeration captures more precisely its intended range.

Combinatorial species In mathematics a *combinatorial species* is an endofunctor on the category of finite sets and bijections. Each object A in this category can be described by its cardinality n and a finite enumeration of its elements: $f : \mathbb{N}_n \to A$. In other words, for each n there is a canoncial object (label set) $\mathbb{N}_n$. Each arrow $phi : A \to B$ in this category is between objects of the same cardinality n, and can be described by a permutation of the set $\mathbb{N}_n$. This means that the object action S_0 of an endofunctor S maps a pair (n, f) to a pair $S_0\ (n, f)$ whose first component is the cardinality of the resulting set (we call it card n). (The arrow action S_1 maps permutations on $\mathbb{N}_n$ to permutations on $\mathbb{N}_{\mathsf{card}\ n}$.)

In the species Haskell library (decribed by Yorgey, (2010)) there is a function $\mathsf{enumerate} : \mathsf{Enumerable}\ f \Rightarrow [a] \to [f\ a]$ which takes a (list representation of) an object a to all $(f\ a)$-structures obtained by the S_0 map. The key to comparing this with our paper is to represent the objects as finite enumerations $\mathbb{N}_n \to a$ instead of as lists $[a]$. Then $\mathsf{enumerate}' : \mathsf{Enumerable}\ f \Rightarrow (\mathbb{N}_n \to a) \to (\mathbb{N}_{\mathsf{card}\ n} \to f\ a)$. We can further let a be $\mathbb{N}_p$ and define $\mathsf{sel}\ p = \mathsf{enumerate}'\ \mathsf{id} : \mathbb{N}_{\mathsf{card}\ p} \to f\ \mathbb{N}_p$. The function sel is basically an inefficient version of the indexing function in the Feat library. The elements in the image of g for a particular n are (defined to be) those of weight n. The union of all those images form a set (a type). Thus a species is roughly a partition of a set into subsets of elements of the same size.

The theory of species goes further than what we present in this paper, and the species library implements quite a bit of that theory. We cannot (yet) handle non-regular species, but for the regular ones we can implement the enumeration efficiently.

Boltzmann samplers A combinatorial class is basically the same as what we call a "functional enumeration": A set C of combinatorial objects with a size function such that all the parts C_n of the induced partitioning are finite. A *Boltzmann model* is a probability distribution (parameterized over a small real number x) over such a class C, such that a uniform discrete probability distribution is used within each part C_n. A *Boltzmann sampler* is (in our terminology) a random generator of values in the class C following the Boltzmann model distribution. The data type generic Bolztmann sampler defined in Duchon et al., 2004 follows the same structure as our generic enumerator. We believe a closer study of that paper could help defining random generators for ASTs in a principled way from our enumerators.

Decomposable combinatorial structures. The research field of enumerative combinatorics has worked on what we call "functional enumeration"

already in the early 1990:s and Flajolet and Salvy, (1995) provide a short overview and a good entry point. They define a grammar for "decomposable" combinatorial structures including constructions for (disjoint) union, product, sequence, sets and cycles (atoms or symbols are the implicit base case). The theory (and implementation) is based on representing the counting sequences $\{C_i\}$ as generating functions as there is a close correspondance between the grammar constructs and algebraic operations on the generating functions. For decomposable structures they compute generating function *equations* and by embedding this in a computer algebra system (Maple) the equations can be symbolically manipulated and sometimes solved to obtain closed forms for the GFs. What they don't do is consider the pragmatic solution of just tabulating the counts instead (as we do). They also don't consider complex algebraic data types, just universal (untyped) representations of them. Complex ASTs can perhaps be expressed (or simulated) but rather awkwardly. They also don't seem to implement the index function into the enumeration (only random generation). Nevertheless, their development is impressive, both as a mathematical theory and as a computer library and we want to explore the connection further in future work.

9 Conclusions and Future work

Since there are now a few different approaches to property based testing available for Haskell it would be useful with a library of properties to compare the efficiency of the libraries at finding bugs. The library could contain "tailored" properties that are constructed to exploit weaknesses or utilise strengths of known approaches, but it would be interesting to have naturally occurring bugs as well (preferably from production code). It could also be used to evaluate the paradigm of property based testing as a whole.

Instance (dictionary) sharing Our solution to instance sharing is not perfect. It divides the interface into separate class functions for consuming and combining enumerations and it requires `Typeable`.

A solution based on stable names (Peyton Jones, Marlow, and Elliot, 1999) would remove the `Typeable` constraint but it is not obvious that there is any stable name to hold on to (the stable point is actually the dictionary function, but that is off-limits to the programmer). Compiler support is always a possible solution (i.e. by a flag or a pragma), but should only be considered as a last resort.

Enumerating functions For completeness, Feat should support enumerating function values. We argue that in practice this is seldom useful for

property based testing because non trivial higher order functions often have some requirement on their function arguments, for instance the $*$By functions in Data.List need functions that are total orderings, a parallel fold needs an associative function etc. This cannot be checked as a precondition, thus the best bet is probably to supply a few manually written total orderings or possibly use a very clever QuickCheck generator.

Regardless of this, it stands to reason that *functional* enumerations should have support for functions. This is largely a question of finding a suitable definition of size for functions, or an efficient bijection from an algebraic type into the function type.

Invariants The primary reason why enumeration cannot completely replace the less systematic approach of QuickCheck testing is invariants. QuickCheck can always be used to write a generator that satisfies an invariant, but often with no guarantees on the distribution or coverage of the generator.

The general understanding seems to be that it is not possible to use systematic testing and filtering to test functions that require e.g. type correct programs. Thus QuickCheck gives you something, while automatic enumeration gives you nothing. The reason is that the ratio type correct/syntactically correct programs is so small that finding valid non-trivial test cases is too time consuming.

It would be worthwhile to try and falsify or confirm the general understanding for instance by attempting to repeat the results of (Pałka et al., 2011) using systematic enumeration.

Invariants and costs We have seen that any bijective function can be mapped over an enumeration, preserving the enumeration criterion. This also preserves the cost of values, in the sense that a value `x` in the enumeration `fmap f e` costs as much as $\texttt{f}^{-1}\texttt{x}$.

This might not be the intention, particularly this means that a strong size guarantee (i.e. that the cost is equal to the number of constructors) is typically not preserved. As we show in §7 the definition of size can be essential in practice and the correlation between cost and the actual number of constructors in the value should be preserved as far as possible. There may be useful operations for manipulating costs of enumerations.

Conclusions We present an algebra of enumerations, an efficient implementation and show that it can handle large groups of mutually recursive data types. We see this as a step on the way to a unified theory of test data enumeration and generation. Feat is available as an open source package from the HackageDB repository:
`http://hackage.haskell.org/package/testing-feat`

Paper II

NEAT: Non-strict Enumeration of Algebraic Types

Undergoing review for publication in the Journal of Functional Programming

NEAT: Non-strict Enumeration of Algebraic Types

Jonas Duregård

Abstract

This paper describes NEAT, an algorithm for lazy search in algebraic data types, and an efficient Haskell implementation. For a data type D and a predicate function $p :: D \to Bool$, NEAT searches for values in D for which the predicate is true. This is useful in property-based testing, either to search directly for counterexamples to properties, or search for test data that satisfies a precondition.

The predicate p is treated as a black box, the only operation used is applying it to a value and get true or false. We also observe the *laziness* of p, so when applied to a value x we know which parts of x are evaluated during execution. This knowledge is used to make the algorithm efficient. For instance, applying a predicate *ordered* to $[1,3,2,4,5]$ may yield false, evaluating only the first three elements of the list. Observing laziness can often exclude large groups of values in a single test. In the example above, no additional lists starting with $[1,3,2]$ need to be tested.

We improve upon existing tools in several ways, mainly by using a size bound instead of a depth bound, and enumerating total values directly without intermediate partial values. We also present and evaluate several algorithms for parallel conjunction (reordering the operators of conjunctions for faster searches). In addition to theoretical arguments for the efficiency of our algorithm we demonstrate experimentally that it finds counterexamples to properties that existing tools do not find.

This paper also introduces the concept of sized functors, an interface of operators for defining this and other enumeration algorithms based on size-bounded search. By overloading these operators, definitions of enumerations can be reused between different enumeration libraries, speeding up experimentation with new algorithms and simplifying comparison of algorithms.

1 Introduction

The problem we solve in this paper starts with a predicate on a data type D, defined as a total (terminating) Boolean function $\mathsf{p} :: \mathsf{D} \to \mathsf{Bool}$. Ideally we would want a function $\mathsf{search_u} :: (\mathsf{D} \to \mathsf{Bool}) \to [\mathsf{D}]$ on such predicates such that $\mathsf{p\ x} \equiv \mathsf{x} \in \mathsf{search_u\ p}$. Essentially $\mathsf{search_u}$ inverts the predicate, finding the set (or list) of inputs for which it holds. From this most general function a whole family of other useful functions can be defined, for instance satisfiability checking:

$$\begin{aligned}&\mathsf{sat_u} :: (\mathsf{D} \to \mathsf{Bool}) \to \mathsf{Bool}\\&\mathsf{sat_u}\ \mathsf{p} = \mathsf{search}\ \mathsf{p} \not\equiv [\,]\end{aligned}$$

Or a counterexample search, checking if p is falsifiable and providing a witness if it is:

$$\begin{aligned}&\mathsf{ctrex_u} :: (\mathsf{D} \to \mathsf{Bool}) \to \mathsf{Maybe\ D}\\&\mathsf{ctrex_u} = \mathbf{case}\ \mathsf{search_u}\ (\mathit{not} \circ \mathsf{p})\ \mathbf{of}\\&\quad [\,] \quad\ \to \mathsf{Nothing}\\&\quad (\mathsf{x}\text{: }\_) \to \mathsf{Just\ x}\end{aligned}$$

The u in $\mathsf{search_u}$, $\mathsf{sat_u}$ and $\mathsf{ctrex_u}$ is for undecidable, which they are for non-trivial D (just encode any undecidable universally quantified formula as a Haskell predicate). But a semi-decidable search procedure is possible, so if p is satisfiable $\mathsf{sat_u}$ p eventually yields True, by enumerating all values $\mathsf{x} :: \mathsf{D}$ and testing p x (here it helps to assume p terminates). Similarly $\mathsf{ctrex_u}$ p eventually yields a counterexample if there is one. One way to define a semi-decidable $\mathsf{sat_u}$ is to first define a size bounded version of sat:

$$\mathsf{sat} :: \mathsf{Int} \to (\mathsf{D} \to \mathsf{Bool}) \to \mathsf{Bool}$$

This is a fully decidable (terminating) function, sat k p searches through the finite set of values of size up to k. Then $\mathsf{sat_u}$ can be defined by iterative deepening:

$$\mathsf{sat_u}\ \mathsf{p} = \mathsf{or}\ [\mathsf{sat\ k\ p} \mid \mathsf{k} \leftarrow [0\,..]]$$

This repeats work since each search is up to size k, not size k exclusively, but with exponential complexity in k the repetition has a limited impact on the total execution time since most time is spent on the last iteration (Korf, 1985). For a ctrex function, size-based iterative deepening has the advantage of producing a counterexample that is minimal in size. Such counterexamples are especially useful for debugging.

As long as every value has a size and there is a finite number of values of each size, the exact definition of size is not important for decidability, but it has an impact on practical performance (Duregård, Jansson, and Wang, 2012). In this paper, the size of a value in an algebraic data type refers to the number of constructor applications in the value. For primitive types like integers, size is defined by an algebraic encoding of the same type, for instance integers can be encoded as lists of binary digits and the size is the number of bits.

Notably, for $\mathsf{sat_u}$ and $\mathsf{ctrex_u}$ iterative deepening works for predicates on functions as well, because if there is a function f for which the predicate p f terminates, then there is a finite partial function $\mathsf{f'}$ for which $\mathsf{p\ f} = \mathsf{p\ f'}$ and finite partial functions can be enumerated by size (Claessen, 2012).

Efficient black-box search For property based testing, semi-decidability means we eventually find a counterexample to a property if there is one. But "eventually" is a small comfort if it takes years to find it.

In a white-box setting, we could possibly translate the source code of p into a logical formula and use a theorem prover or a similar tool for efficient search. This paper assumes a black-box setting: We either lack complete access to the source code of p, or it is too complex for white-box tools (for instance using incompatible language features or extensions).

In a truly black-box setting, our only option is to execute p on every value. In this paper predicates are black-box, but by shaking the box a little while computing p x, information on which parts of x are evaluated falls out. This analysis does not inspect the source of p, but it does expose some information about p that is usually hidden. Because of lazy evaluation, implementations of Haskell tend to keep track of this information at runtime. Extracting the information requires unsafe IO-actions which is not in the Haskell language standard, but is supported by compilers.

Equivalent values For every predicate, lazy evaluation implicitly defines an equivalence relation on its inputs. Two values x_1 and x_2 are equivalent with respect to a lazy predicate p iff they are equal in all parts evaluated by p. Formally, there is a partial value $x_\perp$ for which p $x_\perp$ is defined and $x_\perp \sqsubseteq x_1$ and $x_\perp \sqsubseteq x_2$ (x_1 and x_2 are more defined versions of $x_\perp$). In the remainder of this paper, when we speak of equivalent values we implicitly mean with respect to lazy evaluation of whichever predicate the search is applied to.

By lazy search we mean a search procedure that executes the predicate on a single value from each equivalence class that has any members within the size bound. The result of the procedure is the subset of the tested values that satisfy the predicate.

The worst case complexity of our algorithm, in the number of predicate executions, does not exceed the number of non-equivalent total values within the size bound. This is the best possible complexity in a black-box since the outcome of p cannot be predicted by executing it on a non-equivalent value. For fully eager predicates there are no equivalent values and our algorithm behaves like a fully black-box exhaustive search, testing every total value inside the size bound exactly once.

There is an existing tool called Lazy SmallCheck (Runciman, Naylor, and Lindblad, 2008) that also uses lazy search. Our algorithm improves upon Lazy SmallCheck in two critical aspects:

1. Our algorithm is linear in the number of non-equivalent total values. Lazy SmallCheck is linear in the number of non-equivalent partial values, a strictly greater set.

2. We use size instead of depth to bound the search, reducing the explosion in the size of the search space as the bound increases. This also means our counterexample search gives size-minimal counterexamples (as opposed to depth-minimal).

Paper layout The remainder of this paper is divided into five main sections. In Section 2 we present our algorithm and several examples, along with proofs or proof sketches for important correctness and performance properties. In Section 3 we present sized functors, the interface for defining enumerations for our algorithm. In Section 4 we outline our implementation of the algorithm as a Haskell library, and highlight the most important technical aspects of implementing it efficiently. In Section 5 we present conjunction strategies, a set of extensions to the algorithm that mitigate the effect of operand ordering on laziness in commutative logical operators (like conjunction). The increase in laziness is beneficial to the efficiency of the algorithm. In Section 6 we present experimental results of applying our algorithm to a handful of difficult problems, and evaluate the effect and relative performance of the various conjunction strategies presented.

2 The NEAT algorithm

NEAT operates by building a search tree. Each node in the tree is a value that the predicate is applied to. Constructors in nodes can be labelled as open. We call constructors that are not open *locked*. The intention is that for every node in the search tree, open constructors may be altered in child nodes whereas locked ones may not. The end goal of this section is an algorithm that takes a lazy predicate and constructs a search tree containing a single minimal value from each equivalence class.

The root of the search tree is the minimal value of the enumerated type with regards to a total order $\lessdot$ defined by:

- $\mathsf{a} \lessdot \mathsf{b}$ if the size of a is less than the size of b.
- In case of identical size, lexicographical ordering is used to establish a total order.

All constructors in the root node are labelled as open.

An *alternative* of an open constructor in a node is a modified copy of the node built by swapping the open constructor with another constructor of the same type. The new constructor itself is locked. Any values contained in the constructor are $\lessdot$-minimal and have only open constructors. Note that every open constructor has one alternative for every other constructor

of its type. Below are some examples of $\lessdot$-minimal values of Haskell types, and their alternatives. Open constructors are underlined:

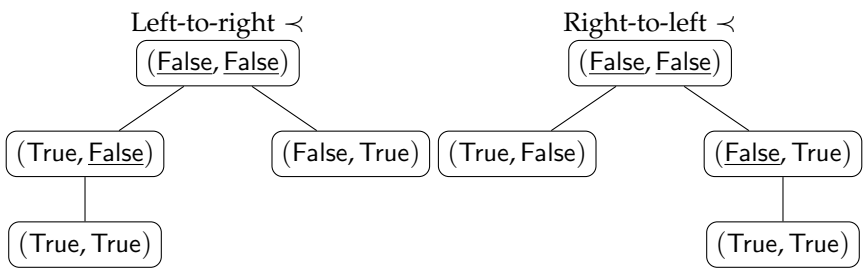
- (False, False) is the minimal value of (Bool, Bool). The left False has (True, False) as an alternative, the right has (False, True).
- Left False is minimal for Either Bool Bool. The Left constructor gives has Right False as alternative, the False constructor has Left True.

Complete search tree Before we define the full algorithm we show a simple procedure to build a search tree containing all values of the type exactly once. Then we modify this algorithm to exclude equivalent values.

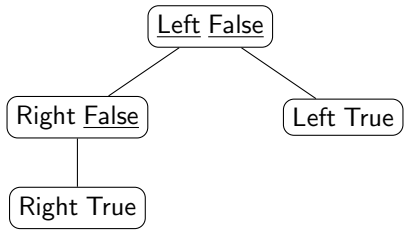
The algorithm is defined by how child nodes are built from a parent node. The first procedure one may consider is using all alternatives of all open constructors as children. While such a procedure finds all values, the trees produced often contain duplicate values. For instance (False, False) has (True, False) and (False, True) as children, and both of those have (True, True) as children. To avoid such duplicates, we apply the following algorithm to build the child nodes of a node x:

- Impose a total order $\prec$ on the open constructors of x. The order must be increasing in the nesting order of constructors in x but is otherwise unrestricted.
- For each open constructor c in x, build a child x' for every alternative of c, but lock all open constructors c' in x' where $c' \prec c$.

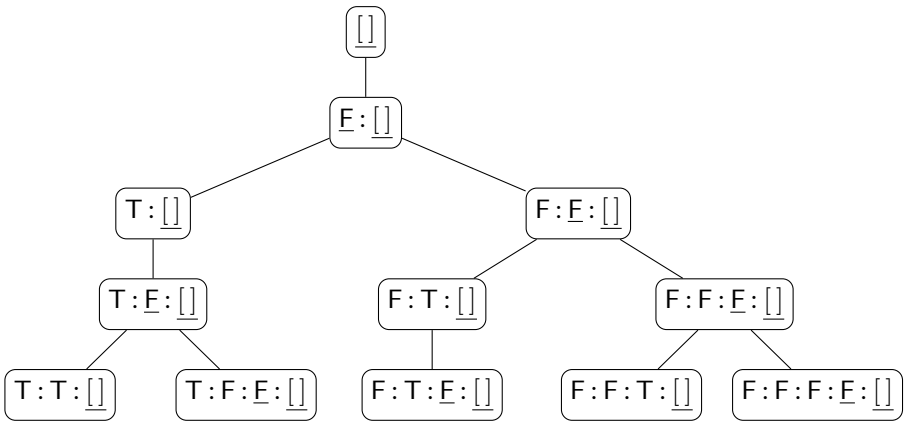
This prevents the repetition we saw earlier, because only one of the children has an open constructor. Depending on the order used, these are the possible search trees for (Bool, Bool):



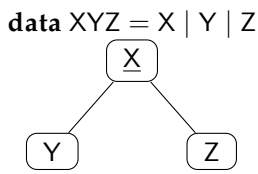
The restriction that $\prec$ is increasing in nesting order prevents locked constructors from being removed by swapping their parent constructors. This is important for example in the Either Bool Bool example: For Left False the only allowed order is that the Left constructor precedes the False constructor, because False is nested under Left. If the order was reversed, there would be repeated values. The search tree for Either Bool Bool is:



As another example of a complete search tree, here are the first few levels of the tree for lists of Booleans (T/F for True/False):



Since all examples above only contain data types with two constructors, they do not show any case where a single open constructor results in more than one child node. Here is an example of a tree for a data type with three constructors X, Y and Z:



It is fairly straightforward to prove that search trees produced by this algorithm are complete, meaning they contain all values, and free of duplicates. To prove completeness, we prove the existence of a path to any finite value x. We use an invariant that in every node along the path, all locked constructors match the constructor at the corresponding position in x. This invariant holds in the root node, because it has no locked constructors. In each node along the path there is a number of open constructors that differ from x, and in each node one of these is minimal with regards to $\prec$, and it has an alternative that swaps it into a constructor matching x. The child

node resulting from this alternative is the next node in the path to x. If any additional constructors are locked in this child node, they already match x. Compared to its parent, each node in the path locks at least one constructor to one that matches x. Since x has only finitely many constructors, there can only be finitely many nodes before a node in the path is identical to x.

To prove the uniqueness of values, we prove that there is exactly one path to x in the search tree. Consider any node n in the path to x. The next node in the path is a child of n that swaps an open constructor c to an alternative constructor c' that matches the constructor in x. We prove that all other child nodes of n lock at least one constructor that is incompatible with x, and so none of them can have a path to x. Looking at any other child of n: If it swaps c, it is to a different alternative than c' and the new constructor is locked and incompatible with x. If it swaps some other constructor c_1, either $c_1 \prec c$ or $c \prec c_1$. If $c_1 \prec c$, then c_1 already matches x and we swap it to an incompatible constructor. If $c \prec c_1$ then c is locked, and c is incompatible with x.

2.1 Avoiding equivalent values

To avoid equivalent values we impose two restrictions on the children of each node:

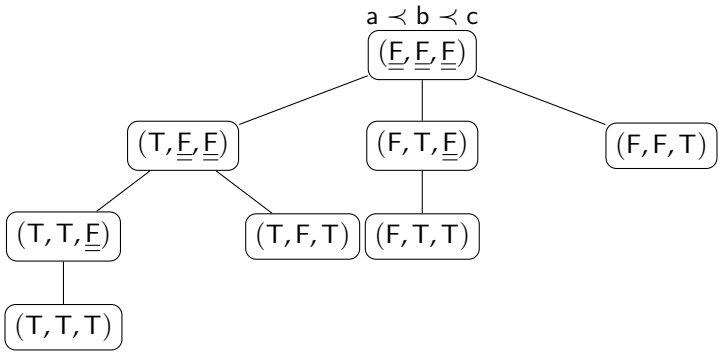
1. Only build child nodes from alternatives of open constructors that are evaluated by the predicate. Open constructors that are not evaluated are left open in all children.
2. Use evaluation order of the predicate to define $\prec$.

Implementing these restrictions require a means of detecting which constructors are evaluated and in which order. In our implementation (Section 4) we use Haskell IO-actions attached to constructors, but other techniques may be possible. In future examples we use double underlining to indicate constructors that are open and evaluated, and single underlined to indicate open but not evaluated.

The first restriction is crucial for avoiding equivalent values, since child nodes that swap an unevaluated constructor are always equivalent to their parent nodes. The necessity of the second restriction is not obvious, but consider this predicate:

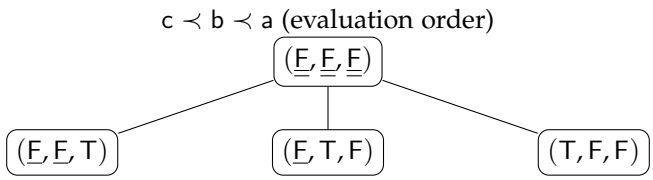
$$\mathsf{pred}\ (a, b, c) = c \vee b \vee a$$

There are many equivalent values here, because $\vee$ short-circuits without inspecting its second operand if the first operand is True. Using a left-to-right definition of $\prec$, means $a \prec b \prec c$ if we use a, b and c to refer to the constructors of the corresponding Boolean variables. With this ordering, the search tree contains all eight possible values:

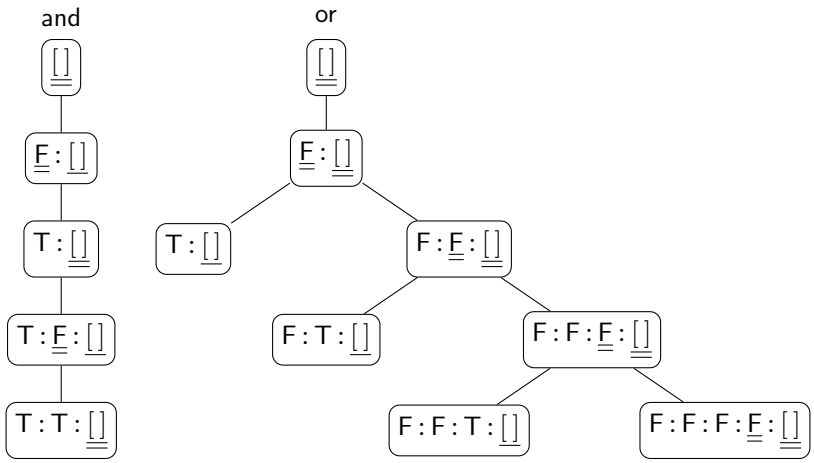


But it for instance (False, False, True) and (True, False, True) are equivalent with respect to pred so only one should be tested in a lazy search. The problem in this example is that whether or not a is evaluated depends on the value of c, so it makes sense to swap a and lock c (because the new a remains relevant) but not the other way around.

Using evaluation order to define $\prec$, so c $\prec$ b $\prec$ a, eliminates the problem. This results in only four executions of the predicate (the root plus one for each immediate child) before the algorithm terminates:



Examples To demonstrate the effect of laziness we show the search trees with the predicate being the standard Haskell functions and and or. The functions take lists of Booleans as input, so part of the complete tree for the same type is shown earlier. Both predicates evaluate the lists from left to right, but terminate when they encounter a False or True value respectively.



In both these examples, the size of the tree is linear in the depth (for or, left subtrees are always leaves). This is in contrast to the exponential size of the complete search tree for lists of Booleans.

Proving completeness The completeness proof for the complete search tree algorithm remains largely unaffected by avoiding equivalent values. The main difference is that the base case of the induction is not a value identical to x, but a value equivalent to x. If no constructors that differ from x are evaluated in a node, the value in the node is equivalent to x.

Similarly, one can prove that the tree never contains any two equivalent values, by observing that two different paths must differ at a constructor c that has been evaluated in some execution. This constructor is also evaluated by the predicate in the final value of both paths, because we know that any subsequent modifications done in either path must be later in the evaluation order and so it can not prevent c from being evaluated.

This result gives a very strong performance guarantee for our algorithm: The number of executions of the predicate is linear in the number of non-equivalent values. It also means that in the worst case of a fully eager predicate, the search performs as many executions as a straightforward search of all total values.

Building a search procedure The algorithm shown above deals with constructing the search tree. Since the algorithm already applies the predicate to the values in the tree, it is straightforward to extend it to report all non-equivalent values satisfying the predicate. Similarly one can build counterexample or satisfiability searches that terminate on the first falsifying/satisfying value in the tree.

Imposing a size limit In practice we want to perform a depth first search on the search tree to avoid keeping more than a single path in the search tree in memory at any point of the algorithm. This requires a size bound on the search tree, which can be implemented by excluding all children that exceed the size bound.

Here it helps that $\lessdot$ is increasing in size: Instead of computing the size of every child node it is enough to keep track of the remaining size in each node and the size increase each alternative gives.

Non-deterministic evaluation order In the examples above the evaluation order is deterministic, meaning it is stable between executions of the predicate. While this is usually the case, there are some exceptions. For instance when parallel execution is involved, scheduling decisions may alter the evaluation order.

If the evaluation order is non-deterministic, our algorithm becomes non-deterministic as well. For instance which counterexample is found by a counterexample search may differ between executions for some predicates. Satisfiability search remains deterministic however, although performance may vary between executions.

3 Sized Functors

This section defines the set of operations used to construct *enumerations* for our algorithm. Enumerations define the set of values enumerated by the search tree. Enumerations are sets with a built in notion of size. They are typically infinite, but a correctly constructed enumeration has finitely many values of every size, and our interface provides a simple way to ensure this. The basic operations for constructing enumerations are union and Cartesian product (and singletons and empty sets). Technically, enumerations are multisets, since there is nothing preventing construction of enumerations with duplicate values. In practice this is avoided by only using the union operator on disjoint sets.

Using enumerations generalize the algorithm described in the previous section in two directions:

- Open constructors are not necessarily a choice between actual constructors, but rather a choice between any disjoint sets of values.
- The definition of size is more flexible. It is not necessarily the number of constructors.

This generalization allows searching in other data types than purely algebraic ones, including both primitive numeric types and even function types.

The `size-based` package To develop the search algorithm we continually benchmarked our implementation, and compared it for correctness against slower reference implementations. The `size-based` package[1] simplifies this process by providing a common interface for defining enumerations. This interface is identical to the one used in Duregård, Jansson, and Wang, (2012) and Claessen, Duregård, and Pałka, (2015), for functional enumerations and uniform random selection respectively, so comparison with these tools is also simplified. The `size-based` package mainly provides three features:

- A common interface for defining size based enumerations.
- A type class for defining default enumerations for data types.
- Default enumerations for all data types in the Haskell Base package.

In Haskell nomenclature, a *functor* is a type constructor `f` that supports the operation $\langle\$\rangle :: (a \to b) \to f\ a \to f\ b$. For sets and other collection types this is intended as mapping a function over members. This operation is supported on enumerations. Functors that support product are called applicative functors (McBride and Paterson, 2008), and are members of the Applicative type class. There is also an Alternative class that provide a union combinator.

In addition to these operators, `size-based` introduces an operation to handle size: pay. It takes an enumeration and returns the same enumeration but increases the size of all contained values by one[2].

Following this scheme used by applicative functors, we call functors that provide the pay operation *sized functors* and introduce a type class called Sized. So for any sized functor E that is an instance of Sized, the following primitive operations are available:

$$
\begin{aligned}
\mathsf{pure} &:: a \to E\ a \\
\mathsf{empty} &:: E\ a \\
\mathsf{pair} &:: E\ a \to E\ b \to E\ (a, b) \\
\langle|\rangle &:: E\ a \to E\ a \to E\ a \\
\langle\$\rangle &:: (a \to b) \to E\ a \to E\ b \\
\mathsf{pay} &:: E\ a \to E\ a
\end{aligned}
$$

The pure and empty operations are for singleton and empty enumerations. For pure a, the value has size 0. The pair and $\langle|\rangle$ operators are for Cartesian product and union of enumerations. The map operator $\langle\$\rangle$ applies a function to all members of the enumeration (without affecting size).

[1] https://hackage.haskell.org/package/size-based

[2] The name pay was introduced in Duregård, Jansson, and Wang, 2012, because it provides a kind of guarded recursion where recursive procedures must pay from a limited resource each time a pay is encountered

Note that the application operation $\langle * \rangle$ from the Applicative class is not included as a primitive here, it is replaced by pair that more directly corresponds to Cartesian product. The application operation can be derived from pair and $\langle \$ \rangle$ as follows (where $ is the standard Haskell function application operator):

```
(⟨*⟩) :: E (a → b) → E a → E b
f⟨*⟩x = ($)⟨$⟩pair f x
```

The operations follow the standard algebraic laws of the corresponding multiset operations as well as several laws for pay (detailed in Duregård, Jansson, and Wang, (2012)). For this paper, the relevant laws are those regarding distributivity of pay over $\langle | \rangle$ and pair:

```
pay (pair a b) ≡ pair (pay a) b ≡ pair a (pay b)
pay (a⟨|⟩b) ≡ pay a⟨|⟩pay b
```

These follow naturally from the definition of size, paying for either component of a pair is the same as paying for the pair itself and paying for a union is the same as paying for both operands.

The following restriction is imposed on recursively defined enumerations: Any cyclic execution path must contain at least one application of pay. This restriction ensures the size invariant, that there is only a finite number of values of each size. These are all valid enumerations with regards to this restriction:

```
e₁ = pay (0⟨|⟩(+1)⟨$⟩e₁)
e₂ = 0⟨|⟩(+1)⟨$⟩pay e₂
e₃ = pay e₃
```

Whereas this is not valid, because it expands to an infinite union, thus violating the size invariant:

```
e = pay 0⟨|⟩(+1)⟨$⟩e
```

The Sized type class is defined by:

```
class Alternative f ⇒ Sized f where
  pay :: f a → f a
  pair :: f a → f b → f (a, b)
  pair a b = (, )⟨$⟩a⟨*⟩b
```

By having Alternative as a superclass, the $\langle * \rangle$, $\langle | \rangle$, $\langle \$ \rangle$, pure and empty operations are already provided and $\langle * \rangle$ can be used to define a default implementation for pair leaving pay as the only required additional operation compared to Alternative.

Enumerable types A type class Enumerable for enumerable data types is defined as such[3]:

```
class Enumerable a where
  enumerate :: Sized f ⇒ f a
```

Once an instance has been defined for a type using the sized functors operators it can be used to build enumerations of any type f a for which we have defined pay and the other sized functor operations.

3.1 Defining Enumerations

Enumerating algebraic types The enumeration for tuples (a, b) is simply pair enumerate enumerate, the tuple constructor does not use any size.

The standard enumeration for an algebraic data type is pay applied to the union of the enumerations for all constructors. To define an enumeration for a constructor we uncurry it (so each constructor of a data type A is a unary function of type (...((t1, t2), t3) ... tn) → A), then we apply the uncurried constructor (using ⟨$⟩) to the default enumeration for the tuple type. The instance for lists could look like this:

```
instance Enumerable a ⇒ Enumerable [a] where
  enumerate = pay (pure [] ⟨|⟩ uncurry (:) ⟨$⟩ enumerate)
```

It is easy to factor out common parts of these definitions and the actual instance for lists look more like this:

```
instance Enumerable a ⇒ Enumerable [a] where
  enumerate = datatype [c0 [], c2 (:)]
```

Where datatype is a function that computes the union of a list of enumerations and applies pay to them, and cn takes an n-ary constructor, uncurries it and applies it to enumerate. This can be done mechanically for algebraic types from their definition, and there is a Template Haskell meta-programming function for doing it automatically. In enumerations that follow this pattern, the size of a value is the number of constructors it contains.

Enumerating primitive types One way to enumerate primitive types such as integers is to encode them as algebraic types, and apply conversion functions to the enumerations for those types (using ⟨$⟩). For instance integers could be encoded as a list of bits. It is also possible to directly specify the encoding using the sized functor operations:

[3]The actual type is different because it deals with technicalities of guaranteeing sharing of enumerations of the same type, which is important for some Sized functors (such as FEAT). Instances can still be defined using the same operations though.

```
natPeano :: Sized f ⇒ f Integer
natPeano = pure 0⟨|⟩pay ((1+)⟨$⟩natPeano)

natBin :: Sized f ⇒ f Integer
natBin = pure 0⟨|⟩positive where
  positive = pay (pure 1
                  ⟨|⟩shift0⟨$⟩positive
                  ⟨|⟩shift1⟨$⟩positive)
  shift1 x = x ∗ 2 + 1
  shift0 x = x ∗ 2
```

These definitions capture the same multiset (the set of non-negative integers) but their definitions of size differ. In the former, the size of a number is the number itself. In the latter the size of a number is the number of significant digits in its binary representation.

Enumerating functions There are various ways to assign size to a function, and they depend on how functions are represented. The default enumeration in `size-based` builds functions from case distinctions and constants. These two together are sufficient to satisfy any terminating satisfiable higher order predicate. The size of such a function is the number of case distinctions it does plus the combined size of all its constants.

A case distinction is enumerated as a product of functions from the components of each constructor to the result of the original function. So an enumeration of case distinctions for $[a] \to b$ would be derived from an enumeration of the type $(b, a \to [a] \to b)$, corresponding to the nil and cons cases.

3.2 Basic sized functors

Counting A simple example of a sized functor is the counting functor, defined as:

```
newtype Count a = Count {count :: [Integer]}
```

This does not enumerate any values, it only counts the values of each size. The list is infinite for infinite enumerations, and `count e !! n` is the number of values of size `n` in the enumeration `e`. This means `pay` simply prepends a zero to the list (thus increasing the size of all contained values by one). Union is just pairwise addition of the list elements and for products every element is a sum of the different ways of dividing size between the components of the pair.

The counting functor is not used directly by our algorithm but it is useful because the sum of the first `n` elements of the list is the worst case number of predicate executions required by our algorithm for size bound `n`. As

such it can be used to measure how many executions we avoid for a given predicate (without actually enumerating all of them).

Values Another basic sized functor is the value functor:

```
newtype Values a = Values {runValues :: Int → [a]}
```

It sequentially enumerates all the total values of a given size. The length of the list `runValues e n` is equal to `count e !! n` (but computing the length is much slower). This is useful as a less efficient reference implementation to our lazy search procedure (by just filtering the list produced). Operations on values are straightforward:

```
pay (Values f) = Values f' where
  f' n | n > 0 = f (n − 1)
  f' n         = []
Values xs⟨|⟩Values ys = Values (λn → xs n ++ ys n)
pair (Values xs) (Values ys) = Values $ λn →
  [(x, y) | k ← [0 .. n], x ← xs k, y ← ys (n − k)]
```

4 Haskell Implementation

In this section we describe our Haskell implementation of the algorithm in Section 2. Implementing the algorithm requires a representation of total values annotated with open constructors that can be swapped and locked, and a way to construct the smallest such value of any enumerable type. To avoid computing sizes of values we extend the information in each node of the search tree with a remaining size, so a node is a value and an integer that is the difference between the size bound of the search and the size of the value in the node. We use the GADT extenstion to Haskell to define the Value type. This extension is not strictly needed but very handy, especially for applying simplifications that keep the representation small. Values are built from tuples, function application and unit values. To make values self-contained, open constructors are represented not just as a label, but by functions that take the current remaining size and builds lists of all possible alternative nodes.

```
data Node a = Node {sizeLeft :: Int, val :: Value a}
data Value a where
  Pair :: Value a   → Value b → Value (a, b)
  Map :: (b → a) → Value b → Value a
  Unit :: a                → Value a
```

```
Open :: Value a → (Int → [Node a]) → Value a
```

For explaining the Value data type, the following function for converting it to the value we represent, ignoring open constructors, is useful:

```
value :: Value a → a
value (Pair v1 v2) = (value v1, value v2)
value (Map f v)    = f (value v)
value (Unit x)     = x
value (Open v _)   = value v
```

To explain how the Open constructor works and how it relates to the examples in Section 2, consider this example of the representation of False, i.e. an open False constructor. In practice this would not be defined manually, but we show it here to explain the intention of our representation.

```
minBool :: Value Bool
minBool = Open (Unit False) alts where
  alts i = [Node i (Unit True)]
```

The alts function takes the remaining size and produces a new Node with the same size but with True replacing False, reflecting that this does not change the size of the value. For the value Nothing of type Maybe Bool, the representation would be equivalent to this definition:

```
minMaybeBool :: Value (Maybe Bool)
minMaybeBool = Open (Unit Nothing) alts where
  alts i | i <= 0 = []
  alts i          = [Node (i − 1) (Map Just minBool)]
```

Here the alts function produces no alternatives if the remaining size is zero, and otherwise replaces Nothing by Just False and decreases the remaining size by one, reflecting that the change increases size by one.

Minimal values Note that Value is not a sized functor, for instance it does not have an empty operation since it always represents a total value. Our sized functor is instead Minimal, intended to represent the minimal Value in an enumeration with respect to $\lessdot$ (see Section 2). This is fairly straightforward except for some recursive enumerations that have no finite values (and thus no minimal value), for instance e = pay e. Minimal is defined as follows:

```
data Minimal a = Pay (Minimal a)
               | Value (Value a)
               | Empty
```

In the problematic case, e is an infinite repetition of the Pay constructor, so a function of type Minimal a → Value a does not terminate. Instead we

have a function of type Minimal a → Int → Maybe (Value a) that takes a size bound and returns the minimal value or Nothing if the size bound is exceeded. We also make the function return what is left of the size, thus producing a Node:

```
minimal :: Minimal a → Int → Maybe (Node a)
minimal _       n | n < 0 = Nothing
minimal Empty   _         = Nothing
minimal (Pay s) n         = minimal s (n − 1)
minimal (Value vf) n      = Just (Node n vf)
```

To define the sized functor operations on Minimal we use the distributive laws for pay to move Pay constructors outwards. For products we have:

```
pair Empty _              = Empty
pair _ Empty              = Empty
pair (Pay a) b            = Pay (pair a b)
pair a (Pay b)            = Pay (pair a b)
pair (Value f) (Value g) = Value (Pair f g)
```

Most of the other operations are trivial (empty = Empty, pay = Pay etc). For $\langle|\rangle$, we use the distributive property pay a$\langle|\rangle$pay b ≡ pay (a$\langle|\rangle$b) to find which operand has the smallest value (biasing to the left in case of ties). When a minimal value is found the other operand is attached as an alternative to it. If the value is already an open constructor it is extended with an additional alternative.

```
Empty   ⟨|⟩m       = m
m       ⟨|⟩Empty   = m
Pay a   ⟨|⟩Pay b   = Pay (a⟨|⟩b)
Value vf⟨|⟩m       = Value (open vf m)
m       ⟨|⟩Value vf = Value (open vf m)

open :: Value a → Minimal a → Value a
open (Open v as) m = Open v (λi → maybeToList (minimal m i) ++ as i)
open v           m = Open v (maybeToList ∘ minimal m)
```

4.1 Building the search tree

The algorithm operates by taking a node, applying the predicate to its value and computing a list of child nodes based on observing the evaluation of open constructors. To separate the evaluation of the predicate from the computation of child nodes (useful when implementing conjunction strategies, see Section 5) we define an IO function that returns a value and another IO-action for computing the list of child nodes based on the evaluation of the value at the time the IO-action is performed. We call this

function observed because it returns a value in which evaluation is being observed. The type of observed is:

```
observed :: Node a → IO (IO [Node a], a)
```

For (m, x) ← observed n, x is equivalent to value n, but running m gives a new node for each alternative of every evaluated open constructor c in x, such that c has been swapped and all earlier evaluated constructors locked. Child nodes that exceed the size bound are not returned by m.

For example, if we have a node n representing the value (False, False), represented internally by something equivalent to Pair minBool minBool, although the actual construction is done by applying the minimal function to a class member of Enumerable. Further assume (m, x) ← obseved n, i.e. m is the observation action and x is the observed value. Immediately running m gives an empty list, reflecting that n has no child nodes if nothing in x is evaluated. Evaluating only the first or second value of x and then running m gives singleton lists containing the representations of (True, False) and (False, True) respectively. If both components of x are evaluated, the list produced by m has two elements. Depending on the evaluation order these can be either [(True, False), (False, True)] or [(False, True), (True, False)].

Evaluation is observed by attaching IO-actions to every open constructor in x. We experimented with different approaches for this, a simple one is to have an IO-action that writes its position in the value to a shared list (using a mutable reference). Then the evaluated constructors and their evaluation order can be extracted from the list after execution has terminated. A simple procedure then takes a value and the list of evaluated positions (in order of evaluation from first to last) and for each constructor swaps it to make new nodes for all alternatives and locks it to process the remainder of the evaluated constructors recursively.

We show a more efficient algorithm for building the child nodes. It has a shared mutable counter for every execution of the predicate and a mutable reference to an integer for each open constructor. When a constructor is evaluated it ticks up the counter and writes the counter's old value to its own reference. The order in which constructors are evaluated is fetched by reading from these references. We define observed such that it only traverses the value once to build all child nodes. The difficult case is for Pair values, where we apply a linear merge algorithm to combine the child nodes from the two components of the tuple.

We define a data type of alternative lists Alts. Each member of Alts contains a list of nodes built by switching a single evaluated open constructor to all possible alternative constructors, and locks all constructors that precedes it in the evaluation order. In addition to this list it contains two things: An ordinal (Int) indicating the evaluation order of the swapped constructor and a value that is built by locking the constructor (instead of swapping it) and all constructors before it in the evaluation order. We also define the

$\prec$ operation on the data type (see Section 2), and a convenience function amap that lifts a function on values to a function on alternative lists by applying it to both the locked value and all the swapped values:

```
data Alts a = Alts
  { evalOrder :: Int
  , locked    :: Value a
  , swapped   :: [Node a]
  }
r ≺ q = evalOrder r < evalOrder q
amap :: (Value a → Value b) → Alts a → Alts b
```

We implement the merge function, merge va vb rs qs where rs and qs are the alternative lists from two components of a pair value, in ascending order by $\prec$, and va and vb are values of the same type as the components of the pair. Initially va and vb are the original values as they were in the input node. In each recursive call va has locked all constructors evaluated before the first constructor in qs, and identically for vb and rs. The result is the list of alternative lists for the pair, satisfying the invariants on the Alts type.

```
merge :: Value a → Value b → [Alts a] → [Alts b] → [Alts (a, b)]
merge va vb [] r = map (amap (va`Pair`)) r
merge va vb r [] = map (amap (`Pair`vb)) r
merge va vb rs@(r : rs′) qs@(q : qs′)
  | q ≺ r     = amap (va`Pair`) q : merge va (locked q) rs qs′
  | otherwise = amap (`Pair`vb) r : merge (locked r) vb rs′ qs
```

Using this operation we first define alts :: Node a → IO Int → IO (IO [Alts a], a) that takes a node and an IO-action that ticks up a counter and produces the counter's old value. The result of alts is the observed value and an IO action for building the alternatives for all inspected constructors. In the Open case a shared reference is introduced using the attach function, and evaluation order is detected by reading this reference. Then observed can be defined on top of alts by extracting the new nodes from each alternative list. See Figure 2.1 for a complete definition of alts and observed.

Note that observed reverses the list of alternative lists before extracting the new nodes. This has the effect of placing the child nodes resulting from the last evaluated constructor first, which is a sound strategy both because this constructor is likely to have caused failure in the predicate (for satisfiability searches it may give faster results) and because it has locked the most constructors and thus tends to have fewer child nodes of its own.

```
alts :: Node a → IO Int → IO (IO [Alts a], a)
alts node tick = go (val node) where
  go :: Value a → IO (IO [Alts a], a)
  go (Pair va vb) = do
    (rs, a) ← go va
    (qs, b) ← go vb
    return (liftM2 (merge va vb) rs qs, (a, b))
  go (Map f v) = do
    (rs, a) ← go v
    return (fmap (map (amap (Map f))) rs, f a)
  go (Unit a) = return (return [], a)
  go (Open v altfun) = do
    (rs, a) ← go v
    (i, a') ← attach a
    let obs = i >>= λx → case x of
      Nothing → return []
      Just ix → case altfun (sizeLeft node) of
        [] → rs
        xs → fmap (Alts ix v xs:) rs
    return (obs, a')
  attach :: a → IO (IO (Maybe Int), a)
  attach a = do
      ref ← newIORef Nothing
      return (readIORef ref, unsafePerformIO $
            tick >>= writeIORef ref ∘ Just >> return a)
observed :: Node a → IO (IO [Node a], a)
observed node = do
  tick ← newCounter
  (rs, a) ← alts node tick
  return (fmap children rs, a)
  where
    children :: [Alts a] → [Node a]
    children rs = concatMap swapped (reverse rs)
    newCounter :: IO (IO Int)
    newCounter = do
      ref ← newIORef 0
      return (atomicModifyIORef ref (λi → (i + 1, i)))
```

Figure 2.1: Definition of observed in all its glory

```
satIO :: (a → Bool) → Node a → IO Bool
satIO p o = do
  (m, a) ← observed o
  return (p a) `orIO` (m >>= satChildren)
  where
    satChildren [] = return False
    satChildren (m : ms) = satIO p m `orIO` satChildren ms
    orIO :: IO Bool → IO Bool → IO Bool
    orIO b1 b2 = b1 >>= λb → if b then return True else b2
```

Figure 2.2: Definition of a simple satisfiability procedure

4.2 Searching

With observed defined, we can define a simple search procedure to determine bounded satisfiability, see Figure 2.2. More advanced search procedures can be defined similarly.

Testing framework To use NEAT for property based testing, we provide an iteratively deepening function test to search for a counterexample to a given property. Starting from size 0, it uses the counting functor (see Section 3) to calculate the worst case number of values and print it, then proceeds to search for a counterexample of this size. If it finds one it prints it and terminates, if not it prints the number of actual tests performed and recursively tests with an increased size bound.

```
test :: Enumerable a ⇒ (a → Bool) → IO ()
```

An example run could look like this (this is an abbreviated output of executing the Lambda program in Section 6):

```
Testing to size 0, worst case 0 tests
No counterexample found in 0 tests

[...]

Testing to size 3, worst case 5 tests
No counterexample found in 3 tests

[...]

Testing to size 16, worst case 571342 tests
No counterexample found in 5852 tests
```

```
Testing to size 17, worst case 1558413 tests
Counterexample found:
Let [(TUnit,0,Lit VUnit)]
  (Let [(TBool,0,Lit (VBool False))] (Var 0))
```

For a fully eager predicate, the number of tests consistently equals the predicted worst case.

5 Conjunction Strategies

In this section we introduce conjunction strategies: Algorithms to mitigate the effect of the operand order of logical operators in predicates. Ideally, each execution of such an operator is evaluated in the order that is most beneficial to searching (gives the smallest search tree).

Lazy SmallCheck (that operates on partial values), has a parallel conjunction operator that works as normal conjunction except it has the following property:

$$\bot \wedge \mathsf{False} = \mathsf{False}$$

If we have a partial value x and a Boolean p x ∧ q x, and x is not sufficiently defined to determine p x but happens to be sufficiently defined to determine q x to False, the result is False. If neither operand can be determined the first operand is evaluated first, in other words: $\bot_1 \wedge \bot_2 = \bot_1$. This means the search is always guided towards determining p x.

In NEAT we cannot have opportunistic parallel conjunctions as defined above, because we operate on total values. Still, an operand being ⊥ corresponds roughly to it evaluating at least one open constructor in our algorithm. So the ⊥ ∧ False case would be the left operand evaluating at least one open constructor and the right operand being false but only evaluating locked constructors.

We refer to this as opportunistic parallel conjunction, because it detects that the right operand is incidentally falsified from trying to falsify the left operand, but it never deliberately acts to falsify the right operand. An opportunistic parallel conjunction p x ∧ q y can reduce the size of the search tree under some conditions, particularly:

- All parts of y evaluated by q y are shared with x (particularly if x = y).
- All parts of y evaluated by q y are evaluated by p before p terminates.

If either of these conditions is not met, opportunistic parallel conjunction has no effect. Parallel conjunction tends to work best if p and q have the same input and the same evaluation order. An example for which parallel conjunction works is checking if a list is both ordered and and has no

repetitions. An example for which it does not work is checking if both the left and the right subtree of a binary tree are balanced (they operate on different values) or running a depth-first and a breadth-first traversal of the same tree.

There is a performance penalty associated with opportunistic parallel conjunction, since it does not short-circuit conjunctions. There is no upper bound for this penalty since q can be arbitrarily more expensive than p. Care must be taken to avoid using parallel conjunction where it is not beneficial. Also, sometimes the correctness or termination of the predicate depends on short-circuiting, like in $\neg\ (\mathsf{null}\ \mathsf{xs}) \wedge \mathsf{head}\ \mathsf{xs}\ ==\ 0$).

The basic principle of parallel conjunction is flipping the operands of some strategically chosen conjunctions. By conjunction we refers to an execution of a conjunction, so a single syntactic application of $\wedge$ can yield several conjunctions. Opportunistic parallel conjunction is one of many possible strategies. Before we define a few other strategies, we address some technical aspects of modifying evaluation order.

Coolean logic Conjunction strategies requires properties to be encoded with a Boolean type where operator evaluation order can be rearranged dynamically. This causes some syntactic overhead. NEAT has a Boolean type called Cool (for Concurrent Boolean). Cool is a deep embedding of Boolean logic with atoms, negation and two operators for parallel and sequential conjunction:

```
data Cool = Atom Bool
          | Not  Cool
          | And Cool Cool
          | Seq  Cool Cool
```

Sequential and parallel versions of other Boolean connectives are implemented as derived operators. The sequential conjunction operator Seq has the condition that if the first operand is false the second operand cannot be evaluated. The parallel conjunction operator And is considered fully commutative and no guarantees are given about the evaluation order. For the remainder of this section, any logical conjunctions refer to parallel ones unless stated otherwise.

Problems with conjunction evaluation We have identified two problematic cases for a conjunction $\mathsf{b} = \mathsf{p}\ \mathsf{x} \wedge \mathsf{q}\ \mathsf{x}$, that can be solved by strategically changing it to $\mathsf{q}\ \mathsf{x} \wedge \mathsf{p}\ \mathsf{x}$. For each problem we specify a condition on the evaluation that avoids it:

- *Optimal short-circuiting*: Suppose p x is True and q x is False (so b is False). In this situation p x evaluates some open constructors and q x

some. But if the expression was instead `q x ∧ p x`, the evaluation would short circuit and not evaluate `p x`. This means that swapping constructors evaluated by `p x` but not by `q x` could have been delayed or avoided completely, reducing the size of the search tree. We say the evaluation of a predicate is optimally short circuiting if it does not evaluate both operands of any false conjunction (meaning it never evaluates `True ∧ False`).

- *Fairness*: Suppose we have an unsatisfiable predicate `p`, and the search tree for `p` is finite independently of the size bound. In this case we want the search tree of `q ∧ p` to be bounded as well, but by default the search algorithm can spend an unbounded amount of time trying to satisfy `q` and for each satisfying value rediscover that `p` cannot be satisfied. We call a conjunction strategy that guarantees bounded search trees for `p ∧ q` fair[4].

Note that opportunistic conjunction does not guarantee either of these. Optimal short-circuiting is not done at all, and fairness is not guaranteed except in the special case that `q` is immediately false (i.e. when the search tree is bounded to a size of one).

Irreversible update limitation A very useful operation would be evaluating two conjuncts `p x` and `q x` and getting the results of both as well as the sets of open constructors evaluated by each. Optimal short-circuiting could be achieved by returning the second set if the first operand is true, and if both are false one could return the smaller set for additional benefits (subsumes opportunistic conjunction, which is the special case where the second set is empty).

Unfortunately such an operation is not compatible with our method of detecting evaluation of open constructors (attaching IO-actions to values). If `p x` is evaluated first, we can easily detect all constructors evaluated. Through this action, `x` has then been irreversibly updated, removing the IO-actions attached to the evaluated constructors. This means that if a constructor is subsequently evaluated by `q x` this is not detected. A Haskell compiler could provide an extension that would allow us to "unevaluate" constructors in `x` to re-enable the IO-actions for the evaluated constructors, or otherwise detect repeated access to a value (e.g. by creating a thunk that cannot be updated, so it is re-evaluated every time it is used), but if this is possible in GHC we failed to discover how.

With this limitation, the only way to compute the sets for each operand in all conjuncts is to execute them on different copies of `x`. While this

[4]consider the similarity to fairness in concurrent programming, if a process terminates in a bounded number of steps in a sequential setting it should also do so in a fair concurrent setting

works, doing it for every conjunction generated by a predicate would be prohibitively expensive.

5.1 Scheduling Strategies

A schedule is a data type describing which operators should be flipped in the execution of a coolean predicate. Essentially it is a tree of Booleans: True means flip and false means don't flip. This is useful both to describe our strategies and to implement them.

Optimal short-circuiting strategy Although the irreversible update limitation prevents us from getting optimal short-circuiting in a single execution of the predicate (discovering that the first operand is true irreversibly evaluates constructors), we do not need to repeat the execution for every conjunction. It is sufficient to run the predicate once to compute the optimal schedule (which positions should be flipped) and then once more to record the constructors this schedule evaluates.

In the worst case, this strategy executes the predicate twice instead of once for each node in the search tree, so the asymptotic complexity is unchanged (unlike most other conjunction strategies including opportunistic parallel conjunction). The first execution is called the lookahead execution, since the actual execution of the predicate uses it to predict its own result.

One interesting thing about optimal short-circuiting is that assuming that the predicate is deterministic (if it is not, the only reasonable answer to the question of satisfiability is "it depends") it can short-circuit sequential conjunctions. So if the precondition of a predicate is true in the lookahead execution, it must be true in the actual execution as well and checking it can be avoided altogether.

Fair parallel strategy One way to implement fairness is that for every conjunction generated by a predicate, we alternate between evaluating the left and right operands first as we progress down the search tree (so every node has a schedule based on its parent, and sibling nodes always have the same schedule). This is done through a function that takes a Cool and a schedule and produces a Boolean result and a new schedule that flips every evaluated conjunction.

Like any strategy that can flip evaluation order when the first operand is false, this strategy has a potentially unbounded increase in complexity. However, this strategy never evaluates both operands of a conjunction if they are both false. If combined with optimal short-circuiting it never evaluates both operands if either of them is false.

Constructor subset strategy Another operation that is permitted despite the irreversible update limitation is to determine if the open constructors evaluated by `q` is a subset of those evaluated by `p`. This can be done by checking that `p` does not evaluate any additional open constructors after `q` is evaluated. This suggests a strategy that flips a conjunction if the open constructors evaluated by the right operand is a subset of those evaluated by the left.

This strategy subsumes opportunistic parallel conjunction, with opportunistic conjunction being the special case that the right operand has an empty set of conjuncts.

Like the optimal short-circuit strategy this can be computed by a single lookahead execution of the predicate to compute the schedule and another execution to compute the set of evaluated open constructors. Unlike the other strategies this strategy always evaluates both operands of a conjunction.

Combining strategies It is possible to combine these strategies in various ways, for instance all three could be combined into a strategy with the following evaluation pattern (using lookahead to determine results before the actual evaluation):

- If both operands are true: Evaluate both operands.
- If one operand is false: Evaluate only the false one.
- If both operands are false: In the lookahead execution, the evaluation order is flipped compared to how this conjunction was evaluated in the parent node (initially not flipping). But if evaluating the other operand evaluates no additional open constructors in the lookahead execution, the order is flipped back.

This can be done using two executions of the predicate, the second has optimal short-circuiting but the first has no short-circuiting at all (because of the subset check).

6 Experimental Evaluation

We have performed two sets of experiments with NEAT, to compare the capacity for finding counterexamples compared to Lazy SmallCheck and to compare the impact of different conjunction strategies on search performance.

6.1 Finding counterexamples

We have tested our algorithm on four different programs. Three of them have deliberately injected bugs and one is deployed code for a Haskell library. Of the three synthetic bugs, one was constructed specifically for this paper and two were adapted from examples provided with Lazy SmallCheck. With Lazy SmallCheck being the most similar tool available, we chose to test our program specifically against problems that are not already solvable using Lazy SmallCheck. For this reason, we first tested each program against Lazy SmallCheck, and if a counterexample was found we modified the problem attempting to make it generally harder (for instance by extending the data type). For each problem we detail what modifications were made for this purpose.

Then we used NEAT on the properties, first searching for a minimal counterexample and then again to exhaustively search as many sizes as possible in one minute (same time limit as used for Lazy SmallCheck).

Program 1: Lambda For this problem we implemented an interpreter for simply typed lambda calculus with lambda abstractions, function applications, literals and let-expressions. To simplify type checking the language requires a type signature on every function argument.

We implemented a type checker, an evaluator and a transformation on the expression type that removes let-expressions using substitution.

The property tested is that for every type correct program `p`, evaluating `p` yields the same result as evaluating `p` with let-expressions removed.

We injected an error in the substitution function that caused a variable capture for nested let expressions. No counterexample was found by Lazy SmallCheck, so no modifications were needed.

Program 2: HSE The `haskell-src-exts` package (HSE, Haskell Source with Extensions) is a well used library for parsing Haskell programs, including several extensions of the language. It contains a massive syntax tree data type, a parser and a pretty printer.

We tested a round-trip property on the parser and pretty printer, expressing the requirement that the result of pretty printing a program should always be parseable. There are multiple counterexamples to this property. Most of these are caused by implicit invariants on the AST type that are not enforced by the type system (but are enforced by the parser), such as lists not being empty or an identifier starting with a capital letter.

Lazy SmallCheck successfully found numerous of these counterexamples, and for each one we modified the enumeration to either accommodate the invariant or exclude a problematic language construct altogether. When

we reached a subset of the AST type for which Lazy SmallCheck did not find any counterexamples, we tested NEAT on the same subset.

Program 3: Redblack Trees The RedBlack program is adapted from an example included in Lazy SmallCheck, which is in turn adapted from Okasaki, (1999). It is an implementation of Red-black trees, with an error injected in a balancing function. The property tested is that insertion preserves the Red-black invariant.

We modified the problem using Peano-encoded natural numbers instead of Int as the element type of the tree, to avoid differences in encoding between the tools. This unintentionally prevented Lazy SmallCheck from finding any counterexample, so no further modifications were made.

Program 4: RegExp This example was adapted from examples included in Lazy SmallCheck. It is a regular expression data type with some derived operations and an accept function that takes a regular expression and a string and checks if the string is accepted. The property tests an identity involving derived operations by checking equality on results from the accepts function.

To make the problem more difficult we changed the atom constructor for the expression type to take a list of symbols instead of a single symbol (interpreted as sequential composition), thus increasing the search space. This change prevented Lazy SmallCheck from finding a counterexample, so no further modifications were made.

Results The results of running the programs are shown in Table 2.5. It shows the maximal completed depth of Lazy SmallCheck after one minute, the maximal completed size of NEAT after one minute, and the size of the smallest counterexample. NEAT found counterexamples in all four cases. In three of the cases there was a comfortable margin between the size of the counterexample and the maximal completed size (more than 25% difference). In the RegExp benchmark NEAT only barely managed to find the counterexample.

6.2 Comparing conjunction strategies

To compare the outcome of various parallel conjunction strategies we implemented two benchmarks that use parallel conjunction. We then executed a search using iterative deepening that we let run for 30 seconds for each tested strategy. We compared the strategies by how large size was reached and how many tests (predicate executions) were required on each size. Six strategies were included in the comparison:

Program	Size Reached (NEAT)	Depth reached (LSC)	Counterexample size/depth
Lambda	25	4	17/5
HSE	18	4	14/6
RedBlack	44	3	26/4
Regexp	19	2	19/3

Table 2.5: Results of the counterexample search, showing the size reached by NEAT, depth reached by Lazy SmallCheck (no counterexample), and the size/depth of the smallest counterexample found by NEAT.

- **D**: Disables parallel conjunction altogether, never flipping any conjunctions.
- **O**: Optimizes short-circuiting using lookahead to flip conjuncts.
- **F**: Guarantees fairness by alternating between flipping and not flipping every individual conjunction.
- **OS**: Optimal short-circuiting and constructor-subset detection, flipping conjunctions when both are false and the second operand evaluates a subset of the open constructors the first one evaluates.
- **OF**: Combines optimal short-circuiting and fairness.
- **OSF**: Combines optimal short-circuiting, constructor-subset detection and fairness (in this order of priority).

There are no S and SF strategies (constructor-subset detection without optimal short-circuiting) because constructor-subset detection already uses lookahead, so not short-circuiting optimally would be senseless. The programs used in the benchmarks are as follows:

Program 5: Imperative This is a data type for imperative programs with variables, while-loops, if-statements and a few other constructs. We have several predicates that mirror the various invariants one could expect in various stages of a compiler: No unused variables, no nested if-statements, no skip operations, all variable declarations on top level. We also have a predicate that checks that every used variable is bound.

The benchmark is searching for programs that satisfy all these properties, such that they could be used to test subsequent stages of a compiler for the language. Each property is a separate predicate on programs, and each of them use only ordinary sequential conjunction, but they are combined with parallel conjunction (so each run of the predicate contains a fixed number of parallel conjunctions).

Program 6: Typecheck This benchmark is identical to the Lambda program, but instead of searching for a counterexample we search for all type correct programs. The typechecker is modified to use parallel conjunction wherever applicable, including in equality comparisons of types.

Results The results for the benchmarks are found in Tables 2.6 and 2.7. For comparison the tables include the total number of values in the search space, which is the number of tests required if laziness is not used at all.

In the case of the Imperative benchmark, conjunction strategies were clearly advantageous. The best strategies (OF and OS) increased the size reached by several steps and decreased the number of tests needed by an order of magnitude compared to disabling parallel conjunctions.

In the case of Typechecking, the effects were not as dramatic but still generally advantageous with two of the strategies reaching a higher size bound.

The difference between strategies in the tests demonstrate that there is no "one strategy to rule them all", but in both cases strategies become better with optimal fairness than without.

Size	20	21	22	23	24
Total	220887	673840	2058353	6301731	19323331
D	4381	X	X	X	X
O	2094	5892	X	X	X
F	952	2103	4587	X	X
OSF	644	1521	3226	X	X
OF	436	949	2050	4418	X
OS	398	857	1829	3906	X

Table 2.6: Number of tests (x1000) required with different strategies (lower is better). Execution was halted after 30 seconds.

Size	26	27	28
Total	17154564	48511651	138824320
D	4978	X	X
F	4534	X	X
OS	2615	X	X
OSF	2289	X	X
O	2730	5272	X
OF	2490	4835	X

Table 2.7: Number of tests (x1000) required for Program 6 with different strategies (lower is better). Execution was halted after 30 seconds.

7 Related Work

Lazy SmallCheck Described by Runciman, Naylor, and Lindblad, (2008), Lazy SmallCheck is the most closely related pre-existing tool. Like our algorithm, Lazy SmallCheck uses inherent laziness of Haskell properties to reduce the search space and find counterexamples faster. The two main differences between our algorithm and Lazy SmallCheck are:

1. Lazy SmallCheck uses a depth-bound instead of a size bound.

2. Lazy SmallCheck uses partial values instead of total values.

Using a depth bound means that, contrary to what the name of the library suggests, tested values are shallow, but not necessarily small. Even for simple data types like lists of lists, the maximal size of a value is exponential in its depth. The number of values in a data type is typically doubly exponential in the depth bound, so depth-bounded search tends to follow a pattern of progressing quickly through a few sparsely populated sets and then hitting a wall where the next step takes ages. In these cases only the first couple of seconds of testing give any solid guarantees on what has been covered, and though it keeps testing values after that it does not cover any well defined sets after that point (it is difficult to appreciate the difference between testing all values to depth 3 and testing all values to depth 3 and then 20 more minutes of testing and such). For counterexamples there is an additional problem: Reported counterexamples are shallow (minimal depth) but not necessarily small (minimal size).

To illustrate the difference between using partial values and total values, consider a data type **data** Exp = Var Int | App Exp Exp. To falsify a predicate p :: Exp → Bool, Lazy SmallCheck would first test p ⊥. If the result is ⊥ it may test p (Var ⊥) and if the result is once again ⊥ further refine it to p (Var 0). Then it would try other variables before backtracking and changing the Var to an App. Our algorithm by contrast would start with Var 0 (the minimal value), execute the predicate on it and detect that both Var and 0 are evaluated, then proceed much like Lazy SmallCheck first testing different variables before swapping Var. If p does not inspect its argument, there is no difference between the algorithms but in the case that it does, this simple example executing p on a single total value does the job of executing it on three partial values. There are other examples where the number of partial values is exponential in the size and the number of total values linear.

In Reich, Naylor, and Runciman, (2013) several extensions and possible future extensions to Lazy SmallCheck are described, some of which may apply to our work as well.

KORAT The tool KORAT for Java (Boyapati, Khurshid, and Marinov, 2002) is also similar to our algorithm although it is not black-box (it performs source-to-source transformations).

FEAT See also the chapter in this thesis (Paper I).

(Functional Enumeration of Algebraic Data types, Duregård, Jansson, and Wang, (2012)) is an earlier enumeration algorithm also based on sized-functors (although the name is not introduced there, it has the same operations). It is capable of both exhaustive enumeration and random or systematic sampling by size. It provides cardinality computations (like the counting functor) and an injective indexing function (hence *Functional* Enumerations) to quickly compute any value in the enumeration by its size and index in the enumeration. This allows the tool to make meaningful progress beyond the exponential explosion. FEAT does not identify equivalent values, so for predicates with preconditions that are difficult to satisfy by randomly picking values, this progress is very limited.

Uniform selection See also the chapter in this thesis (Paper III).

In Claessen, Duregård, and Pałka, (2015) another sized functor is introduced, it also uses laziness but is geared towards uniform selection from an enumeration instead of systematic enumeration. It behaves similarly to a breadth first search version of the algorithm described in this paper, but with uniformly random ordering of found values. The uniformity is achieved using cardinality computations like those done in the counting functor described in this paper.

QuickCheck (Claessen and Hughes, 2000) is a widely used tool for random property based testing that has inspired most other tools described here including FEAT and Lazy SmallCheck. It does not identify equivalent values and it does not have a general scheme to automatically create generators for test data, instead the user must write a generator for each type.

EasyCheck (Christiansen and Fischer, 2008) is a testing library written in Curry, using logic programming features such as free variables to generate test data. Instead of iterative deepening it uses a technique called level diagonalization that interleaves values generated with different depth bounds.

The Small Scope Hypothesis argues that most bugs can be found by testing the program for all test inputs within some small scope (Jackson, 2006; Jackson and Damon, 1996; Andoni, Daniliuc, and Khurshid, 2003).

8 Conclusions and future work

We have developed NEAT, an efficient algorithm for size-bounded lazy search. NEAT has improved algorithmic complexity compared to previously available tools (Lazy SmallCheck), because NEAT operates on total instead of partial values.

Another difference compared to Lazy SmallCheck is that we opted for a size-limited search (instead of a depth-limited). While this is not a fundamental requirement for the algorithm (a depth-based implementation would also be possible) we believe size-based algorithms are generally better suited for practical testing. To be precise, we argue for two conjectures:

1. If no counterexample is found, a size-based search tends to provide stronger claims of correctness. For instance a depth-based approach may exclude all inputs up to size three, a set including thousands of values, and in the same time a size based approach excludes all inputs up to size fourteen, a set including millions of values. While both approaches tests approximately as many values, the finer granularity of the the size-based approach means it can report excluding entire sets of values instead of excluding an unknown fraction of a much larger set.

2. While it is possible to construct scenarios where either size-based or depth-based searches find counterexamples faster, we believe that the latter scenarios are generally more contrived.

The first conjecture holds for all properties where the number of required tests grows faster with depth than with size. Claiming that this is generally the case should be uncontroversial.

The second conjecture relies more on the nature of "typical bugs" and is much more difficult to verify empirically. Slightly simplified, one can say the following about depth-based versus size based approaches: If the tree structure of the minimal counterexample is sparse, a size based approach is faster. If the minimal example is dense, like a complete tree, a depth-based approach is faster. This can be shown by counting the number of values of the same size versus same depth as the counterexample; for a sparse tree the former number is smaller and for a dense tree the latter is smaller. Thus, our conjecture boils down to a claim that minimal counterexamples are sparse in most non-trivial practical cases. We argue that the tree structure of a minimal counterexample tends to have a few paths that are essential to the failure, and branches from these paths are just leaf values required to construct the counterexample. This pattern has been observed previously in the context of counterexample shrinking used by QuickCheck (Hughes, 2007), where entire branches of counterexamples can often be replaced by leaves.

In this paper, we experimentally verify that our algorithm is capable of finding counterexamples for some example properties that are out of reach of Lazy SmallCheck. The bugs in the examples were artificially introduced and the examples were included on the criteria that Lazy SmallCheck failed to falsify them. Results showed NEAT was able to falsify them, generally with a wide margin (capable of exhaustively testing to larger sizes than the smallest counterexample). The results are consistent with the conjectures above, but due to the small sample size it is not a definitive proof.

Further experiments With several tools for property based testing in Haskell available, a large repository of realistic properties with counterexamples of various difficulties would be useful. This would allow a broader comparison of the available tools and assess their relative strengths and weaknesses. Recent versions of the TIP problem collection (Claessen et al., 2015) contains several such problems, and it supports generating Haskell code including properties for Lazy SmallCheck. As such it could form a base for future experiments in this area. Since TIP has multiple backends, it could also facilitate comparisons with white-box tools.

More powerful conjunction strategies In this paper we present three basic conjunction strategies (and several combinations of those), but many more could be implemented.

All the strategies presented in this paper are geared towards reducing the search space, but strategies can also reduce the execution time of properties. For instance if computing an operand of a conjunction only evaluates locked constructors, it need not be re-evaluated in this search tree. A generalization of this is that an operand that evaluates a set of open constructors does not need to be re-evaluated until one of those is changed. This could increase performance for expensive predicates, at the cost of some extra bookkeeping.

These issues relate closely to SAT-solving and concurrent programming, and lessons from those fields could perhaps be adapted to this area.

Breath first search Replacing iterative deepening with a breadth first search would improve performance, if the high memory usage typically associated with it can be avoided.

It may be possible to modify the algorithm into an exact size search, executing the predicate at most once for every value of an exact size (instead of up to that size). One way would be as follows: When an open constructor is evaluated, it is immediately swapped to an alternative if using the original constructor prevents the construction of a value of the desired size. For instance for a binary tree (and a large size bound) it could not choose

a leaf constructor as the root, immediately opting for a branch instead. It could use a leaf constructor for whichever subtree is evaluated first (because the remaining constructor in the other subtree can still expand to a large enough value).

This requires an oracle that determines when using a constructor prevents generation of sufficiently large values. In general it is not enough to check that any other open constructors remain, we need to check if the product of all unevaluated open constructors can yield a value of the correct size. This can be done by using the counting functor described in this paper, but it would cause a lot of overhead.

Parallel search A simple feature to add is parallel search (as in parallel execution, not related to parallel conjunction). The implementation we have already operates on an explicit stack of nodes in the search tree, and an easy way to parallelize would be to have multiple threads continuously consuming nodes from the stack and inserting children. This would slightly deviate from the depth-first search pattern and occasionally process two sibling nodes in the search tree in parallel, but we hypothesize that it would not have any of the memory problems that breadth first searches have. As the algorithm is already non-deterministic, there would be no further loss of determinism from this straightforward implementation of parallel execution.

Default conjunction strategy Our experiments verify that different conjunction strategies work best for different problems, and that which strategy is selected can have a large impact on performance. This makes it difficult to choose a default strategy, but our experiments hint at two things:

- Enabling optimal short-circuiting is generally a good idea, and it should be enabled by default if there are any parallel conjunctions in the predicate.
- It may be possible to predict an optimal strategy automatically by running multiple strategies on small sizes and extrapolate their relative performance on the given predicate to larger sizes.

Availability The package (`lazy-search`) is available on Hackage (hackage.haskell.org). For a NEAT party trick, install the package, load Control.Search in GHCi and run:

```
test (/= "you can never find this").
```

Paper III

Generating Constrained Random Data with Uniform Distribution

This chapter is adapted from a paper published in the Journal of Functional Programming

Generating Constrained Random Data with Uniform Distribution

Koen Claessen, Jonas Duregård, Michał H. Pałka

Abstract

We present a technique for automatically deriving test data generators from a given executable predicate representing the set of values we are interested in generating. The distribution of these generators is uniform over values of a given size. To make the generation efficient we rely on laziness of the predicate, allowing us to prune the space of values quickly. In contrast, implementing test data generators by hand is labour intensive and error prone. Moreover, handwritten generators often have an unpredictable distribution of values, risking that some values are arbitrarily underrepresented. We also present a variation of the technique that has better performance, but where the distribution is skewed in a limited, albeit predictable way. Experimental evaluation of the techniques shows that the automatically derived generators are much easier to define than hand-written ones, and their performance, while lower, is adequate for some realistic applications.

1 Introduction

Random property based testing has proven to be an effective method for finding bugs in programs (Claessen and Hughes, 2000; Arts et al., 2006). Two ingredients are required for property based testing: A *test data generator* and a *property* (sometimes called a test oracle). For each test, the test data generator generates input to the program under test, and the property checks whether or not the observed behaviour is acceptable. This paper focuses on the test data generators.

The popular random testing tool QuickCheck (Claessen and Hughes, 2000) provides a library for defining random generators for data types. Typically, a generator is a recursive function that at every recursion level chooses a random constructor of the relevant data type. Relative frequencies for the constructors can be specified by the programmer to control the distribution. An extra resource argument that shrinks at each recursive call is used to control the size of the generated test data and ensure termination.

The above method for test generation works well for generating structured data. But it becomes much harder when the data must *satisfy an extra condition*. A motivating example is the random generation of programs as test data for testing compilers. In order to successfully test different phases

```
data Expr = Ap Expr Expr Type
          | Vr Int
          | Lm Expr
data Type = A | B | C
          | Type :→ Type

check :: [Type] → Expr → Type → Bool
check env (Vr i)      t          = env !! i == t
check env (Ap f x tx) t          = check env f (tx :→ t) && check env x tx
check env (Lm e)      (ta :→ tb) = check (ta : env) e tb
check env _           _          = False
```

Figure 3.1: Data type and type checker for simply-typed lambda calculus. The Type in the Ap nodes represents the type of the argument term.

of a compiler, programs not only need to be grammatically correct, they may also need to satisfy other properties such as all variables are bound, all expressions are well-typed, certain combinations of constructs do not occur in the programs, or a combination of such properties.

In previous work by some of the authors, it was shown to be possible but very tedious to manually construct a generator that (a) could generate random well-typed programs in the polymorphic lambda-calculus, and at the same time (b) maintain a reasonable distribution such that no programs were arbitrarily excluded from generation (Pałka et al., 2011; Pałka, 2012).

The problem is that generators mix concerns that we would like to separate: (1) what is the structure of the test data, (2) which properties should it obey, and (3) what distribution do we want.

In this paper, we investigate solutions to the following problem: Given a definition of the structure of test data (a data type definition), and given one or more executable predicates (functions computing a Boolean value) on the data type, can we automatically generate test data that satisfies all the predicates and at the same time has a predictable, useful distribution?

To be more concrete, let us take a look at Figure 3.1. Here, a data type for typed lambda expressions is defined, together with a function that given an environment, an expression, and a type, checks whether or not the expression has the stated type in the environment. From this input alone, we would automatically generate random well-typed expressions with a good distribution.

What does a 'good' distribution mean? First, we need to have a way to restrict the size of the generated test data. In any application, we are only ever going to generate a finite number of values, so we need a decision

on what test data sizes to use. An easy and common way to control test data size is to control the *depth* of a term. This is for example done in SmallCheck (Runciman, Naylor, and Lindblad, 2008). The problem with using depth is that the number of terms grows extremely fast as the depth increases (doubly exponential even for simple binary trees). Moreover, useful distributions for sets of trees of depth d are hard to find, because there are many more complete trees of depth d than there are sparse trees. This may lead to an overrepresentation of almost full trees in randomly generated values.

Another possibility is to work with the set of values of a given *size* n, where size is understood as the number of data constructors in the term. Previous work by one of the authors on FEAT (Duregård, Jansson, and Wang, 2012) has shown that it is possible to efficiently index in, and compute cardinalities of, sets of terms of a given size n. This is the choice we make in this paper.

The simplest useful and predictable distribution that does not arbitrarily exclude values from a set is the *uniform distribution*, which is why we chose to focus on uniform distributions in this paper. We acknowledge the need for other distributions than uniform in certain applications. However, we think that a uniform distribution is at least a useful building block in the process of crafting test data generators. We anticipate methods for controlling the distribution of our generators in multiple ways, but that remains future work.

Our first main contribution in this paper is an algorithm that, given a data type definition, a predicate, and a test data size, generates random values satisfying the predicate, with a perfectly uniform distribution. It works by first computing the cardinality of the set of all values of the given size, and then randomly picking indices in this set, computing the values that correspond to those indices, until we find a value for which the predicate is true. The key feature of the algorithm is that every time a value `x` is found for which the predicate is false, it is removed from the set of values, together with all other values that would have lead to the predicate returning false with the same execution path as `x`. We also outline a proof that this sampling procedure is uniform.

Unfortunately, perfect uniformity turns out to be too inefficient in many practical cases. We have also developed a backtracking-based generator that is more efficient, but has no guarantees on the distribution. Our second main contribution is a hybrid generator that combines the uniform algorithm and the backtracking algorithm, and is ‘ almost uniform’ in a precise and predictable way.

This paper extends and improves a paper presented at FLOPS 2014 (Claessen, Duregård, and Pałka, 2014). The technical content is essentially unchanged, but we made several presentation and restructuring modifi-

cations. In this version we expand the description of the algorithm (Section 3), provide a detailed example of its operation (Section 3.2), demonstrate that the distribution of the generated values is uniform (Section 4), and discuss an alternative algorithm better suited for non-deterministic predicates (Section 5.4).

2 Generating Values of Algebraic Data Types

In this section we explain how to generate random values of an algebraic data type (ADT) uniformly. Our approach is based on a representation of sets of values that allows efficient *indexing*, inspired by FEAT (Duregård, Jansson, and Wang, 2012), which is used to map random indices to random values. In the next section we modify this procedure to efficiently search for values that satisfy a predicate.

Algebraic Data Types (ADTS) are constructed using units (atomic values), disjoint unions of data types, products of data types, and may refer to their own definitions recursively. For instance, consider these definitions of Haskell data types for natural numbers and lists of natural numbers:

```
data ℕ = Zr | Sc ℕ
data ListNat = Nil | Cons ℕ ListNat
```

In general, ADTS may contain an infinite number of values, which is the case for both data types above. Our approach for generating random values of an ADT uniformly is to generate values of a specific *size*, understood as the number of constructors used in a value. For example, all of `Cons (Sc (Sc Zr)) (Cons Zr Nil)`, `Cons (Sc Zr) (Cons (Sc Zr) Nil)` and `Cons Zr (Cons Zr (Cons Zr Nil))` are values of size 7. As there is only a finite number of values of each size, we can create a sampling procedure that generates a uniformly random value of ListNat of a given size.

2.1 Indexing

Our method for generating random values of an ADT is based on an *indexing* function, which maps integers to corresponding data type values of a given size (a procedure also known as *unranking* (Knuth, 2006)).

$$\text{index}_{S,k} : \{0 \ldots |S_k| - 1\} \to S_k$$

Here, S is the data type, and S_k is the set of k-sized values of S. The intuitive idea behind efficient indexing is to quickly calculate *cardinalities* of subsets of the indexed set. For example, when $S = T \oplus U$ is a sum type,

then indexing is performed as follows:

$$\text{index}_{T\oplus U,k}(i) = \begin{cases} \text{index}_{T,k}(i) & \text{if } i < |T_k| \\ \text{index}_{U,k}(i - |T_k|) & \text{otherwise} \end{cases}$$

When $S = T \otimes U$ is a product type, we need to consider all ways size k can be divided between the components of the product. The cardinality of the product can be computed as follows:

$$|(T \otimes U)_k| = \sum_{k_1+k_2=k} |T_{k_1}||U_{k_2}|$$

When indexing $(T \otimes U)_k$ using index i, we first select the division of size $k_1 + k_2 = k$, such that:

$$0 \le i' < |T_{k_1}||U_{k_2}| \quad \text{where} \quad i' = i - \sum_{\substack{l_1<k_1 \\ l_1+l_2=k}} |T_{l_1}||U_{l_2}|$$

Then, elements of T_{k_1} and U_{k_2} are selected using the remaining part of the index i'.

$$\text{index}_{T\otimes U,k}(i) = (\text{index}_{T,k}(i' \text{ div } |U_{k_2}|), \text{index}_{U,k}(i' \text{ mod } |U_{k_2}|))$$

In the rest of this section, we outline how to implement indexing in Haskell.

2.2 Representation of Spaces

We define a Haskell Generalized Algebraic Data Type (GADT) Space to represent ADTs, and allow efficient cardinality computations and indexing.

```
data Space a where
  Empty :: Space a
  Pure  :: a          → Space a
  (:+:) :: Space a → Space a → Space a
  (:*:) :: Space a → Space b → Space (a, b)
  Pay   :: Space a → Space a
  (:$:) :: (a → b) → Space a → Space b
```

Spaces can be built using four basic operations: Empty for empty space, Pure for unit space, (:+:) for the (disjoint) union of two spaces and (:*:) for a product. Spaces also have an operator Pay which represents a unit cost imposed by using a constructor. The last operation (:$:), applies a function to all values in the space.

It is possible to construct spaces with duplicate elements from these operations, although it is rarely useful in practice (typically the operands of :+:

are disjoint and functions used in $:\$:$ are injective, particularly constructor functions). This means that in general there is a multiset of values of any size and whenever we speak of uniform sampling procedures it is understood to be uniform over the set of occurrences of values, not over the set of values themselves. Simply put: repeated values are overrepresented exactly as one might expect from a uniform sampler of a multiset.

A convenient operator on spaces is the lifted application operator, that takes a space of functions and a space of parameters and produces the space of all results from applying the functions to the parameters:

$$(<*>) :: \mathsf{Space}\ (\mathsf{a} \to \mathsf{b}) \to \mathsf{Space}\ \mathsf{a} \to \mathsf{Space}\ \mathsf{b}$$
$$\mathsf{s}_1 <*> \mathsf{s}_2 = (\lambda(\mathsf{f}, \mathsf{a}) \to \mathsf{f}\ \mathsf{a})\ :\$:\ (\mathsf{s}_1\ :*:\ \mathsf{s}_2)$$

With the operators defined above, the definition of spaces mirrors the definitions of data types. For example, spaces for the $\mathbb{N}$ and ListNat data types can be defined as follows:

$$\mathsf{spNat} :: \mathsf{Space}\ \mathbb{N}$$
$$\mathsf{spNat} = \mathsf{Pay}\ (\mathsf{Pure\ Zr}\ :+:\ (\mathsf{Sc}\ :\$:\ \mathsf{spNat}))$$
$$\mathsf{spListNat} :: \mathsf{Space\ ListNat}$$
$$\mathsf{spListNat} = \mathsf{Pay}\ (\mathsf{Pure\ Nil}\ :+:\ (\mathsf{Cons}\ :\$:\ \mathsf{spNat} <*> \mathsf{spListNat}))$$

Unit constructors are represented with Pure, whereas compound constructors are mapped on the spaces for the types they contain. In this example, Pay is applied each time we introduce a constructor, which makes the size of values equal to the number of constructors they contain. This is a common pattern, but the user may choose to assign costs differently, which would change the sizes of individual values and consequently the distribution of size-limited generators. The only rule when assigning costs is that all recursion is guarded by at least one Pay operation, otherwise the sets of values of a given size may be infinite, causing non-terminating cardinality computations.

2.3 Uniform sampling by size

Uniform sampling on spaces can be reduced to two subproblems: Extracting the finite set of values of a particular size, and uniform sampling from such sets. Assume we have a data type Set a for finite multisets, constructed by combining the empty set ($\{\}$), singleton sets ($\{\mathsf{a}\}$), (disjoint) union ($\uplus$) and Cartesian product ($\times$). We can also apply a function f to all members set s with fmap f s, such that fmap f a = fmap f (sized a k). From the definition of a finite set, its cardinality can be defined as follows:

$$|\{\}| = 0$$
$$|\{\mathsf{a}\}| = 1$$

$$
\begin{aligned}
|a \times b| &= |a| * |b| \\
|a \uplus b| &= |a| + |b| \\
|\mathsf{fmap}\ f\ a| &= |a|
\end{aligned}
$$

Guided by this definition we can define an indexing function on the type that maps integers in the range $(0, |a| - 1)$ to (occurrences of) values in the multiset:

$$
\begin{array}{lll}
\mathsf{indexSet}\ \{a\} & 0 & = a \\
\mathsf{indexSet}\ (a \uplus b) & i \mid i < |a| & = \mathsf{indexSet}\ a\ i \\
\mathsf{indexSet}\ (a \uplus b) & i \mid i \geqslant |a| & = \mathsf{indexSet}\ b\ (i - |a|) \\
\mathsf{indexSet}\ (a \times b) & i & = (\mathsf{indexSet}\ a\ (i \div |b|), \mathsf{indexSet}\ b\ (i \bmod |b|)) \\
\mathsf{indexSet}\ (\mathsf{fmap}\ f\ a)\ i & & = f\ (\mathsf{indexSet}\ a\ i)
\end{array}
$$

Since indexSet is bijective from a finite integer range into the values of the multiset, the only remaining component for uniform sampling from sets is a function for uniform sampling from ranges. For this purpose, suppose we have a monad Random a with the only side-effect of generating random values, and the following function:

$$\mathsf{uniformRange} :: (\mathsf{Integer}, \mathsf{Integer}) \to \mathsf{Random\ Integer}$$

Computing uniformRange (lo, hi) returns a uniformly random integer in the inclusive interval (lo, hi). On top of this we can build the following procedure for uniform sampling from finite sets:

$$
\begin{aligned}
&\mathsf{uniformSet} :: \mathsf{Set\ a} \to \mathsf{Random\ a} \\
&\mathsf{uniformSet\ s} \mid |s| == 0 = \mathsf{error}\ \texttt{"empty set"} \\
&\qquad\qquad \mid \mathsf{otherwise} = \mathbf{do} \\
&\qquad\qquad\quad i \leftarrow \mathsf{uniformRange}\ (0, |s| - 1) \\
&\qquad\qquad\quad \mathsf{return}\ (\mathsf{indexSet\ s\ i})
\end{aligned}
$$

With these definitions at hand, all we have to do to uniformly sample values of size k from a space is to define a function sized which extracts the finite set of values of a given size.

$$
\begin{array}{lll}
\mathsf{sized} :: \mathsf{Space\ a} \to \mathsf{Int} \to \mathsf{Set\ a} & & \\
\mathsf{sized}\ \mathsf{Empty} & k & = \{\} \\
\mathsf{sized}\ (\mathsf{Pure\ a}) & 0 & = \{a\} \\
\mathsf{sized}\ (\mathsf{Pure\ a}) & k & = \{\} \\
\mathsf{sized}\ (\mathsf{Pay\ a}) & 0 & = \{\} \\
\mathsf{sized}\ (\mathsf{Pay\ a}) & k & = \mathsf{sized\ a}\ (k - 1) \\
\mathsf{sized}\ (a\ :+:\ b) & k & = \mathsf{sized\ a\ k} \uplus \mathsf{sized\ b\ k} \\
\mathsf{sized}\ (f\ :\$:\ a) & k & = \mathsf{fmap\ f}\ (\mathsf{sized\ a\ k})
\end{array}
$$

We define sized Pure to be empty for all sizes except 0, since we want values of an exact size. For Pay we get the values of size $k - 1$ in the

underlying space. Union and function application translate directly to union and application on sets. Selecting k-sized values of a product space requires dividing the size between the components of the resulting pair. Thus, we can consider the set as a disjoint union of the $k + 1$ different ways of dividing size between the components:

$$\mathsf{sized}\ (\mathsf{a}\ :*:\ \mathsf{b})\ \mathsf{k} = \biguplus_{k_1+k_2=k} \mathsf{sized}\ \mathsf{a}\ \mathsf{k}_1 \times \mathsf{sized}\ \mathsf{b}\ \mathsf{k}_2$$

Knowing how to sample finite sets, we can implement a sampling procedure on spaces by composing the sized function with the uniformSet function.

```
uniformSized :: Space a → Int → Random a
uniformSized s k = uniformSet (sized s k)
```

Computing cardinalities (and indexing) requires arbitrarily large integers, which are provided by Haskell's Integer type. Calculating cardinalities can be computationally expensive, and practical use requires memoising cardinalities of recursive data types, which is implemented using an additional constructor of the Space a data type not shown here.

3 Predicate-Guided Uniform Sampling

Having solved the problem of generating members of algebraic data types, we now extend the problem with a predicate that all generated values must satisfy.

A first approach for uniform generation is to use a simple form of *rejection sampling*. To generate a value that satisfies p :: a → Bool of a desired size, we simply generate values until we find one:

```
uniformFilter :: (a → Bool) → Space a → Int → Random a
uniformFilter p s k = do
  a ← uniformSized s k
  if p a then return a
         else uniformFilter p s k
```

The procedure returns values of a given size from the space that satisfy the predicate with a uniform distribution over the occurrences of these values in the space, as formally stated in Section 4.

This approach works well for cases where the proportion of values that satisfy the predicate is large enough, but it is far too inefficient in many practical situations (and if there are no values that satisfy the predicate it will never terminate).

In order to speed up random generation of values satisfying a given predicate, we propose another sampling procedure that uses the lazy behaviour

of the predicate to know its result on sets of values, rather than individual values, similarly to Runciman, Naylor, and Lindblad, 2008, Runciman *et al.*, (2008). For instance, consider a predicate ordered that tests if a list is sorted by comparing each pair of consecutive elements, starting from the front.

```
ordered :: Ord a ⇒ [a] → Bool
ordered []         = True
ordered [x]        = True
ordered (x : y : xs) = x <= y  &&  ordered (y : xs)
```

Applying the predicate to $1:2:1:3:5:[\,]$ will yield False after the pair $(2,1)$ is encountered, before the predicate inspects the later elements, thanks to the short-ciruiting && operator. This means that ordered is False for all lists starting with $1,2,1$. Once we have computed a set of values for which the predicate is going to return false, we remove all of these values from our original Space.

To detect this we exploit Haskell's call-by-need semantics by applying the predicate to a partially-defined value. In this case, observing that our predicate returns False when applied to a partially-defined list $1:2:1:\bot$, implies that the undefined part ($\bot$) can be replaced with any value without affecting the result. Thus, we could remove all lists that start with $1,2,1$ from the space. For many realistic predicates this removes a large number of values with each failed generation attempt, improving the chances of finding a value satisfying the predicate next time.

We implement this using the function universal, that determines if a given predicate is universally true, universally false, or if it (potentially) depends on its argument. The function universal returns Nothing if the predicate need to inspects its argument to yield a result, and Just True if the predicate is universally true and Just False if is universally false.

```
universal :: (a → Bool) → Maybe Bool
```

For example universal $(\lambda a \to$ True$)$ = Just True, universal $(\lambda a \to$ False$)$ = Just False, universal $(\lambda x \to x + 1 > x)$ = Nothing. Implementing universal involves applying the predicate to $\bot$ and catching the resulting exception if there is one. Catching the exception is an impure operation in Haskell, so the function universal is also impure (specifically, it breaks monotonicity).

The function universal is used to implement a new sized function: sizedP, which takes a predicate as a parameter along with a space and a size. Where sized resulted in just a set of values (Set a), sampling a value from the result of sizedP will either give a value that satisfies the predicate or an updated space that excludes a non-zero number of values that falsify the predicate.

```
sizedP :: (a → Bool) → Space a → Int → Set (Either a (Space a))
```

The intention of sizedP is best explained by implementing a sampling procedure that uses it. The sampling procedure picks a random value from the resulting set, if it is Left x, x satisfies the predicate and the procedure terminates, otherwise it recursively searches the new smaller space for a satisfying value:

```
uniform :: (a → Bool) → Space a → Int → Random a
uniform p s k = do
  x ← uniformSet (sizedP p s k)
  case x of Left  a  → return a
            Right s' → uniform p s' k
```

A complete definition of sizedP can be found in Figure 3.2. The function is implemented by recursion on its Space a argument, in a similar way to sized from Section 2. One difference is that it also reconstructs the updated space for the values for which the predicate fails. As the recursion proceeds it composes the predicate with constructor functions (from the :$: constructor). The cases for Pay, sum (:+:) and Empty follow the pattern set by the definition of sized. The case for Pay decreases the size parameter for the recursive call, and applies Pay to all residual spaces returned by it. The case for (:+:) returns a sum of sets returned by the recursive calls, and similarly applies (:+: b) or (a :+:) to the residual spaces contained in them. The Empty space is handled by the default case returning the empty set $\{\}$.

For function applications (:$:), sizedP constructs a new predicate p' by composing the input predicate with the applied function. If the new predicate is universally false (universal p' returns Just False) then sizedP returns an empty space. This reflects the fact that for a universally false predicate, there can be no values in the space that satisfy it. If the predicate is universally true or unknown, sizedP is called recursively on the underlying space with the updated predicate (this could be optimised to use sized in the universally true case). The apply function applies the function f to either the returned space or the returned value.

A key point of sizedP is that the cardinality of the resulting set does not depend on the predicate. In fact it is always the case that |sizedP p s k| = |sized s k|, in other words the result of sizedP contains one element for every value of size k regardless of how many of them that satisfy p. This allows efficient computations of cardinalities through memoisation. To accommodate this in the definition of sizedP we extend our multiset type with a new construct replicateSet :: Integer → a → Set a that adds a given number of occurrences of a value to the multiset (similar to the Haskell replicate function on lists). It is specified as follows:

```
indexSet (replicateSet n a) i = a
|replicateSet n a|            = n
```

```
sizedP :: (a → Bool) → Space a → Int → Set (Either a (Space a))
sizedP p (f :$: a) k = case universal p' of
  Just False → replicateSet |sized a k| (Right Empty)
  _          → fmap (apply f) (sizedP p' a k)
  where p' = p ∘ f
        apply f x = case x of
          Left x  → Left (f x)
          Right a → Right (f :$: a)
sizedP p (a :*: b) k = if inspectsFst p
    then sizedP p (swap :$: (b *** a)) k
    else sizedP p (a *** b) k
    where swap (a,b) = (b,a)
sizedP p (a :+: b) k = rebuild (:+: b) (sizedP p a k) ⊎
                       rebuild (a :+:) (sizedP p b k)
  where
    rebuild :: (Space a → Space a) →
               Set (Either a (Space a)) →
               Set (Either a (Space a))
    rebuild f s = fmap (fmap f) s
sizedP p (Pay a)  k
  | k > 0           = fmap (fmap Pay) $ sizedP p a (k - 1)
sizedP p (Pure a) 0
  | p a             = {Left a}
  | otherwise       = {Right Empty}
sizedP _ _        _ = {}
```

Figure 3.2: Definition of a predicate guided sized-function.

The final case to discuss is the one for $(:*:)$, which is a little more involved, as we need to decide which component of the pair to refine. The following subsection describes how to make this choice based on the order of evaluation of the predicate.

3.1 Predicate-Guided Refinement Order

When implementing sizedP for products, it is no longer possible to divide the size between the components, as was done in the implementation of sized (see Section 2). The reason is that it is not possible to split a predicate on pairs into two independent predicates for the first and second components.

We solve this problem using the algebraic nature of our spaces to eliminate products altogether. We can use the following algebraic laws to eliminate products:

$$
\begin{array}{lll}
a \otimes (b \oplus c) \equiv (a \otimes b) \oplus (a \otimes c) & & \textit{[distributivity]} \\
a \otimes (b \otimes c) \equiv (a \otimes b) \otimes c & & \textit{[associativity]} \\
a \otimes 1 \equiv a & & \textit{[identity]} \\
a \otimes 0 \equiv 0 & & \textit{[annihilation]}
\end{array}
$$

Expressing these rules on our Haskell data type is more complicated, because we need to preserve the types of the result, i.e. we only have associativity of products if we provide a function that transforms the left associative pair back to a right associative one, etc. The equalities above can be used to define an operator $(***)$ on spaces that pushes top level products inwards without loss of information:

$$
\begin{array}{lll}
a *** (b :+: c) = (a :*: b) :+: (a :*: c) & & \textit{[distributivity]} \\
a *** (b :*: c) = & & \\
\quad (\lambda((x,y),z) \to (x,(y,z))) :\$: ((a :*: b) :*: c) & & \textit{[associativity]} \\
a *** (\mathsf{Pure}\ x) = (\lambda y \to (y,x)) :\$: a & & \textit{[identity]} \\
a *** \mathsf{Empty} = \mathsf{Empty} & & \textit{[annihilation]}
\end{array}
$$

Additionally, we need two cases for lifting Pay and function application.

$$
\begin{array}{lll}
a *** (\mathsf{Pay}\ b) = \mathsf{Pay}\ (a :*: b) & & \textit{[lift-pay]} \\
a *** (f :\$: b) = (\lambda(x,y) \to (x, f\ y)) :\$: (a :*: b) & & \textit{[lift-fmap]}
\end{array}
$$

The first law states that paying for the component of a pair is the same as paying for the pair, the second that applying a function f to one component of a pair is the same as applying a modified (lifted) function on the pair. If recursion is always guarded by a Pay, we know that the transformation will terminate after a bounded number of steps.

Using these laws we could define sizedP on products by applying the transformation, so $\mathsf{sizedP}\ p\ (a :*: b) = \mathsf{sizedP}\ p\ (a *** b)$. This is problematic,

because $(***)$ imposes a right-first order of evaluation, which means that for our generators the first component of a pair is never generated before the right one is fully defined. This is detrimental to performance, since the predicate may not require the right operand to be defined at all. In the end this would mean that when the predicate is falsified sizedP would not remove as many values from the space as it potentially could.

To change this, and guide the refinement order by the evaluation order of the predicate, we need to 'ask' the predicate which component should be defined first. We define a function similar to universal that takes a predicate on pairs:

```
inspectsFst :: ((a, b) → Bool) → Bool
```

The expression inspectsFst p is True iff p evaluates the first component of the pair before the second. Just like universal, inspectsFst exposes some information of the Haskell runtime, which cannot be observed directly.

Thus, to define the final, product $(:*:)$ case of sizedP in Figure 3.2 we combine inspectsFst with another algebraic law: Commutativity of products. If the predicate 'pulls' at the first component, the operands of the product are swapped before applying the transformation for the recursive call.

The end result is an algorithm that gradually refines a value, by expanding only the part of the space that the predicate needs in order to progress. With every refinement, the space is narrowed down until the predicate is guaranteed to be false for all values in the space, or until a single value satisfying the predicate is found.

3.2 Example

To further illustrate how our algorithm operates we provide an example of using it to find lambda terms of a given type and size. The example is similar to that in the introduction but slightly simplified for ease of presentation. For the purpose of this example, we define a simple data type representing lambda terms with De Bruin indices.

```
data Term = Ap (Term, Term) | Lam Term | Var ℕ
```

We are not required to 'uncurry' the Ap constructor, but it helps the presentation of the example. The space of all lambda terms is defined as follows:

```
spTerm, spAp, spLam, spVar :: Space Term
spTerm = Pay (spAp :+: spLam :+: spVar)
spAp   = Ap  :$: (spTerm :*: spTerm)
spLam  = Lam :$: spTerm
spVar  = Var :$: spNat
```

Furthermore, this example assumes we have a type checking function, which decides whether a lambda term is well scoped and of a given type.

```
data Type = TInt | Type ↦ Type
check :: Type → Term → Bool
```

For instance, check (TInt ↦ TInt) (Lam (Var Z)) evaluates to True whereas check TInt (Lam (Var Z)) is False. The type checker is lazy so for instance check TInt (Lam ⊥) is False. We deliberately omit the implementation of the type checker, and treat it as a black box: We can only observe its behaviour by executing it, and using the functions universal and inspectsFst.

We will not directly step through the execution of the uniform function described earlier in this section. Instead we show the algorithm as a series of transformations on spaces, which illustrate how indexing is performed.

The first step in every iteration of the algorithm is to transform the space we are currently working with into this normal form:

$$\mathsf{Pay}^n\ (\mathsf{f}\ \mathbin{:\$:}\ (\mathsf{s}_1\ \mathbin{:+:}\ \ldots\ \mathbin{:+:}\ \mathsf{s}_k))$$

Here, Pay^n represents n applications of the Pay constructor. The general idea is that the partial value f ⊥ is the result of all the choices we have made up to this point, and the sum is the next choice we would have to make in order to further define the value. To determine if further choices are necessary to decide predicate p, we use the universal function from Section 3. If universal (p ∘ f) is Nothing the space must be further refined before the predicate yields a result. This is done by choosing one of the summands s_i of the sum, weighted by the cardinalities to guarantee uniformity. If the result of universal is Just b we know that the result of p is b for all values in the space.

Any space can be transformed into this form provided that it contains at least one sum. We call this transformation *lifting choices*. It involves applying the distributivity, associativity, identity and annihilation laws of $:*:$ mentioned earlier in this section, along with the following laws:

$$\begin{aligned}
&\mathsf{f}\ \mathbin{:\$:}\ \mathsf{Pay\ s} &&\equiv \mathsf{Pay}\ (\mathsf{f}\ \mathbin{:\$:}\ \mathsf{s})\\
&\mathsf{f}\ \mathbin{:\$:}\ (\mathsf{g}\ \mathbin{:\$:}\ \mathsf{s}) &&\equiv (\mathsf{f} \circ \mathsf{g})\ \mathbin{:\$:}\ \mathsf{s}
\end{aligned}$$

To demonstrate the execution of the algorithm, we consider generating well-typed terms of type Int → Int of size 11, for which we will use the space spTerm and the predicate p = typeCheck (Int ↦ Int). The number of lambda terms of size 11 can be computed from the definition of the space (see Section 2), and for our space it is c = 465.

To generate a random well-typed term our uniformSet function chooses a candidate index between 0 and c − 1 uniformly at random—let's say that it chose 407. The first iteration of the algorithm starts with the space

spTerm. Transforming it to normal form only requires applying the identity function inside of the Pay constructor.

$$\mathsf{Pay}\ (\mathsf{id}\ \mathsf{:\$:}\ (\mathsf{spAp}\ \mathsf{:+:}\ \mathsf{spLam}\ \mathsf{:+:}\ \mathsf{spVar}))$$

Next we use universal to check if the type checker p cares at all about which value we use. In this case universal $(\mathsf{p} \circ \mathsf{id})$ is Nothing, so we need to define at least some part of the term to know if the predicate holds or not.

The space needs to be refined by committing to one of the three summands in spAp :+: spLam :+: spVar. The refined spaces corresponding to the choices are as follows:

$$\mathsf{Pay}\ (\mathsf{id}\ \mathsf{:\$:}\ \mathsf{spAp})$$
$$\mathsf{Pay}\ (\mathsf{id}\ \mathsf{:\$:}\ \mathsf{spLam})$$
$$\mathsf{Pay}\ (\mathsf{id}\ \mathsf{:\$:}\ \mathsf{spVar})$$

The choice is made based on the candidate index selected earlier, and the cardinalities of the refined spaces we may choose. For size 11, the cardinalities are 257, 207 and 1 respectively. Since $257 \leqslant 407 \leqslant 257 + 207$, we choose the second space, and obtain a residual index of $407 - 257 = 150$. Expanding the definition of spLam, lifting the choice it contains, and eliminating a composition with id we get:

$$\mathsf{Pay}^2\ (\mathsf{Lam}\ \mathsf{:\$:}\ (\mathsf{spAp}\ \mathsf{:+:}\ \mathsf{spLam}\ \mathsf{:+:}\ \mathsf{spVar}))$$

Next, universal $(\mathsf{p} \circ \mathsf{Lam})$ evaluates to Nothing, indicating that the space needs to be further refined. The refinement is selected by the same procedure as in the previous iteration, now using the residual index from the last choice (150) instead of a random index. Lambda is chosen again yielding $\mathsf{Pay}^2\ (\mathsf{Lam}\ \mathsf{:\$:}\ \mathsf{spLam})$ as our refined space, in normal form it is:

$$\mathsf{Pay}^3\ (\mathsf{Lam} \circ \mathsf{Lam}\ \mathsf{:\$:}\ (\mathsf{spAp}\ \mathsf{:+:}\ \mathsf{spLam}\ \mathsf{:+:}\ \mathsf{spVar}))$$

At this point we test universal $(\mathsf{p} \circ \mathsf{Lam} \circ \mathsf{Lam})$, and get Just False, which means that there are no lambda terms of type Int → Int with two head lambdas.

Now our only option is to discard the current space (the space of terms with two head lambdas), choose a new random index, and restart. To reconstruct the space of remaining terms we reiterate through the choices we have made to get to this point, and build a space from the sum of all the options we did not choose (see the definition of the :+: case in Figure 3.2). In this case we made two choices, and each had two alternative options. Thus the space we construct is this:

$$\mathsf{Pay}\ (\mathsf{spAp}\ \mathsf{:+:}\ \mathsf{spVar})\ \mathsf{:+:}\ \mathsf{Pay}^2\ (\mathsf{Lam}\ \mathsf{:\$:}\ (\mathsf{spAp}\ \mathsf{:+:}\ \mathsf{spVar}))$$

Analysing the space we see that it excludes only the values that start with two lambdas, because we have four choices that represent starting with Ap or Var or starting with Lam followed by Ap or Var.

The cardinality of this new space is 371, which means that the next candidate index must be in range from 0 to 370. Let's say that this time we choose an index, which leads to Pay spAp being selected as the refined space. Expanding the definition of spAp leads to Pay (Ap :$: (spTerm :∗: spTerm)), which cannot be trivially transformed into a normal form. A choice needs to be lifted up from one of the operands of the product. This is done by applying either left or right distributivity depending on which component of the pair is requested by the predicate. To determine which component is required we evaluate inspectsFst (p ∘ Ap). In our example inspectsFst returned True, so we know that the predicate decided to evaluate the parameter of the function application first. This means that we get our new space from the ∗∗∗ transformation defined earlier (which expands its right operand) and get the following sum:

```
Pay (Ap :$: (spTerm ∗∗∗ spTerm)) ≡
  Pay² (Ap :$:    (   (spTerm :∗: spAp)
                  :+: (spTerm :∗: spLam)
                  :+: (spTerm :∗: spVar)))
```

At this point, further iterations with our particular index will lead to a space with no further choices, whose normal form is a function application on a unit. Since there is only one value in the space in this special case we can simply construct it and type check it. One possible value is Ap (Lam (Var Z)) (Ap (Lam (Var Z)) (Lam (Var Z))), or $(\lambda x \to x)\ ((\lambda x \to x)\ (\lambda x \to x))$ in a more readable syntax. As the predicate p answers True for it, it is a term of the type we were looking for.

4 Uniformity of the Generators

It is easy to prove that uniformSet s is uniform over the occurrences in s, by proving that indexSet s is bijective and that uniformRange (lo, hi) is uniform over the inclusive interval (lo, hi). Many of the subsequent proofs in this section rely on the Set type being an accurate representation of multisets, so properties like commutativity and distributivity of union on Set follow from the corresponding theorems for multisets. These properties should be straightforward to prove by providing bijections between the indexes of indexSet.

From these preliminaries we can prove that the rejection sampler uniformFilter p s k (described in Section 3) provides uniform sampling of values of size k from the space s, constrained by the predicate p.

Theorem 1. Consider space s, a non-negative size k, and a predicate p. Let the multiset a = sized s k and $b = \{x \mid x \in a, p\ x\}$. If b is non-empty, then uniformFilter p s k is uniform over b.

Proof. Let $\mathsf{n} = |\mathsf{a}|$, $\mathsf{m} = |\mathsf{b}|$ and x be an occurrence in b. We will calculate the probability of $\mathsf{x} = \mathsf{uniformFilter\ p\ s\ k}$. In every iteration of uniformFilter, a uniformly random value y is drawn from a. The probability that $\mathsf{y} \notin \mathsf{b}$ (causing the procedure to retry) is $(\mathsf{n} - \mathsf{m})\ /\ \mathsf{n}$, whereas the probability that $\mathsf{y} = \mathsf{x}$ is $1\ /\ \mathsf{n}$. Thus, the probability of x being drawn during the i-th iteration is equal to $(1/\mathsf{n})((\mathsf{n} - \mathsf{m})/\mathsf{n})^{i-1}$. Finally, the probability of x being drawn after any number of retries becomes the limit of the geometric series $(1/\mathsf{n}) \sum_{i=0}^{\infty}((\mathsf{n} - \mathsf{m})/\mathsf{n})^i = 1/\mathsf{m}$. □

Proving uniformity of the predicate-guided uniform function is more difficult, as the mapping from indexes to values depends on the evaluation order of the predicate. Particularly the space is transformed differently depending on which component of a pair is inspected first. The two possible resulting spaces will contain the same values but in different internal ordering. However, if we assume that the evaluation order of the predicate is deterministic, this also determines a fixed order of elements in the space, which defines an injective mapping.

First we need a lemma that the $(***)$ operator preserves equality of spaces if we disregard the order of elements returned by indexing of multisets.

Lemma 1. Let $\mathsf{a} :: \mathsf{Space\ a}$ and $\mathsf{b} :: \mathsf{Space\ b}$ be two spaces, and $\mathsf{k} :: \mathsf{Int}$ a non-negative integer. Then, the following equivalence holds between multisets:

$$\mathsf{sized\ (a\ :*:\ b)\ k} = \mathsf{sized\ (a} *** \mathsf{b)\ k}$$

Proof sketch. A straightforward proof by structural induction on b is possible. Each of the cases can be proven using the corresponding laws for multisets and Pay. □

The next lemma contains most of the complexity of the proof. It shows that values of the form $\mathsf{Left\ x}$ in the multiset returned by $\mathsf{sizedP\ p\ s\ k}$ contains exactly the subset of values returned by $\mathsf{sized\ s\ k}$ that satisfy p.

Lemma 2. Let $\mathsf{s} :: \mathsf{Space\ a}$ be a space, $\mathsf{p} :: \mathsf{a} \to \mathsf{Bool}$ a Boolean predicate, and $\mathsf{k} :: \mathsf{Int}$ a non-negative integer. Then, the following equivalence holds between multisets:

$$\{\mathsf{x} \mid \mathsf{Left\ x} \in \mathsf{sizedP\ p\ s\ k}\} = \{\mathsf{x} \mid \mathsf{x} \in \mathsf{sized\ s\ k}, \mathsf{p\ x}\}$$

Proof sketch. The proof is carried out by induction and case analysis of sizedP. The most interesting part of the proof are cases for $\mathsf{a\ :*:\ b}$ and $\mathsf{f\ :\$:\ a}$. We first show the sketch for $\mathsf{a\ :*:\ b}$, which requires proving the following equation.

$$\{\mathsf{x} \mid \mathsf{Left\ x} \in \mathsf{sizedP\ p\ (a\ :*:\ b)\ k}\} = \{\mathsf{x} \mid \mathsf{x} \in \mathsf{sized\ (a\ :*:\ b)\ k}, \mathsf{p\ x}\}$$

Out of the two subcases we will only show the more complex case when inspectsFst p, using equational reasoning. Note that we only consider the recursive case of the second sizedP invocation here. It can be shown that the other case, when universal p′ = Just False, results in both sides of the equation being equal to {}.

```
{x | Left x ∈ sizedP p (a :∗: b) k}
 {-Definition of sizedP -}
{x | Left x ∈ sizedP p (swap :$: (b ∗∗∗ a)) k}
 {-Definition of sizedP (recursive case) -}
{x | Left x ∈ {apply swap y | y ∈ sizedP (p ∘ swap) (b ∗∗∗ a) k}}
 {-Simplification -}
{swap x | Left x ∈ sizedP (p ∘ swap) (b ∗∗∗ a) k}
 {-Induction hypothesis -}
{swap x | x ∈ sized (b ∗∗∗ a) k, (p ∘ swap) x}
 {-From Lemma 1 -}
{swap x | x ∈ sized (b :∗: a) k, (p ∘ swap) x}
 {-Commutativity of space products -}
{x | x ∈ sized (a :∗: b) k, p x}
```

Notably, to be well-founded this inductive step requires an external convergence measure where b ∗∗∗ a is smaller than a :∗: b, which is a little intricate to construct. Without going into extreme detail, we present an outline of such a measure. The measure consists of three components (k, r, q), ordered lexicographically from left to right. The component k is the size parameter of sizedP. The measure r keeps track of how many steps the algorithm might need to take before the next Pay constructor is consumed, and corresponds, more less, to the longest path without a Pay constructor. The measure q keeps track of the number of times that ∗∗∗ needs to be applied to remove the :∗: from the top level of the space. The measure q depends on the predicate p, and for a space that is a tree of nested products, q is the position (in a breadth first order) of the leaf that is requested by the evaluation of the predicate, as indicated by repeated application of inspectsFst.

With this definition it can be proven that for a given invocation of sizedP with measure $m = (k, r, q)$, the measure for a recursive call performed by it is strictly smaller than m. For the :∗: case above, the measure k for the recursive call is the same as for the original call, but either r or q is always decreased.

One complication in the proof is that fact that a ∗∗∗ b might add an extra :$: constructor to the result space, which is immediately removed by the next invocation of sizedP. The convergence measure needs to account for this fact.

Furthermore, the convergence measure requires the evaluation of the predicate to be deterministic, as otherwise the repeated application of the ∗∗∗

operator might lead to an infinite loop. Specifically, inspectsFst is required to interact in a standard way with functions like swap, for example satisfying the law inspectsFst p ⇒ *not* (inspectsFst (p ∘ swap)).

The next case of sizedP that we will consider is the f :$: a case, which requires proving the following equation.

$$\{x \mid \text{Left } x \in \text{sizedP } p\ (f :\$: a)\ k\} = \{x \mid x \in \text{sized } (f :\$: a)\ k, p\ x\}$$

The subcase when universal (p ∘ f) = Just False yields both sides of the equation to be equal to { }. The recursive subcase can be proven using equational reasoning.

$$\begin{aligned}
&\{x \mid \text{Left } x \in \text{sizedP } p\ (f :\$: a)\ k\}\\
&\quad \text{\{-Definition of sizedP -\}}\\
&\{x \mid \text{Left } x \in \{\text{apply } f\ y \mid y \in \text{sizedP } (p \circ f)\ a\ k\}\}\\
&\quad \text{\{-Simplification -\}}\\
&\{f\ x \mid \text{Left } x \in \text{sizedP } (p \circ f)\ a\ k\}\\
&\quad \text{\{-Induction hypothesis -\}}\\
&\{f\ x \mid x \in \text{sized } a\ k, (p \circ f)\ x\}\\
&\quad \text{\{-Simplification -\}}\\
&\{x \mid x \in \{f\ y \mid y \in \text{sized } a\ k\}, p\ x\}\\
&\quad \text{\{-Definition of sized -\}}\\
&\{x \mid x \in \text{sized } (f :\$: a)\ k, p\ x\}
\end{aligned}$$

The recursive call in this case has the same measure k as the original call, but measure r is reduced by one.

The remaining cases can be proved in a similar way.

□

If indexing in the result of sizedP hits an element that does not satisfy the predicate, the result is Right s, where s is the residual space, which is used by uniform to continue sampling. We show that the spaces returned by sizedP retain all elements from the original space that satisfy the predicate, and are strictly smaller than the original space.

Lemma 3. Let s be a space, p a Boolean predicate, k a non-negative integer, and Right $s' \in$ sizedP p s k. Then the following equation between multisets holds.

$$\{x \mid x \in \text{sized } s\ k, p\ x\} = \{x \mid x \in \text{sized } s'\ k, p\ x\}$$

Furthermore, sized s' k is a proper subset of sized s k.

The proof of the above lemma is by induction and case analysis on sizedP, similarly to the proof of Lemma 2.

The final theorem shows the uniformity of the distribution of elements returned by uniform.

Theorem 2. Consider space s, a non-negative size k, and a predicate p. Let the multiset a = sized s k and b = $\{x \mid x \in a, p\ x\}$. If b is non-empty, then uniform p s k is uniform over b.

Proof. Let n = |a|, m = |b| and x be an occurrence in b. We will show that the probability of x = uniform p s k is equal $1/m$, by performing induction on n. When uniform p s k executes, first sizedP p s k is used to construct a multiset of indexing results, then one element of this set is selected uniformly at random using uniformSet. For the base case of the induction, we take that n = m. Then, by Lemma 2, all occurrences from the multiset are of the form Left y, and exactly 1 of them is the occurrence of Left x. Thus, the probability of generating x is $1/m$.

For the induction step, assume that n > m. From Lemma 2, Left x is an occurrence in sizedP p s k corresponding to x. Moreover, $n - m$ elements are of the form Right s′, since |sizedP p s k| = |sized s k|. Thus, the probability of generating x directly is $1/n$, whereas the probability of retrying is $(n - m)/n$. The uniform procedure will retry with a modified space s′. From Lemma 3, sized s′ k contains the same elements that satisfy p as sized s k, but fewer elements that do not satisfy p. We can now invoke the induction hypothesis, which gives that the probability of generating x by uniform p s′ k is $1/m$. Thus, the overall probability of generating x is $1/n + ((n - m)/n)(1/m) = 1/m$ □

Non-deterministic predicates The proof above assumes that the evaluation order of the predicate is deterministic. There are two reasons for this. Firstly, there is a risk of non-termination when the order of evaluation of the predicate is not consistent between different invocations of inspectsFst. This problem can be addressed by introducing a generalised inspectsFst that returns the index in a nesting of pairs that a predicate inspects, and a generalised associativity transformation.

Even so, there is a more subtle problem with non-deterministic evaluation order—it may cause biased distribution of the generated data. For example, consider a space containing the four total values of type (Bool, Bool), and a predicate with non-deterministic evaluation order defined on this type.

```
spaceBool :: Space Bool
spaceBool = Pay (Pure False :+: Pure True)

spacePairB :: Space (Bool, Bool)
spacePairB = Pay ((,) :$: spaceBool :*: spaceBool)

oracle :: IO Bool

nonDetB :: (Bool, Bool) → Bool
nonDetB (a, b) = unsafePerformIO $ do
  x ← oracle
```

```
if x then a `pseq` b `pseq` True
     else b `pseq` a `pseq` True
```

The predicate has access to non-deterministic oracle returning a Boolean value. If the oracle returns True, the predicate evaluates the first component of the pair using pseq, then the second one before returning True. Otherwise, it evaluates the pair's components in the opposite order.

Depending on the value returned by oracle, indexing values of size 3 in spacePairB with the predicate nonDetB yields values in one of two orders.

oracle result	Index: 0	1	2	3
True	(False, False)	(False, True)	(True, False)	(True, True)
False	(False, False)	(True, False)	(False, True)	(True, True)

Now suppose that oracle contains a race condition, which is triggered by events in the part of the program that selects the randomly chosen index, and results in the following: If the index is 1 then oracle returns False, otherwise it returns True. Then, indexing using nonDetB will yield (True, False) for indices 1 and 2, whereas (False, True) will never be returned, leading to a biased distribution.

Although it is hard to implement oracle so it exhibits this behaviour making this particular example largely hypothetical, it highlights the risks of allowing evaluation order to influence semantics. In Section 5.4 we discuss an alternative algorithm with deterministic indexing. For our main algorithm this example demonstrates the need for assuming deterministic evaluation order, or possibly a weakened assumption that the evaluation order is independent from the choice of index.

5 Efficient Implementation and Alternative Algorithms

The previous sections give a high level description of our algorithm. Implementing it required making a number of engineering choices, some of which had a considerable effect on the performance of the generator.

This section we describes some of these choices that we found most important for performance, and also experiments with alternative versions of the core algorithm aimed at improving performance.

5.1 Relaxed Uniformity Constraint

When our uniform generator encounters a subspace for which the predicate is false, the algorithm must retry with a new random index in the

reconstructed set. The new index must be chosen independently from the old in order to achieve uniform distribution. We have implemented two alternative algorithms that violate this restriction, compromising uniformity, in favour of better performance.

The first one is to backtrack and try the alternative in the most recent choice. Such generators are no longer uniform, but potentially more efficient. Even though the algorithm start searching at a uniformly chosen index, since an arbitrary number of backtracking steps is allowed the distribution of generated values may be arbitrarily skewed. In particular, values satisfying the predicate that are 'surrounded' by many values for which it does not hold may be much more likely to be generated than other values.

The second algorithm also performs backtracking, but imposes a bound b for how many values the backtracking search is allowed to skip over. When the bound b is reached, a new random index is generated and the search is restarted. The result is an algorithm which has an 'almost uniform' distribution in a precise way: The probabilities of generating any two values differ at most by a factor $b + 1$. So, if we pick $b = 1000$, generating the most likely value is at most 1001 times more likely than the least likely value.

The bounded backtracking search strategy generalises both the uniform search (when the bound b is 0) and the unlimited backtracking search (when the bound b is infinite).

We expected the backtracking strategy to be more efficient in terms of time and space usage than the uniform search, and the bounded backtracking strategy to be somewhere in between, with higher bounds leading to results closer to unlimited backtracking. Our intention for developing these alternative algorithms was that trading the uniformity of the distribution for higher performance may lead to a higher rate of finding bugs. Section 6 contains experimental verification of these hypotheses.

5.2 Parallel Conjunction

It is possible to improve the generation performance by introducing the parallel conjunction operator (Runciman, Naylor, and Lindblad, 2008), which makes pruning the search space more efficient. Suppose we have a predicate `p x = q x && r x`. Given that `&&` is left-biased, if `universal r` is `Just False` and `universal q` is `Nothing` then the result of `universal p` will be `Nothing`, even though we expect that refining `q` will make the conjunction return `False` regardless of what `q x` returns.

We can define a new operator `&&&` for parallel conjunction with different behaviour when the first operand is undefined: $\bot$ `&&& False = False`. This may make the `sizedP` function terminate earlier when the second operand of a conjunction is false, without needing to perform refinements needed

by the first operand at all. Similarly, we define parallel disjunction that is True when either operand is True.

Note that the parallel conjunction and disjunction still have the same evaluation order as their normal counterparts, that is when both operands are undefined, the left one is evaluated first.

5.3 Sharing in the representation of spaces

The implementation of the algorithm described in Sections 2 and 3 starts with a compact representation of the whole search space, where recursive references to the space are shared. The representation is subsequently expanded, and subspaces are created from it as a result of refinement.

We found it important to ensure that as much sharing as possible is achieved between the representations of subspaces in order to save memory, and share the results of the cardinality computations. The measures used to increase sharing included folding subspaces that have no choices left in them into single units, and rebalancing the tree representation of spaces.

Increasing sharing turned memory-bound computations into CPU-bound ones, while improving run time performance at the same time. As a result, most benchmarks that we ran, including the ones presented in Section 6 were limited by the run time rather than the available memory.

5.4 Deterministic Indexing

As demonstrated in Section 4, allowing evaluation order to influence indexing is potentially problematic if the evaluation order is non-deterministic, for instance in parallel computations.

To address this problem, we propose making the mapping of indices to values in the space independent of the predicate, restricting non-determinism to affect only which falsifying values are removed in an iteration of the top level algorithm. This requires significant modifications of the algorithm, which involve replacing sizedP with three distinct steps:

1. A deterministic indexing procedure produces a tree structure containing all indexing choices (left or right operands of a union) required to produce a random total value, without considering the predicate at all.
2. A non-deterministic procedure prunes this tree, keeping only the choices required to build a partial value for which the given predicate terminates.
3. A subtraction procedure removes all values resulting from the pruned tree from the space.

The tree structure that containing indexing choices can be defined by this data type:

```
data Select = This | Fst Select | Snd Select | Pair Select Select
```

The Fst and Snd constructors mark the selection of the first or the second component of a union respectively. The Pair constructor combines choices made in the components of a product space. With this data type the three steps above would correspond to functions of these types:

```
sizedDet :: Space a → Int → Set Select
pruneChoices :: (a → Bool) → Space a → Select → Select
subtractChoices :: Space a → Select → Space a
```

In addition, a fourth function is needed that returns the value identified by the choices.

```
selectedValue :: Space a → Select → a
```

Here we discuss implementation of these functions, which is not included in the paper. Function sizedDet is a simple adaptation of sized (Section 2.3). Function pruneChoices uses inspectsFst, universal and $***$ similarly to how they are used in sizedP (Figure 3.2) in order to prune the tree of choices. Function subtractChoices performs algebraic transformations to remove the subspace specified by the choice tree from the original space. Function selectedValue is implemented using structural recursion on both arguments.

Preliminary experimentation with this approach showed that implementing subtractChoices efficiently is key for efficiency. Several implementations of it were tried but none were as fast as the original algorithm. Further experimentation with this approach remains a topic of future work.

5.5 In-place Refinement

The algorithm described in Section 3 is implemented by applying the predicate to partial values and by throwing and catching exceptions, determine which part of the value needs to be further defined (this is the inner workings of inspectsFst and universal). This process might be computationally expensive as it requires repeated evaluation of the predicate.

As an alternative mechanism for observing the evaluation order of predicates we experimented with a variant of the algorithm that uses Haskell's lazy evaluation with fully-defined values. In this algorithm, the indexing function directly builds a random fully-defined value, and attaches a Haskell IO-action to each subcomponent of it. When the predicate is applied to the value, the IO-actions will fire only for the parts that needs to be inspected to determine the outcome. Whenever the indexing function is required to make a choice, the corresponding IO-action records the option

it took. After the predicate has terminated, the pruned choice tree can be constructed from the recorded trace.

This approach has the advantage of deterministic indexing, and reduces the maximal number of times the predicate is executed for each iteration of the algorithm from `n` to `1` (where `n` is the number of choices made, usually proportional to the size of the value). In particular, the exact value returned by the indexing function will not depend on the evaluation order of the predicate.

The algorithm based on in-place refinement can be summarised as follows:

1. Sample the space for a lazily-defined value uniformly at random. Inspecting any constructor of the sampled value makes a record in the trace.
2. Execute the predicate on the value.
3. If the result of the predicate is `True`, return the generated value, otherwise continue.
4. Determine which parts of the value were inspected by examining the trace. This information determines which choices had to be made by the indexing function, and which are redundant.
5. *Subtract* the space of all values from the previous point from the current space, and restart the algorithm.

Despite the clear advantage of not having to re-evaluate the predicate on many partial values for each falsifying total value, the generator based on this technique turned out to be slower than our original implementation for the predicates and spaces we used. On the other hand, this generator used less memory in most cases compared the original one.

In addition to the performance problems, defining parallel conjunction for this type of refinement is difficult because inspecting the result of a predicate irreversibly makes the choices required to compute the result. For these reasons our implementations of in-place refinement remains a separate branch of development and a topic of future work.

6 Experimental Evaluation

We evaluated our approach in four benchmarks. Three of them involved measuring the time and memory needed to generate 2000 random values of a given size satisfying a predicate. The fourth benchmark compared a derived simply-typed lambda term generator against a hand-written one in triggering strictness bugs in the GHC compiler. Some benchmarks were also run with a naïve generator that generates random values from a space, as in Section 2, and filters out those that do not satisfy a predicate.

6.1 Trees

Our first example is binary search trees (BSTs) with Peano-encoded natural numbers as their elements, defined as follows.

```
data Tree a = L
  | N a (Tree a) (Tree a)
isBST :: Ord a ⇒ Tree a → Bool
data ℕ = Z | Suc ℕ
instance Ord ℕ where
  _     < Z     = False
  Z     < Suc _ = True
  Suc x < Suc y = x < y
```

The isBST predicate decides if the tree is a BST, and uses a supplied lazy comparison function for type ℕ for increased laziness.

```
isBST :: Ord a ⇒ Tree a → Bool
isBST t = aux Nothing Nothing t where
  Nothing ⊴ y = True
  Just x  ⊴ y = x ⩽ y
  x ≼ Nothing = True
  x ≼ Just y  = x ⩽ y
  aux _ _ L = True
  aux lb ub (N x t1 t2) = lb ⊴ x && x ≼ ub
    && aux lb (Just x) t1 && aux (Just x) ub t2
```

The predicate's auxiliary function accepts two optional bounds and a subtree and decides whether the subtree is a BST with all elements within the bounds.

The benchmark involved measuring the time and space needed to generate 2000 trees for each size from a range of sizes, allowing at most 300 s of CPU time and 4 GiB of memory to be used. Derived generators based on three different search strategies (see Section 5.1) were used: One performing uniform sampling (*uniform*), one bounded backtracking allowed to skip at most 10k values (*backtracking 10k*), and one performing unbounded backtracking (*backtracking*). A naïve generate-and-filter generator was also used for comparison.

Both *backtracking 10k* and *backtracking* generators produce non-uniform distributions of values. The skew of the *backtracking 10k* generator is limited, as the least likely values are generated at most 10k times less likely than the most common ones, as mentioned in Section 5.1.

Figure 3.3 shows the time and memory consumed the runs with resource limits marked by dotted lines in the plots. Run times for all derived generators rise sharply with the increased size of generated values and seem

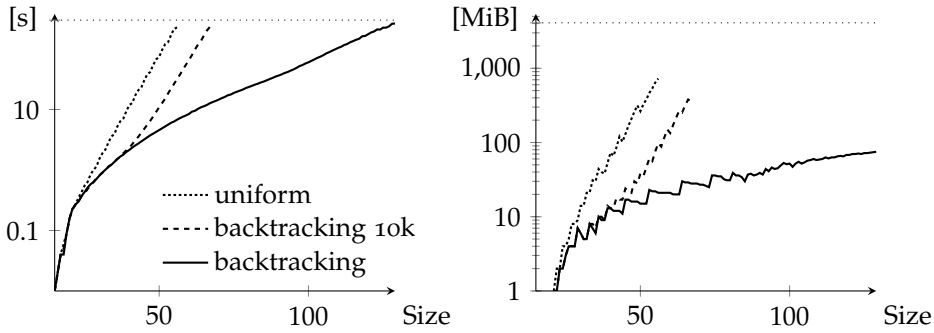


Figure 3.3: Run times in (left) and memory consumption (right) of derived generators generating 2000 BSTs depending on the size of generated values.

to approach exponential growth for larger sizes. The backtracking generator performs best of all, and has a slower exponential growth rate for large sizes than the other derived generators. The *backtracking 10k* generator achieved similar performance as the *uniform* one when generating values that are about 11 size units larger. The generate-and-filter generator was not able to complete any of the runs in time, and is omitted from the graphs.

6.2 Simply-typed Lambda Terms

Generating random simply-typed lambda terms was our motivating application. Simply-typed lambda terms can be turned into well-typed Haskell programs and used for testing compilers. Developing a hand-written recursive generator for them requires the use of backtracking, because of the inability of predicting whether a given local choice can lead to a successful generation, and because typing constraints from two distant parts of a term can cause conflict. Achieving satisfactory distribution and performance requires careful tuning, and it is difficult to assess if any important values are severely underrepresented (Pałka, 2012).

On the other hand, obtaining a generator that is based on our framework requires only the definitions from Figure 3.1, and a space definition. Defining the space of closed expressions spaceClosedExprs also requires auxiliary definitions of spaces containing open expressions.

```
spaceExprs :: [Space Expr]
spaceExprs = go 0 where
  go k = let go_k_1 = go (k + 1)
             go_k   = Pay  $  (Ap :$: go_k :*: go_k :*: spaceType)
                          :+: (Vr  :$: fin k, Lm :$: head go_k_1)
```

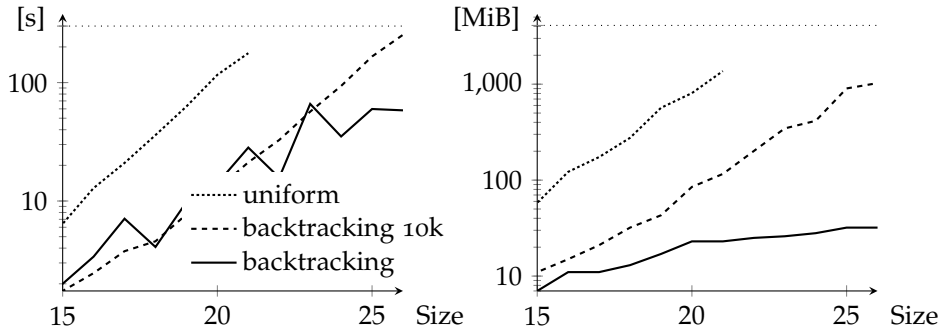


Figure 3.4: Run times (left) and memory consumption (right) of derived generators generating 2000 simply-typed lambda terms depending on the size of generated terms.

```
            in go_k : go_k_1
spaceClosedExprs = head spaceExprs
```

To ensure sharing, we define the top-level list spaceClosedExprs of spaces of expressions with 0, 1, 2, and so on free variables. The definition of the n-th space refers to the $n + 1$-th space, as the *Lm* constructor requires its body to be an expression with one more free variable.

The code for the type checker is standard and uses a type stored in each application node (tx in Ap f x tx) to denote the type of the argument term for simplicity.

To evaluate the generators, we generated 2000 terms with a simple initial environment of 6 constants. The derived generator with three search strategies and one based on generate-and-filter were used. Figure 3.4 shows the results. The uniform search strategy is capable of generating terms of size up to 23. For larger sizes, the generator exceeded the resource limits (300 s and 4 GiB, marked with dotted lines). The generator that used limited backtracking allowed generating terms up to size 28, using 9 times less CPU time and over 11 times less memory than the uniform one at size 23. Unlimited backtracking improved memory consumption dramatically, up to 30-fold, compared to limited backtracking. The run time is improved only slightly with unlimited backtracking. Finally, the generator based on generate-and-filter exceeded the run times for all sizes, and is not included in the plots.

6.3 Testing GHC

Discovering strictness bugs in the GHC optimising Haskell compiler was our prime reason for generating random simply-typed lambda terms. To

Generator	Hand-written	Derived (size 30)
Terms per ctr ex. (k)	18.6	52.5
Gen. CPU time per ctr ex. (min)	1.7	14.0
Test CPU time per ctr ex. (min)	1.8	10.4
Tot. CPU time per ctr ex. (min)	3.5	24.4

Table 3.8: Performance of the reference hand-written term generator compared to a derived generator using backtracking with size 30. We compare the average number of terms that have to be generated before a counterexample (ctr ex.) is found, and how much CPU time the generation and testing consumes per found counterexample.

evaluate our approach, we compared its bug finding power to a hand-written generator that had been developed before (Pałka, 2012) using the same test property that had been used there.

Random simply-typed lambda terms were used for testing GHC by first generating type-correct Haskell modules containing the terms, and then using them as test data. In this case, we generated modules containing expressions of type $[\mathsf{Int}] \to [\mathsf{Int}]$ and compiled them with two different optimisation levels. Then, we tested their observable behaviour and compared them against each other, looking for discrepancies.

We implemented the generator using a similar data type as in Figure 3.1 extended with polymorphic constants and type constructors. For efficiency reasons we avoided having types in term application constructors, and used a type checker based on type inference, which is more complex but still easily implementable. It allows generators to scale up to larger effective term sizes because not having types in the term representation increases the density of well-typed terms.

A backtracking generator based on this data type was capable of generating terms containing 30 term constructors, and was able to trigger GHC failures. Other derived generators were not able to find counterexamples. Table 3.8 shows the results of testing GHC both with the hand-written simply-typed lambda term generator and our derived generator. The hand-written generator used for comparison generated terms of sizes from 0 to about 90, with most terms falling in the range of 20–50. It needed the least total CPU time to find a counterexample, and the lowest number of generated terms. The derived generator needs almost 7 times more CPU time per failure than the hand-written one.

The above results show that a generator derived from a predicate can be used to effectively find bugs in GHC. The derived generator is less effective than a hand-written one, but is significantly easier to develop. Developing an efficient type-checking predicate required for the derived generator

Predicates	Backtracking	Backtracking c/o
1, 2, 3, 4, 5	13	15
1, 3, 4, 5	13	30
1, 3, 5	31	30

Table 3.9: Maximum practical sizes of values generated by derived program generators that use unlimited backtracking and backtracking with cut-off of 10k.

took a few days, whereas the development and tuning of the hand-written generator took an order of months.

6.4 Programs

The Program benchmark is meant to simulate testing of a simple compiler by generating random programs, represented by the following data type.

```
type Name    = String
data Program = New Name Program
             | Name := Expr
             | Skip
             | Program ≫ Program
             | If Expr Program Program
             | While Expr Program
data Expr    = Var Name
             | Add Expr Expr
```

The programs contain some common imperative constructs and declarations of new variables using New, which creates a new scope.

A compiler may perform a number of compilation passes, which would typically transform the program into some kind of normal form that may be required by the following pass. Our goal is to generate test data that satisfy the precondition in order to test the code of each pass separately. We considered 5 predicates on the program data type that model simple conditions that may be required by some compilation phases: (1) boundProgram saying that the program is well-scoped, (2) usedProgram saying that all bound variables are used, (3) noLocalDecls requiring all variables to be bound on the top level, (4) noSkips forbidding the redundant use of ≫ and Skip, and (5) noNestedIfs forbidding nested *if* expressions.

Table 3.9 shows maximum value sizes that can be practically reached by the derived generators for the program data type with different combinations of predicates. All runs were generating 2000 random programs with

resource limits (300 s and 4 GiB). When all predicates were used, the generators performed poorly being able to reach at most size 15. When the usedProgram predicate was omitted, the generator that uses limited backtracking improved considerably, whereas the one using unlimited backtracking remained at size 13. Removing the noSkips predicate turns the tables on the two generators improving the performance of the unlimited backtracking generator dramatically.

A generator based on generate-and-filter was also benchmarked, but did not terminate within the time limit for the sizes we tried.

6.5 Summary

All derived generators performed much better than ones based on generate-and-filter in three out of four benchmarks. In the GHC benchmark, using a generator based on generate-and-filter was comparable to using our uniform or near-uniform derived generators, and slower than a derived generator using backtracking. The backtracking generator was the only automatic generator that found any counterexamples, although less efficiently than a hand-written generator. However, as creating the derived generators was much quicker, we consider them an appealing alternative to hand-written generators.

The time and space overhead of the derived generators appeared to rise exponentially, or almost exponentially with the size of generated values in most cases we looked at, similarly to what can be seen in Figures 3.3 and 3.4.

In most cases the backtracking generator provided the best performance, which means that sometimes we may have to sacrifice our goal of having a predictable distribution. However, we found the backtracking generator to be very sensitive to the choice of the predicate. For example, some combinations of predicates in Section 6.4 destroyed its performance, while having less influence on the uniform and near-uniform generators. We hypothesise that this behaviour may be caused by regions of search space where the predicates evaluate values to a large extent before returning False. The backtracking search remain in such regions for a long time, in contrast to the other search that gives up and restarts after a number of values have been skipped.

Overall, the performance of the derived generators is practical for some applications, but reaching higher sizes of generated data might be needed for effective bug finding. In particular, being able to generate larger terms may improve the bug-finding performance when testing for GHC strictness bugs.

7 Related Work

There is a substantial amount of research on generating combinatorial structures both in pure mathematics and in computer science. These structures sometimes include trees or even (recursive) algebraic data types. Although this work does not deal with data constrained by arbitrary predicates, some of it could potentially be adapted to it in a similar way as we do in this paper.

An efficient algorithm to index into a size based enumeration of binary trees can be derived from a bijection to strings of nested parenthesis (Knuth, 2006). Boltzmann samplers can be used to generate objects from a wide range of combinatorial structures, with uniform distribution over values of an approximate or exact size (Flajolet, Zimmermann, and Cutsem, 1994; Flajolet, Fusy, and Pivoteau, 2007). Yorgey has explored the relation between a class of combinatorial objects called *species* and algebraic data types (Yorgey, 2010; Yorgey, 2014). This work can potentially be used for uniform random generation of algebraic types as well as some more complex structures involving symmetries.

Feat See also the chapter in this thesis (Paper I).

Our representation of spaces and efficient indexing is based on FEAT (Functional Enumeration of Algebraic Types) (Duregård, Jansson, and Wang, 2012). The practicalities of computing cardinalities and the deterministic indexing functions are described there. The inability to deal with complex data type invariants is the major concern for FEAT, which is addressed by this paper.

Lazy SmallCheck and Korat Lazy SmallCheck (Runciman, Naylor, and Lindblad, 2008) uses laziness of predicates to get faster progress in an exhaustive depth-limited search. Our goal was to reach larger, potentially more useful values than Lazy SmallCheck by improving on it in two directions: Using size instead of depth and allowing random search in sets that are too large to search exhaustively. Korat is a framework for testing Java programs (Boyapati, Khurshid, and Marinov, 2002). It uses similar techniques to exhaustively generate size-bounded values that satisfy the precondition of a method, and then automatically check the result of the method for those values against a postcondition.

SmallCheck has been applied to the problem of generating programs to test compilers (Reich, Naylor, and Runciman, 2012). The work focuses on limiting the search space to include interesting programs without containing too many variants of similar programs. Notably some of these limitations, such as limiting function arity, arise from the use of depth-bound as opposed to size-bound.

Lazy instantiation A framework for generating values satisfying a computable predicate has been proposed based on explicit term representation of computable predicates (Lindblad, 2008). It uses logic variables to represent unrefined parts of the input data, and performs backtracking search with their successive refinement. The framework performs reductions of the predicate program explicitly, and shares its intermediate results for similar arguments, which may be beneficial for computationally expensive predicates. The framework can be adapted to perform random search, but only limited experiments have been performed on it.

EasyCheck: Test Data For Free EasyCheck is a library for generating random test data written in the Curry functional logic programming language (Christiansen and Fischer, 2008). Its generators define search spaces, which are enumerated using diagonalisation and randomising local choices. In this way values of larger sizes have a chance of appearing early in the enumeration, which is not the case when breadth-first search is used. The Curry language supports narrowing, which can be used by EasyCheck to generate values that satisfy a given predicate. The examples that are given in the paper suggest that, nonetheless, micro-management of the search space is needed to get a reasonable distribution. The authors point out that their enumeration technique has the problem of many very similar values being enumerated in the same run.

Metaheuristic Search In the GödelTest (Feldt and Poulding, 2013) system, so-called metaheuristic search is used to find test cases that exhibit certain properties referred to as *bias objectives*. The objectives are expressed as fitness metrics for the search such as the mean height and width of trees, and requirements on several such metrics can be combined for a single search. It may be possible to write a GödelTest generator by hand for well typed lambda terms and then use bias objectives to tweak the distribution of values in a desired direction, which could then be compared to our work.

Lazy Nondeterminism There is some recent work on embedding nondeterminism in functional languages (Fischer, Kiselyov, and Shan, 2011). As a motivating example an `isSorted` predicate is used to derive a sorting function, a process which is quite similar to generating sorted lists from a predicate. The framework is very general and could potentially be used both for implementing SmallCheck style enumeration and for random generation.

Generating Lambda Terms There are several other attempts at enumerating or generating well typed lambda terms. Generic programming has

been used to exhaustively enumerate lambda terms by size (Yakushev and Jeuring, 2009). The description focuses mainly on the generic programming aspect, and the actual enumeration appears to be mainly proof of concept with very little discussion of the performance of the algorithm. There has been some work on counting lambda terms and generating them uniformly (Grygiel and Lescanne, 2013). This includes generating well typed terms by a simple generate-and-filter approach.

8 Conclusion

The performance of our generators depends on the strictness and evaluation order of the used predicate. The generator that performs unlimited backtracking was especially sensitive to the choice of predicate, as shown in Section 6.4. Similar effects have been observed in Korat (Boyapati, Khurshid, and Marinov, 2002), which also performs backtracking.

We found that for most predicates unbounded backtracking is the fastest. But unexpectedly, for some predicates imposing a bound on backtracking improves the run time of the generator. This also makes the distribution more predictable, at the cost of increased memory consumption. We found tweaking the degree of backtracking to be a useful tool for improving the performance of the generators, and possibly trading it for distribution guarantees.

Our method aims at preserving the simplicity of generate-and-filter type generators, but supporting more realistic predicates that accept only a small fraction of all values. This approach works well provided the predicates are lazy enough.

Our approach reduces the risk of having incorrect generators, as coming up with a correct predicate is usually much easier than writing a correct dedicated generator. Creating a predicate which leads to an efficient derived generator on the other hand, is more difficult, and often requires careful reasoning about its strictness and evaluation order.

Even though performance remains an issue when generating large test cases, experimental results show that our approach is a viable option for generating test data in many realistic cases.

Acknowledgements

This research has been supported by the Resource-Aware Functional Programming (RAW FP) grant awarded by the Swedish Foundation for Strategic Research. We would like to thank David Christiansen for his valuable feedback, and anonymous referees for their detailed and helpful reviews.

Paper IV

Black-box Mutation Testing

This chapter is previously unpublished work. The work described was first presented at in a short Lightning Talk at the 2014 Haskell Implementors Workshop.

Black-box Mutation Testing

Jonas Duregård

Abstract

Mutation testing evaluates software test suites by automatically mutating the source code of the function under test, intentionally injecting errors. Test suites are scored by how often they detect such errors. But in languages with higher order functions, tested functions can be mutated like any other value and passed as a parameter to the test suite. This approach is completely black-box, it does not access the source code of the function under test. As such it can mutate functions that use any language features and extensions including FFI and meta-programming, where a traditional mutation testing framework would fail.

We define a proof of concept implementation of automatic black-box mutation testing for QuickCheck that is easy to use, trivial to implement and can mutate any function merely by mutating its output.

We demonstrate that our implementation is useful for measuring the quality in some realistic examples.

1 Introduction

The problem we address in this paper can easily be exemplified. Suppose we are using QuickCheck (Claessen and Hughes, 2000) to test an implementation of an efficient bucket sort algorithm, implemented as a Haskell function: $\mathsf{bSort} :: [\mathsf{Int}] \to [\mathsf{Int}]$. We test it by comparison to a trusted (but less efficient) reference implementation of sorting. The execution of the test could look like this:

```
*Main> quickCheck (\xs -> bSort xs == sort xs)
+++ OK, passed 100 tests.
```

Or it could (accidentally) look like this:

```
*Main> quickCheck (\xs -> bSort xs == bSort xs)
+++ OK, passed 100 tests.
```

The simple mistake of writing bSort instead of sort renders the second test useless. Still, the output from QuickCheck is identical in both cases. One of the aims of this paper is for a great big warning sign to flash in the second case, indicating that nothing interesting was actually tested. More generally, we want QuickCheck to produce an estimate of the quality of the predicate as a specification for the function. In this case the first property is

a complete (functional) specification and the second has almost zero value as a specification.

Two common techniques for evaluating test suite strength are code coverage analysis and mutation testing. Code coverage is unlikely to help in this case, since the same code is executed (twice) in the latter test run. But for mutation testing this is an ideal example, representing the two extremes on the mutation score spectrum: The first one can kill any (proper) mutant of `bSort`, the second one is unable to kill any mutant. But this raises another problem, we need to know a lot more about `bSort`.

QuickCheck is black-box. We know nothing about the implementation of `bSort`, only its type. If it can be compiled it can be tested. For all we know, `bSort` could be using ten bleeding edge extensions to Haskell, all of which would have to be supported by the mutation testing framework for mutation testing to work. It may even be a chunk of C-code accessed via FFI, in which case effective mutation testing becomes extraordinarily complicated.

It would seem that to evaluate black-box test suites, we need black-box mutation testing.

2 Background

Traditionally, mutation testing is performed by mutating the source code of the tested function in ways that emulate programmer mistakes. The mutants created through this process are compiled and the test suite is executed to test the mutant instead of the original code. If the test suite reports a failure for a mutant, we say that mutant is killed. The strength of the test suite is estimated by its capacity to kill mutants, reported to the tester as a mutation score calculated from the ratio of killed mutants.

The mutants are typically generated using mutators for various language constructs and functions. Each mutator describes a change such as swapping the operand order of a non-commutative operator or switching a $<=$ to $<$. A separate mutant is created for every source code element that match the pattern, so the number of mutants is proportional to the code size of the tested function.

Evidently, black-box mutation cannot operate remotely like this. Since the only thing we know about the tested function is its type, and by applying it to various inputs we can learn what output it yields.

3 A Tiny Mutation Testing Framework

We have developed a prototype of a mutation testing framework based on QuickCheck. It is implemented in the QuickCheck property language, without extending the library. Our tool measures mutation score statistically. For every generated test case, we runs the property both on the original function and on a randomly generated mutant of the function. After all tests are completed the tool reports the number of killed and surviving mutants. This means that mutation score can be defined as the probability that a random mutant survives a random test case of the property, and we estimate this probability by sampling.

This approach is rather different from the classic mutation testing strategy, where each mutant is tested against a whole test suite. Our hope is that this gives a more fine grained mutation score that can distinguish even between fairly strong properties.

Another significant deviation from traditional mutation testing is that we ensure not just that mutants differ from the original function, but that they differ for the particular inputs exercised by the current test case. This guarantees that a complete specification always kills all mutants, because there is always an observable change relevant for the particular test case.

The only requirement on properties compared to QuickCheck is that they parametrize over the function under test, and that the function can be randomly mutated (it is an instance of the Mutant type class). Other than this, the property can be any testable property, so adapting an existing property is often as easy as adding a parameter for the tested function. The complete code for the mutation testing framework is as follows (there is no need to understand the specifics other than the type signature for mut, the code is just included to demonstrate it really is tiny):

```
class Arbitrary a ⇒ Mutant a where
  mutate :: a → Gen a

mut :: (Mutant f, Testable p) ⇒ f → (f → p) → Property
mut f p = property $ do
  g ← mutate f
  return $ collectFirst (p g, p f)

copySeed :: Gen a → Gen a → Gen (a, a)
copySeed (MkGen a) (MkGen b) = MkGen (λr n → (a r n, b r n))

addLabel :: String → Result → Result
addLabel s res = res {
    labels = Map.insertWith max s 0 (labels res),
    stamp = Set.insert s (stamp res) }

collectFirst :: Testable p ⇒ (p, p) → Property
collectFirst (p1, p2) = MkProperty $ do
  (r1, r2) ← copySeed pr1 pr2
```

```
    return $ MkProp $ IORose $ do
      MkRose a _ ← reduceRose (unProp r1)
      return $ fmap (addLabel (killStatus $ ok a)) (unProp r2)
    where
      MkProperty pr1 = property p1
      MkProperty pr2 = property p2
  killStatus (Just True)  = "Mutants survived"
  killStatus (Just False) = "Mutants killed"
```

Granted, saying that this is complete is a slight overstatement, since it still requires instances of Mutant to be usable for anything. Instances for lists, Booleans, etc. are straightforward. Just randomly change a constructor for another one or recursively mutate some other value. Writing the mutating functions is very similar to writing shrinking functions and the process can be automated using Template Haskell or generic programming libraries. The most important and trickiest instance is for functions. Just mutating the output for a single random input would not work, since the statistical chance that a single test exercises this particular input is insignificant. Instead we mutate the function a little bit for all inputs. We use the CoArbitrary class of QuickCheck to make the random seed used for mutating the output depend on the input (so identical output can be mutated differently).

```
instance (CoArbitrary a, Mutant b) ⇒ Mutant (a → b) where
  mutate = mutateFun
mutateFun :: (Mutant b, CoArbitrary a) ⇒ (a → b) → Gen (a → b)
mutateFun f = promote (λa → coarbitrary a (mutate (f a)))
```

With this and some additional instances in place (Int and [] specifically) we can finally start killing mutants! Looking back at the problematic example in the introduction to this paper, the problem is now detected:

```
*Main> quickCheck $ mut bSort (\bSort xs -> bSort xs == sort xs)
+++ OK, passed 100 tests (100% Mutants killed).

*Main> quickCheck $ mut bSort (\bSort xs -> bSort xs == bSort xs)
+++ OK, passed 100 tests (100% Mutants survived).
```

To test if this scales to giving useful output for properties that are incomplete but not tautological, we wrote several properties to specify the insert function from the Data.List module. In Figure 4.1 we define six different properties on insert. The properties are of various quality ranging from a very poor quality (prop_insert0) to a complete specification (prop_insert5). Executing the run function gives the following output:

```
run :: IO ()
run = do
  let
    qc = quickCheckWith stdArgs {maxSuccess = 1000}
    runMut :: (Testable p) => ((Int -> [Int] -> [Int]) -> p) -> IO ()
    runMut n ip = putStrLn ("prop_insert" ++ show n) qc $ mut insert ip
  runMut 0 prop_insert0
  runMut 1 prop_insert1
  runMut 2 prop_insert2
  runMut 3 prop_insert3
  runMut 4 prop_insert4
  runMut 5 prop_insert5
type Insert = Int -> [Int] -> [Int]
prop_insert0 :: Insert -> Int -> [Int] -> Bool
prop_insert1, prop_insert2, prop_insert3, prop_insert4, prop_insert5
  :: Insert -> Int -> OrderedList Int -> Bool
prop_insert0 insert x xs =
    not (ordered xs) ∨ ordered (insert x xs)
prop_insert1 insert x (Ordered xs) = ordered (insert x xs)
prop_insert2 insert x (Ordered xs) = let outp = insert x xs in
  ordered outp && x ∈ outp
prop_insert3 insert x (Ordered xs) = let outp = insert x xs in
  ordered outp
  && length outp == length xs + 1
prop_insert4 insert x (Ordered xs) = let outp = insert x xs in
  ordered outp
  && x ∈ outp
  && length outp == length xs + 1
prop_insert5 insert x (Ordered xs) = let outp = insert x xs in
  ordered outp
  && null (xs \\ outp)
  && [x] == (outp \\ xs)
ordered :: [Int] -> Bool
ordered (x : xys@(y: _)) = x <= y && ordered xys
ordered _ = True
```

Figure 4.1: Five different properties of insert on lists, of various quality

```
*Main> run
prop_insert0
+++ OK, passed 1000 tests:
96% Mutants survived
 3% Mutants killed

prop_insert1
+++ OK, passed 1000 tests:
62% Mutants survived
37% Mutants killed

prop_insert2
+++ OK, passed 1000 tests:
64% Mutants killed
35% Mutants survived

prop_insert3
+++ OK, passed 1000 tests:
87% Mutants killed
12% Mutants survived

prop_insert4
+++ OK, passed 1000 tests:
90% Mutants killed
 9% Mutants survived

prop_insert5
+++ OK, passed 1000 tests
100% Mutants killed.
```

As can be seen, the mutation scores of these predicates correlate well with their specification power.

Mutating single outputs Mutating all values of a function works well when the function is only used once in the property, but consider for instance to test associativity of an operator:

$$\mathsf{assoc}\ (+)\ \mathsf{a}\ \mathsf{b}\ \mathsf{c} = (\mathsf{a}+\mathsf{b})+\mathsf{c}\ ==\ \mathsf{a}+(\mathsf{b}+\mathsf{c})$$

In this case up to four points of the function are mutated independently. This contradicts the overall strategy of introducing minimal change in mutants. Furthermore, since applications of the tested operator are nested, the input to the top level applications are always mutated. This tends to give more drastic changes than desired.

The procedure can be modified to mutate the function for a single input or a given maximal number of inputs, but still preserving the requirement

that one of the tested applications is mutated. This is done by first running the property with the original function but monitoring what inputs it is fed, like a simple form of memoization. Then for the mutation run of the same test case a mutant is constructed by mutating the output for one of the used inputs.

4 Advantages of black-box mutation testing

Ultra-lightweight The greatest advantage of black-box mutation testing over white-box mutation testing is the ease with which it can be applied and implemented.

Full language support White-box mutation testing only works for code using supported language constructs. Syntactic mutations for every language feature must be coded to preserve type correctness (and termination). Some language features are clearly out of reach (such as foreign function interfaces).

In black-box mutation testing, any function that can be compiled can be mutated. The only requirement is that a mutation procedure is provided for the output type of the function (using a type class). For algebraic data types this can be automated.

Avoiding equivalent mutants In white-box mutation testing, there is a risk that a syntactic modification of the source code does not change the semantics of the function. In such cases a perfect mutation score is impossible even for complete specifications. Compensating for this is very difficult and typically requires manual effort (Wong and Mathur, 1995).

In black-box mutation testing, equivalent mutants can be avoided altogether. If mutation is carried out by modifying the output of the function it inevitably causes an observable change in its behaviour.

5 Drawbacks and future Work

This paper is intended as a proof of concept for black-box mutation testing. Our current implementation picks the low-hanging fruit in this area, but much can be done to improve it.

Mutation Heuristics The foremost challenge of black-box mutation testing is generating mutants that convincingly mimics bugs. In white-box mutation testing, this relies on the idea that small changes in the source

code correspond to simple programmer mistakes. This means that capability to kill mutants corresponds somewhat to capability of finding programmer mistakes.

In black-box mutation testing, individual mutants typically do not correspond to naturally occurring bugs, so one cannot rely on this to motivate the significance of mutation score.

In both black-box and white-box mutation testing it is possible that different strategies for generating mutants give significantly different scores for the same test suites. Determining which strategy is best requires a comparison of how mutation scores correlate with bug finding capacity. This question is outside the scope of this proof of concept paper, but interesting as a topic of future research.

Black-box data coverage check Our tool is limited to checking the specification strength of a predicate on inputs for which it is used in the property. A high mutation score should be interpreted as a strong specification for the inputs the property covers. This works well for pre-post condition type properties that applies the tested function on a universally quantified variable and checks the output against an oracle. For functions that apply the tested function to a non-variable, the output may be misleading. For instance consider this property:

```
prop_singlesort x = sort [x] == [x]
```

This is clearly not a complete specification of sorting, but it has a 100% mutation score with our tool. The reason is that it is a complete specification for the domain on which it is used (singleton lists).

Also, if the random generator used for the input is flawed, e.g. if it only generated singleton lists for list types, this weakness in the test suite would not be detected.

One way to circumvent this is to change the way mutations are generated and tested, so that it may generate mutants that mutate the tested function for inputs that are not used by the test suite. Such mutants are trivially survivors, since the evaluated parts of the function are identical to the original function. This means that the actual mutation of the output is irrelevant, only the choice of input matters.

This means that the mutation score in this case becomes an amalgam of two independent measures: One testing if it detects mutants in covered data, and one testing data coverage. In our opinion it would be beneficial to keep these measures separate, placing properties on a two-dimensional scale of strength/coverage. There are many potential ways in which one can automatically measure data coverage in a black-box context and research is needed to evaluate the usefulness of such methods.

Exhaustive search Our implementation uses randomly generated mutants for random testing. Other popular property based testing frameworks such as SmallCheck (Runciman, Naylor, and Lindblad, 2008) and FEAT (Duregård, Jansson, and Wang, 2012) use bounded exhaustive search for counterexamples. It seems natural that black-box mutation testing in this context would use bounded exhaustive search for mutants. Instead of testing random mutants on random test cases it would test all small mutants on all small inputs. A possible problem with this approach is that testing all mutants may use too much computational resources, depending on how size is defined for mutants.

The idea of using random search for surviving mutants in random testing, and exhaustive search for mutants in exhaustive testing highlights an advantages of adding black-box mutation testing to black-box testing frameworks: Searching for counterexamples and searching for surviving mutants are two closely related problems, and the latter can be implemented on top of the former with relative ease.

6 Related Work

Research on mutation testing dates back to the seventies (DeMillo, Lipton, and Sayward, 1978). Despite strong academic interest in mutation testing, application of the technique is limited in the software industry. This is often attributed to high costs (Usaola and Mateo, 2010; Wong and Mathur, 1995). A recent survey claims there is an increasing interest in mutation testing and that the technique is reaching maturity (Jia and Harman, 2011).

MuCheck is a white-box mutation testing framework for Haskell (Le et al., 2014). It features a number of built-in mutation transformations for various language features along with a set of simpler substitution based mutation transformations for standard Haskell functions, and users can write custom transformations. An example of a mutation transformation could be "replace $<$ with $>$". Given a source file MuCheck will then create a mutant for each syntactic occurrence of $<$, by replacing it with $>$.

Semantic Mutation Testing One way to characterize black-box mutation testing is that it mutates the semantics of tested functions rather than the syntax. This is not to be confused with the technique called semantics mutation testing (Clark, Dan, and Hierons, 2010) that creates mutants by altering the semantics of the language the tested code is written in.

Specification Based Mutation testing Budd and Gopal, 1985 introduced a technique called specification mutation. As the name implies it mutates

a specification of the code instead of the code itself. The system also derives test cases from the specification. The technique has been referred to as black-box (Murnane and Reed, 2001), because it only inspects input and output of the tested program. Still, the specification is mutated syntactically (white-box) and the language used for specifications is very limited compared to QuickCheck.

Although the goal of specification mutation is the same as conventional mutation testing (improving a test suite by mutation analysis) it is takes a drastically different approach: Rather than finding if mutated programs tend to satisfy the specification, this technique checks if the program tends to satisfy mutated specifications.

FitSpec Braquehais and Runciman, (2016) have been working on a closely related tool called FitSpec. It works on property sets with the purpose of finding minimal complete specifications, and they use their own version of black-box mutation testing to find them.

7 Conclusion

Projects that use QuickCheck for rapid development with limited resources for testing have even more limited resources for evaluating test suites.

Black-box mutation testing brings the advantages of QuickCheck to mutation testing: it is lightweight and black-box. We have shown that a very simplistic approach can provide useful information about the relative strengths of properties with almost no extra effort from the tester.

References

Andoni, Alexandr, Dumitru Daniliuc, and Sarfraz Khurshid (2003). *Evaluating the "Small Scope Hypothesis"*. Tech. rep. (cit. on p. 88).

Arts, Thomas et al. (2006). "Testing Telecoms Software with Quviq QuickCheck". In: *Proceedings of the Workshop on Erlang (Erlang 2006)*. New York, NY, USA: ACM, pp. 2–10 (cit. on p. 95).

Barendregt, H.P. (1984). *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland. ISBN: 9780444867483 (cit. on p. 3).

Beizer, Boris (1990). *Software Testing Techniques (2Nd Ed.)* New York, NY, USA: Van Nostrand Reinhold Co. ISBN: 0-442-20672-0 (cit. on p. 1).

Bird, Richard and Lambert Meertens (1998). "Nested datatypes". In: *Mathematics of Program Construction*. Ed. by J. Jeuring. Vol. 1422. LNCS. Springer-Verlag, pp. 52–67 (cit. on p. 3).

Boyapati, Chandrasekhar, Sarfraz Khurshid, and Darko Marinov (2002). "Korat: Automated Testing Based on Java Predicates". In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*. New York, NY, USA: ACM, pp. 123–133 (cit. on pp. 88, 126, 128).

Braquehais, Rudy and Colin Runciman (2016). "FitSpec: Refining Property Sets for Functional Testing". In: *Proceedings of the 2016 Haskell Symposium*. Nara, Japan: ACM (cit. on p. 140).

Budd, Timothy A. and Ajei S. Gopal (1985). "Program testing by specification mutation". In: *Computer Languages* 10.1, pp. 63 –73. ISSN: 0096-0551. DOI: 10.1016/0096-0551(85)90011-6 (cit. on p. 139).

Christiansen, Jan and Sebastian Fischer (2008). "EasyCheck: test data for free". In: *FLOPS'08*. Springer, pp. 322–336 (cit. on pp. 50, 88, 127).

Claessen, Koen (2012). "Shrinking and Showing Functions: (Functional Pearl)". In: *Proceedings of the 2012 Haskell Symposium*. Haskell '12. Copenhagen, Denmark: ACM, pp. 73–80. ISBN: 978-1-4503-1574-6. DOI: 10.1145/2364506.2364516 (cit. on p. 58).

Claessen, Koen, Jonas Duregård, and Michał H. Pałka (2015). "Generating constrained random data with uniform distribution". In: *Journal of Functional Programming* 25. ISSN: 1469-7653. DOI: 10.1017/S0956796815000143 (cit. on pp. 67, 88).

Claessen, Koen, Jonas Duregård, and Michał H. Pałka (2014). "Generating Constrained Random Data with Uniform Distribution". In: *Proceedings of the International Conference on Functional and Logic Programming (FLOPS 2014)*. Ed. by Michael Codish and Eijiro Sumii. Vol. 8475. LNCS. Springer International Publishing, pp. 18–34. ISBN: 978-3-319-07150-3. DOI: 10.1007/978-3-319-07151-0_2 (cit. on p. 97).

Claessen, Koen and John Hughes (2000). "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *ICFP '00*. ACM, pp. 268–279 (cit. on pp. 8, 22, 49, 88, 95, 131).

Claessen, Koen et al. (2015). "Intelligent Computer Mathematics: International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings." In: Cham: Springer International Publishing. Chap. TIP: Tons of Inductive Problems, pp. 333–337. ISBN: 978-3-319-20615-8. DOI: 10.1007/978-3-319-20615-8_23 (cit. on p. 90).

Clark, J. A., H. Dan, and R. M. Hierons (2010). "Semantic Mutation Testing". In: *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pp. 100–109. DOI: 10.1109/ICSTW.2010.8 (cit. on p. 139).

Danielsson, Nils Anders et al. (2006). "Fast and Loose Reasoning is Morally Correct". In: *POPL'06*. Charleston, South Carolina, USA: ACM Press, pp. 206–217 (cit. on p. 4).

De Bruijn, Nicolaas Govert (1972). "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae*. Vol. 75. 5. Elsevier, pp. 381–392 (cit. on p. 3).

DeMillo, R. A., R. J. Lipton, and F. G. Sayward (1978). "Hints on Test Data Selection: Help for the Practicing Programmer". In: *Computer* 11.4, pp. 34–41. ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218136 (cit. on pp. 6, 139).

Duchon, Philippe et al. (2004). "Boltzmann Samplers for the Random Generation of Combinatorial Structures". In: *Combinatorics, Probability and Computing* 13.4–5, pp. 577–625. DOI: 10.1017/S0963548304006315 (cit. on p. 51).

Duregård, Jonas (2009). "AGATA: Random generation of test data". Series/report no. 2009/67. MA thesis. University of Gothenburg (cit. on p. 50).

Duregård, Jonas and Patrik Jansson (2011). "Embedded Parser Generators". In: *Proceedings of the 4th ACM Symposium on Haskell*. Tokyo, Japan: ACM, pp. 107–117. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034689 (cit. on p. 44).

Duregård, Jonas, Patrik Jansson, and Meng Wang (2012). "Feat: Functional Enumeration of Algebraic Types". In: *Proceedings of the 2012 symposium on Haskell*. Copenhagen, Denmark: ACM, pp. 61–72. ISBN: 978-1-4503-1574-6. DOI: 10.1145/2364506.2364515 (cit. on pp. 58, 67, 68, 88, 97, 98, 126, 139).

Feldt, Robert and Simon Poulding (2013). "Finding Test Data with Specific Properties via Metaheuristic Search". In: *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE 2013)*. IEEE, pp. 350–359 (cit. on p. 127).

Fischer, Sebastian, Oleg Kiselyov, and Chung-chieh Shan (2011). "Purely functional lazy nondeterministic programming". In: *Journal of Functional Programming* 21.4-5, pp. 413–465 (cit. on p. 127).

Flajolet, Philippe, Éric Fusy, and Carine Pivoteau (2007). "Boltzmann sampling of unlabelled structures". In: *Proceedings of the Workshop on Analytic Algorithmic and Combinatorics*. New Orleans, USA: SIAM Press, pp. 201–211 (cit. on p. 126).

Flajolet, Philippe and Bruno Salvy (1995). "Computer Algebra Libraries for Combinatorial Structures". In: *J. Symb. Comput.* 20.5/6, pp. 653–671. DOI: 10.1006/jsco.1995.1070 (cit. on p. 52).

Flajolet, Philippe, Paul Zimmermann, and Bernard Van Cutsem (1994). "A calculus for the random generation of labelled combinatorial structures". In: *Theoretical Computer Science* 132.1–2, pp. 1 –35. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90226-7 (cit. on p. 126).

Gibbons, Jeremy (2007). "Datatype-Generic Programming". In: *Spring School on Datatype-Generic Programming*. Ed. by R. Backhouse et al. Vol. 4719. LNCS. Springer-Verlag (cit. on p. 3).

Grygiel, Katarzyna and Pierre Lescanne (2013). "Counting and generating lambda terms". In: *Journal of Functional Programming* 23 (05), pp. 594–628 (cit. on p. 128).

Hanus, Michael et al. (2006). *Curry: An Integrated Functional Logic Language*. Version 0.8.2. Available from `http://www.informatik.uni-kiel.de/~curry/report.html`. (cit. on p. 50).

Herzig, Kim, Sascha Just, and Andreas Zeller (2013). "It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, pp. 392–401. ISBN: 978-1-4673-3076-3 (cit. on p. 5).

Hinchey, Michael G and Jonathan P Bowen (2012). *Industrial-strength formal methods in practice*. Springer Science & Business Media (cit. on p. 5).

Hughes, John (2007). "QuickCheck testing for fun and profit". In: *International Symposium on Practical Aspects of Declarative Languages*. Springer, pp. 1–32 (cit. on p. 89).

Huizinga, Dorota and Adam Kolawa (2007). *Automated defect prevention: best practices in software management*. John Wiley & Sons (cit. on p. 6).

Jackson, D. and C. A. Damon (1996). "Elements of style: analyzing a software design feature with a counterexample detector". In: *IEEE Transactions on Software Engineering* 22.7, pp. 484–495. ISSN: 0098-5589. DOI: `10.1109/32.538605` (cit. on p. 88).

Jackson, Daniel (2006). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press. ISBN: 0262101149 (cit. on p. 88).

Jia, Yue and Mark Harman (2011). "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Trans. Softw. Eng.* 37.5, pp. 649–678. ISSN: 0098-5589. DOI: `10.1109/TSE.2010.62` (cit. on p. 139).

Knight, John C. et al. (1997). "Why Are Formal Methods Not Used More Widely?" In: *Fourth NASA Formal Methods Workshop*, pp. 1–12 (cit. on p. 5).

Knuth, Donald E. (2006). *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. Addison-Wesley Professional. ISBN: 0321335708 (cit. on pp. 98, 126).

Korf, Richard E. (1985). "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search". In: *Artificial Intelligence* 27, pp. 97–109 (cit. on p. 58).

Le, Duc et al. (2014). "MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: ACM, pp. 429–432. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2628052 (cit. on p. 139).

Lindblad, Fredrik (2008). "Property Directed Generation of First-Order Test Data." In: *Trends in Functional Programming (TFP 2007)*. Intellect, pp. 105–123 (cit. on p. 127).

McBride, Conor and Ross Paterson (2008). "Applicative programming with effects". In: *Journal of functional programming* 18.01, pp. 1–13 (cit. on p. 67).

Murnane, T. and K. Reed (2001). "On the effectiveness of mutation analysis as a black box testing technique". In: *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pp. 12–20. DOI: 10.1109/ASWEC.2001.948492 (cit. on p. 140).

Myers, Glenford J. and Corey Sandler (2004). *The Art of Software Testing*. John Wiley & Sons. ISBN: 0471469122 (cit. on pp. 1, 6).

Offutt, A Jefferson (1994). "A practical system for mutation testing: help for the common programmer". In: *Test Conference, 1994. Proceedings., International*. IEEE, pp. 824–830 (cit. on p. 6).

Okasaki, Chris (1999). "Red-black Trees in a Functional Setting". In: *Journal of Functional Programming* 9.4, pp. 471–477. ISSN: 0956-7968. DOI: 10.1017/S0956796899003494 (cit. on p. 84).

Pałka, Michał H. (2012). "Testing an Optimising Compiler by Generating Random Lambda Terms". Licentiate Thesis. Chalmers University of Technology, Gothenburg, Sweden (cit. on pp. 14, 96, 121, 123).

Pałka, Michał H. et al. (2011). "Testing an optimising compiler by generating random lambda terms". In: *AST '11*. ACM, pp. 91–97 (cit. on pp. 50, 53, 96).

Peyton Jones, Simon, Simon Marlow, and Conal Elliot (1999). "Stretching the storage manager: weak pointers and stable names in Haskell". In: *IFL'99*. LNCS. Springer (cit. on p. 52).

Reich, Jason S, Matthew Naylor, and Colin Runciman (2012). "Lazy generation of canonical test programs". In: *Implementation and Application*

of Functional Languages (IFL 2012). Vol. 7257. LNCS. Springer, pp. 69–84 (cit. on p. 126).

Reich, Jason S., Matthew Naylor, and Colin Runciman (2013). "Advances in Lazy SmallCheck". In: *Implementation and Application of Functional Languages: 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*. Ed. by Ralf Hinze. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 53–70. ISBN: 978-3-642-41582-1. DOI: 10.1007/978-3-642-41582-1_4 (cit. on p. 87).

Rodriguez, Alexey et al. (2008). "Comparing Libraries for Generic Programming in Haskell". In: *Haskell'08*. ACM, pp. 111–122 (cit. on p. 3).

Runciman, Colin, Matthew Naylor, and Fredrik Lindblad (2008). "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values". In: *Haskell '08*. ACM, pp. 37–48 (cit. on pp. 9, 13, 21, 49, 59, 87, 97, 103, 116, 126, 139).

Takanen, Ari, Jared D Demott, and Charles Miller (2008). *Fuzzing for software security testing and quality assurance*. Artech House (cit. on p. 7).

Tassey, Gregory (2002). "The economic impacts of inadequate infrastructure for software testing". In: *National Institute of Standards and Technology, RTI Project* (cit. on p. 1).

Usaola, Macario Polo and Pedro Reales Mateo (2010). "Mutation testing cost reduction techniques: a survey". In: *IEEE Software* 27.3, p. 80 (cit. on p. 139).

Uszkay, Gordon J. and Jacques Carette (2012). *The Haskell package* gencheck. http://hackage.haskell.org/package/gencheck (cit. on p. 50).

Wang, Jue (2005). *Generating random lambda calculus terms*. Tech. rep. Boston University (cit. on p. 50).

Wong, W. Eric and Aditya P. Mathur (1995). "Reducing the Cost of Mutation Testing: An Empirical Study". In: *J. Syst. Softw.* 31.3, pp. 185–196. ISSN: 0164-1212. DOI: 10.1016/0164-1212(94)00098-0 (cit. on pp. 137, 139).

Yakushev, Alexey Rodriguez and Johan Jeuring (2009). "Enumerating Well-Typed Terms Generically". In: *AAIP 2009*. Vol. 5812. LNCS. Springer, pp. 93–116 (cit. on pp. 50, 51, 128).

Yorgey, Brent A. (2010). "Species and functors and types, oh my!" In: *Proceedings of the Haskell Symposium (Haskell 2010)*. Baltimore, Maryland,

USA: ACM, pp. 147–158. ISBN: 978-1-4503-0252-4. DOI: 10.1145/1863523.1863542 (cit. on pp. 51, 126).

Yorgey, Brent A. (2014). "Combinatorial Species and Labelled Structures". PhD thesis. University of Pennsylvania (cit. on p. 126).