SmallCheck and Lazy SmallCheck

automatic exhaustive testing for small values

Colin Runciman Matthew Naylor
University of York, UK
{colin,mfn}@cs.york.ac.uk

Fredrik Lindblad
Chalmers University of Technology /
University of Gothenburg, Sweden
fredrik.lindblad@cs.chalmers.se

Abstract

This paper describes two Haskell libraries for property-based testing. Following the lead of *QuickCheck* (Claessen and Hughes 2000), these testing libraries *SmallCheck* and *Lazy SmallCheck* also use type-based generators to obtain test-sets of finite values for which properties are checked, and report any counter-examples found. But instead of using a sample of randomly generated values they test properties for all values up to some limiting depth, progressively increasing this limit. The paper explains the design and implementation of both libraries and evaluates them in comparison with each other and with QuickCheck.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Verification

Keywords Embedded Language, Property-based Testing, Exhaustive Search, Lazy Evaluation, Type Classes

1. Introduction

In their ICFP'00 paper Claessen and Hughes propose an attractive approach to *property-based testing* of Haskell programs, as implemented in their *QuickCheck* library. Properties relating the component functions of a program are specified in Haskell itself. The simplest properties are just Boolean-valued functions, in which the body is interpreted as a predicate universally quantified over the argument variables, and a small library of operators provides for variations such as properties that are conditionally true. QuickCheck exploits Haskell's *type classes* to check properties using test-sets of *randomly generated* values for the universally-quantified arguments. If a failing case is discovered, testing stops with a report showing the counter-example.

Specifying properties in QuickCheck forces programmers to think hard about what makes their programs correct, and to record their conclusions in a precise form. Even this preliminary outcome of exact documentation has value. But the big reward for specifying properties is that they can be *tested automatically*, perhaps revealing bugs.

Haskell'08 September 25, 2008, Victoria, BC, Canada.

1.1 Motivation

Although QuickCheck is widely used by Haskell developers, and is often very effective, it has drawbacks. The definition of appropriate test generators for user-defined types is a necessary prerequisite for testing, and it can be tricky to define them so as to obtain a suitable distribution of values. However they are defined, if failing cases are rare, none may be tested *even though some of them are very simple*; this seems to be an inevitable consequence of using randomly selected tests.

We have therefore developed variations inspired by QuickCheck but using a different approach to the generation of test-data. Instead of random testing, we test properties for *all the finitely many values up to some depth*, progressively increasing the depth used. For data values, depth means depth of construction. For functional values, it is a measure combining the depth to which arguments may be evaluated and the depth of possible results.

The principal motivation for this approach can be summarised in the following observations, akin to the *small scope hypothesis* behind model-checking tools such as Alloy (Jackson 2006). (1) If a program fails to meet its specification in some cases, it *almost always* fails in some *simple* case. Or in contrapositive form: (2) If a program does not fail in any simple case, it *hardly ever* fails in *any* case. A successful test-run using our tools can give exactly this assurance: specified properties do not fail in any simple case. There is also a clear demarcation between tested and untested cases. Other advantages include a simple standard pattern for generators of user-defined types.

1.2 Contributions

Our main contributions are:

1. the design of *SmallCheck*, a library for property-based testing by exhaustive enumeration of small values, including support for *existential quantification*;
2. the design of *Lazy SmallCheck*, an alternative which tests properties for *partially-defined* values, using the results to prune test spaces automatically and *parallel conjunction* to enable further pruning (currently only first-order properties with universal quantifiers are supported);
3. a comparative evaluation of these tools and QuickCheck applied to a range of example properties.

1.3 Road-map

The rest of this paper is arranged as follows. Section 2 reviews QuickCheck. Section 3 describes SmallCheck. Section 4 describes Lazy SmallCheck. Section 5 is a comparative evaluation. Section 6 discusses related work. Section 7 suggests avenues for future work and concludes.

2. QuickCheck: a Review

2.1 Arbitrary Types and Testable Properties

QuickCheck defines a class of `Arbitrary` types for which there are random value generators. There are predefined instances of this class for most `Prelude` types. It also defines a class of `Testable` property types for which there is a method mapping properties to test computations. The `Testable` instances include:

```
instance Testable Bool
instance (Arbitrary a, Show a, Testable b)
  => Testable (a -> b)
```

Any `Testable` property can be tested automatically for some pre-assigned number of random values using

```
quickCheck :: Testable a => a -> IO ()
```

a class-polymorphic test-driver. It reports either success in all cases tested, or else a counterexample for which the property fails.

Example 1 Suppose the program being tested includes a function

```
isPrefix :: Eq a => [a] -> [a] -> Bool
```

that checks whether its first argument is a prefix of its second. One expected property of `isPrefix` can be specified as follows.

```
prop_isPrefix :: [Int] -> [Int] -> Bool
prop_isPrefix xs xs' = isPrefix xs (xs++xs')
```

The argument variables `xs` and `xs'` are understood to be *universally quantified*: the result of `prop_isPrefix` should be `True` *for all* (finite, fully defined) `xs` and `xs'`. As `prop_isPrefix` has a `Testable` type — its explicitly declared monomorphic type enables appropriate instances to be determined — it can now be tested.

```
Main> quickCheck prop_isPrefix
OK, passed 100 tests.
```

Alternatively, if `isPrefix` actually interprets its arguments the other way round, the output from `quickCheck` might be

```
Falsifiable, after 1 tests:
[1]
[2]
```

as the property then fails for `xs=[1]`, `xs'=[2]`. □

2.2 Generators for User-defined Types

For properties over user-defined types, appropriate `Arbitrary` instances must be written to generate random values of these types. QuickCheck provides various functions that are useful in this task.

Example 2 Consider the following data-type for logical propositions. To shorten the example, we restrict connectives to negation and disjunction.

```
data Prop = Var Name | Not Prop | Or Prop Prop
```

Assuming that an `Arbitrary Name` instance is defined elsewhere, here's how a QuickCheck user might define an `Arbitrary Prop` instance.

```
instance Arbitrary Prop where
  arbitrary = sized arbProp
    where arbProp 0 = liftM Var arbitrary
          arbProp n = frequency
            [ (1,liftM Var arbitrary)
            , (2,liftM Not (arbProp (n-1)))
            , (4,liftM2 Or (arbProp (n `div` 2))
                           (arbProp (n `div` 2))) ]
```

The `sized` function applies its argument to a random integer. The `frequency` function also abstracts over a random source, choosing one of several weighted alternatives: in the example, the probability of a `Var` construction is $1/7$. □

As this example shows, defining generators for recursive types requires careful use of controlling numeric parameters.

2.3 Conditional Properties

Often the body of a property takes the form of an implication, as it is only expected to hold under some condition. If implication were defined simply as a Boolean operator, then cases where the condition evaluates to `False` would count as successful tests. Instead QuickCheck defines an implication operator `==>` with the signature

```
(==>) :: Testable a => Bool -> a -> Property
```

where `Property` is a new `Testable` type. Test cases where the condition fails do not count.

Example 3 Suppose that an abstract data type for sets is to be implemented. One possible representation is an ordered list. Of course, sets are unordered collections, but an ordered list permits the uniqueness of the elements to be preserved more efficiently by the various set operations.

```
type Set a = [a]
```

Each set operation may assume that the lists representing the input sets are ordered, and must ensure that the same is true of any output sets. For example, the operation to insert an element into a set, of type

```
insert :: Ord a => a -> Set a -> Set a
```

should preserve the familiar `ordered` predicate on lists.

```
prop_insertSet :: Char -> Set Char -> Property
prop_insertSet c s =
  ordered s ==> ordered (insert c s)
```

If we apply `quickCheck` to `prop_insertSet`, few of the cases generated satisfy the condition, but a larger test-set is used to compensate. □

This example illustrates a difficulty with conditional properties that often arises in practice: what if a condition is rarely satisfied by randomly generated values of the appropriate type? QuickCheck has to limit the total number of cases generated, whether or not they satisfy the condition, so few if any valid tests are performed. The recommended solution is to define *custom generators*, designed to give only values that satisfy the desired condition. There are three main drawbacks of this: (1) writing good custom generators can be hard; (2) the property that *all and only* required values can be generated may be hard to verify; (3) properties showing that some invariant condition is preserved (a common pattern) must express the pre-condition in a generator, but the post-condition in a predicate.

2.4 Higher-order Properties

Higher-order functions are important components in many Haskell programs, and they too have properties that should be tested. One of the nice surprises in QuickCheck is that even functional values can be generated at random. The details of how this is done are quite subtle, but the key is an auxiliary method `coarbitrary` that transforms a generator for the result type in a way that depends on a given value of the argument type.

Functional test values do have the disadvantage that when a test fails QuickCheck is not able to show the functional values involved in the counter-example. They are displayed only by a `<<fun>>` place-holder.

2.5 Test Coverage

In the opinion of its authors "The major limitation of QuickCheck is that there is no measurement of test coverage." (Claessen and Hughes 2000). Users who want to assess coverage of the input domain can define functions that compute attributes of test data; QuickCheck provides helper functions to report the distribution of the attribute values for each randomly generated test-set. But this arrangement is quite ad hoc, and it requires extra work. A few years on, a coverage tool such as Hpc (Gill and Runciman 2007) can provide fine-grained source-coverage information not only for the program under test, but also for test-case generators and the tested properties. Yet even with 100% coverage of all these sources, simple failing cases may never be tested.

2.6 Counter Examples

A small counter-example is in general easier to analyse than a large one. QuickCheck, although beginning each series of tests with a small size parameter and gradually increasing it, is in many cases unlikely to find a simplest counter-example. To compensate for this, QuickCheck users may write type-specific *shrinking* functions. However, writing shrinking functions requires extra work and the mechanism still does not guarantee that a reported counter-example is minimal.

3. SmallCheck

3.1 Small Values

SmallCheck re-uses many of the property-based testing ideas in QuickCheck. It too tests whether properties hold for finite total values, using type-driven generators of test cases, and reports counter-examples. But instead of generating test cases at random, it enumerates all *small* test cases exhaustively. Almost all other changes follow as a result of this one. The principle SmallCheck uses to define *small values* is to bound their *depth* by some small natural number.

Small Data Structures

Depth is most easily defined for the values of algebraic data types. As usual for algebraic terms, the depth of a zero-arity construction is zero, and the depth of a positive-arity construction is one greater than the maximum depth of a component argument.

Example 2 (revisited) Recalling the data-type `Prop` of logical propositions, suppose the `Name` type is defined by:

```
data Name = P | Q | R
```

Then all `Name` values have depth 0, and the `Prop` value-construction `Or (Not (Var P)) (Var Q)` has depth 3. □

Small Tuples

The rule for tuples is a little different. The depth of the zero-arity tuple is zero, but the depth of a positive-arity tuple is just the maximum component depth. Values are still bounded as tuples cannot have recursive components of the same type.

Small Numeric Values

For primitive *numeric types* the definition of depth is with reference to an imaginary representation as a data structure. So the depth of an *integer* `i` is its absolute value, as if it was constructed algebraically as $\texttt{Succ}^{\texttt{i}}\ \texttt{Zero}$. The depth of a *floating point* number $\texttt{s} \times 2^{\texttt{e}}$ is the depth of the integer pair `(s,e)`.

Example 4 The small floating point numbers, of depth no more than 2, are `-4.0`, `-2.0`, `-1.0`, `-0.5`, `-0.25`, `0.0`, `0.25`, `0.5`, `1.0`, `2.0` and `4.0`. □

Small Functions

Functions generated as test cases should give totally defined results (given totally defined arguments) so that they do not cause undefined test computations. There is a natural link between this requirement and *depth-bounded recursion* which allows any function of a data-type argument to be represented non-recursively by formulating its body as nested case expressions. The depth of a function represented in this way is defined as the maximum, for any argument-result pair, of the *depth of nested case analysis of the argument plus the depth of the result* [1]. This rule is consistent with the principle of appealing to an algebraic-term representation: we are treating each `case` like a constructor with the bodies of its alternatives as components.

Example 5 The `Bool -> Bool` functions of depth zero are:

```
\b -> True
\b -> False
```

And those of depth one are:

```
\b -> case b of {True  -> True  ; False -> True }
\b -> case b of {True  -> True  ; False -> False}
\b -> case b of {True  -> False ; False -> True }
\b -> case b of {True  -> False ; False -> False}
```

As `True` and `False` have no sub-components for deeper analysis, there are no `Bool -> Bool` functions of depth two or more. □

3.2 Serial Types

Instead of a class `Arbitrary` of types with a random value generator, SmallCheck defines a class `Serial` of types that can be enumerated up to a given depth.

Serial Data

For all the `Prelude` data types, `Serial` instances are predefined. Writing a new `Serial` instance for an algebraic datatype is very straightforward. It can be concisely expressed using a family of combinators `cons<N>`, generic across any combination of `Serial` component types, where `<N>` is constructor arity.

Example 2 (revisited) The `Prop` datatype has constructors `Var` and `Not` of arity one, and `Or` of arity two. A `Serial` instance for it can be defined by

```
instance Serial Prop where
  series = cons1 Var \/ cons1 Not \/ cons2 Or
```

assuming a similar `Serial` instance for the `Name` type. □

A `series` is just a function from depth to finite lists

```
type Series a = Int -> [a]
```

and sum and product over two series are defined by

```
(\/) :: Series a -> Series a -> Series a
s1 \/ s2 = \d -> s1 d ++ s2 d

(><) :: Series a -> Series b -> Series (a, b)
s1 >< s2 = \d -> [(x,y) | x <- s1 d, y <- s2 d]
```

The `cons<N>` family of combinators is defined in terms of `><`, decreasing and checking the depth appropriately. For instance:

```
cons0 c = \d -> [c]
cons1 c = \d -> [c a   | d > 0, a <- series (d-1)]
cons2 c = \d -> [c a b | d > 0,
                (a,b) <- (series >< series) (d-1)]
```

[1] The current implementation by default generates strict functions, and counts only *nested* `case` occurrences when determining depth.

Serial Functions

To generate functions of type `a->r` requires, in addition to a `Serial` instance for result type `r`, an auxiliary method `coseries` for argument type `a`, analogous to QuickCheck's `coarbitrary`. Again predefined combinators support a standard pattern of definition: this time the `alts<N>` family to generate case alternatives.

Example 2 (Revisited) Here is a `coseries` definition for the `Prop` datatype of propositions, using the standard pattern.

```
coseries rs d = [ \p -> case p of
                        Var n    -> var n
                        Not p1   -> not p1
                        Or p1 p2 -> or p1 p2
                | var <- alts1 rs d ,
                  not <- alts1 rs d ,
                  or  <- alts2 rs d ]
```

Explicit fresh variable names are needed (1) in the `case` alternatives of the lambda body, (2) in the function-list generators, and (3) to pass on the result series and the bounding depth. So the pattern of definition here, though still straightforward, is more verbose than for `series`. □

The first few members of the `alts<N>` family are defined by:

```
alts0 as d = as d
alts1 bs d = if d > 0
             then coseries bs (d-1)
             else [\_ -> x | x <- bs d]
alts2 cs d = if d > 0
             then coseries (coseries cs) (d-1)
             else [\_ _ -> x | x <- cs d]
```

For programs with many or large datatype definitions, mechanical derivation of `Serial` instances is preferable. The standard patterns are sufficiently regular that they can be inferred by the *Derive* tool (Mitchell and O'Rear 2007), for example.

3.3 Testing

Just as QuickCheck has a top-level function `quickCheck` so SmallCheck has `smallCheck d`.

```
smallCheck  :: Testable a => Int -> a -> IO ()
```

It runs series of tests using depth bounds `0..d`, stopping if any test fails, and prints a summary report. An interactive variant

```
smallCheckI :: Testable a => a -> IO ()
```

invites the user to decide after each completed round of tests, and after any failure, whether to continue.

Example 6 Consider testing the (ill-conceived) property that all Boolean operations are associative.

```
prop_assoc op = \x y z ->
   (x `op` y) `op` z == x `op` (y `op` z)
 where typeInfo = op :: Bool -> Bool -> Bool
```

Testing soon uncovers a failing case:

```
Main> smallCheckI prop_assoc
Depth 0:
  Failed test no. 22. Test values follow.
  {True->{True->True;False->True};
   False->{True->False;False->True}}
  False
  True
  False
```

Being able to generate a series of all (depth-bounded) values of an argument type, SmallCheck can give at least partial information about the extension of a function. □

3.4 Properties & the Pragmatics of Implication

The language of testable properties in SmallCheck is deliberately very close to that in QuickCheck. (It omits operators for gathering statistics about attributes as their main use in QuickCheck is to obtain information about the actual distribution of randomly generated tests.) As in QuickCheck, the `==>` operator can be used to express a *restricting condition* under which a property is expected to hold. Again separate counts are maintained of tests that satisfy the condition and tests that do not, but the operational semantics of `==>` are different. Regardless of the counts, the full (finite) set of tests is applied exhaustively, unless a failing counter-example brings testing to a halt. The following example illustrates an important pragmatic rule for property writers.

Example 2 (revisited) Recall again the type `Prop` of logical propositions. Suppose there are functions

```
eval :: Prop -> Env -> Bool
tautology :: Prop -> Bool
```

where `eval` evaluates the truth of a proposition in a given environment, and `tautology` is some procedure to decide whether a proposition is true in every environment. We expect the following property to hold.

```
prop_tautEval :: Prop -> Env -> Property
prop_tautEval p e = tautology p ==> eval p e
```

However, if the property is so-defined, SmallCheck may test *all possible combinations* of values for `p` and `e`. We are using an *embedded* property language and there is no reflective facility by which SmallCheck can itself discover that the condition depends only on `p`. The following alternative formulation avoids the problem.

```
prop_tautEval' p = tautology p ==> \e -> eval p e
```

Now SmallCheck only tests cases involving the minority of properties `p` for which `tautology p` holds. Although the difference between the two formulations is of no consequence in QuickCheck, as for each test it generates a single random pair of values `p` and `e`, in SmallCheck the second is clearly to be preferred. □

3.5 Existential Properties

SmallCheck extends the property language to permit *existential* quantifiers. Testing a random sample of values as in QuickCheck would rarely give useful information about an existential property: often there is a unique witness and it is most unlikely to be selected at random. But SmallCheck can exhaustively search for a *small* witness. There are several existential variants, but the basic one has the following signature.

```
exists :: (Show a, Serial a, Testable b) =>
  (a -> b) -> Property
```

The interpretation of `exists f` is that for some argument `x` testing the result `f x` succeeds. To illustrate the application of existentials, and some issues with their use, we begin with a reappraisal of previous examples.

Example 1 (revisited) The only property of `isPrefix` specified so far is:

```
prop_isPrefix xs xs' = isPrefix xs (xs++xs')
```

This property is *necessary* but not *sufficient* for a correct `isPrefix`. For example, it holds for the erroneous definition

```
isPrefix []     ys     = True
isPrefix (x:xs) []     = False
isPrefix (x:xs) (y:ys) = x==y || isPrefix xs ys
```

or even for an `isPrefix` that always returns `True`! In terms of the following full specification for `isPrefix`

$$\forall \texttt{xs} \forall \texttt{ys} (\texttt{isPrefix xs ys} \iff \exists \texttt{xs'} (\texttt{xs++xs'} = \texttt{ys}))$$

the partial specification `prop_isPrefix` captures only the $\Longleftarrow$ direction — re-expressing the existential implicitly by the introduction of `xs'` rather than `ys` as the second variable in the property. Viewing `isPrefix` as a decision procedure, `prop_isPrefix` assures its *completeness* but ignores its *soundness*.

Using SmallCheck, we can test for soundness too. The $\Longrightarrow$ direction of the specification can be expressed like this:

```
prop_isPrefixSound xs ys =
  isPrefix xs ys ==>
    exists $ \xs' -> xs++xs' == ys
```

Testing `prop_isPrefixSound` for the erroneous definition of `isPrefix` gives:

```
Main> smallCheckI prop_isPrefixSound
...
Depth 2:
  Failed test no. 11. Test values follow.
  [-1]
  [0]
  non-existence
  Continue?
```

The nearest a QuickCheck user can get to the soundness property is a constructive variant introducing a *Skolem function*, e.g.

```
prop_isPrefixSound' xs ys =
    isPrefix xs ys ==> xs ++ skolem xs ys == ys
  where skolem = drop . length
```

A Skolemised formulation of this kind demands extra information compared to the existential original, making the property harder to read. A more significant drawback is that a suitable Skolem function has to be invented and correctly defined. In this example it is both simple and unique, but that is often not so. □

Example 2 (revisited) For a decision procedure such as

```
satisfiable :: Prop -> Bool
```

we can similarly define a soundness property.

```
prop_satSound p =
  satisfiable p ==> exists $ \e -> eval p e
```

But this time there is no unique Skolem function (`p` may be true in many different environments `e`), nor is there a simple choice of such a function that can be defined as a one-liner. □

Unique Existentials

For some existential properties it is important that there is a unique witness. A formulation based on the equivalence

$$\exists! \texttt{x} (\texttt{P x}) \iff \exists \texttt{x} (\texttt{P x} \wedge \forall \texttt{y} (\texttt{P y} \Rightarrow \texttt{y} = \texttt{x}))$$

would be cumbersome to write, inefficient to test and cannot be used for types outside the `Eq` class, such as functions. So SmallCheck defines the variant `exists1`. When unique existential properties are tested, any failure reports conclude with "non-existence" or "non-uniqueness" followed by two witnesses.

Depth of Existential Searches

The default testing of existentials is bounded by the *same limiting depth* as for universals. This rule has important consequences. A universal property may be satisfied when the depth-bound on test values is shallow but fail when it is deeper. Dually, an existential property may only succeed if the depth-bound on test-values is large enough. So when testing properties involving existentials it can make sense to continue with deeper testing after a shallow failure.

Sometimes the default same-depth-bound interpretation of existential properties can make testing of a valid property fail at all depths. SmallCheck provides *customising* existential quantifiers for use in such circumstances. They take as an additional argument an `Int->Int` function that transforms the depth-bound for testing.

Example 7 The property

```
prop_apex :: [Bool] -> [Bool] -> Property
prop_apex xs ys = exists $ \zs -> zs == xs++ys
```

inevitably fails at all depths greater than zero, but the variant

```
prop_apex' xs ys =
  existsDeeperBy (*2) $ \zs -> zs == xs++ys
```

succeeds at all depths. □

3.6 Dealing with Large Test Spaces

Using the standard generic scheme to define series of test values, it often turns out that at some small depth d the 10,000–100,000 tests are quickly checked, but at depth d+1 it is infeasible to complete the billions of tests. This combinatorial explosion is an inevitable consequence of relentlessly increasing a uniform depth-limit for exhaustive testing. We need ways to reduce some dimensions of the search space so that in other dimensions it can be tested more deeply.

Small Base Types

Although numbers may seem an obvious choice for basic test values, the test-spaces for compound types (and particularly functional types) with numeric bases grow very rapidly as depth increases. For many properties, `Bool` or even `()` is a perfectly sensible choice of type for some variables, greatly reducing the test-space to be covered.

Depth-Adjustment and Filtering

As in QuickCheck *customisation* can be achieved by using specialised test-data generators to redefine the scope of some or all quantified properties. Instead of defining completely fresh generators by *ad hoc* means, there are two natural techniques for adapting the standard machinery. A `series` generator for type `t` is just a function of type `Int -> [t]`. It can be composed to the left of a *depth adjustment* function of type `Int -> Int`, or to the right of a *filtering function* of type `[t] -> [t]`, or both. So although each constructor-layer in an algebraic data value normally adds *one* to the depth, this default is easily over-ridden: we can assign *any* preferred non-negative integer depth to a constructor by composing `cons<N>` and `alts<N>` applications in `Serial` methods with applications of a `depth` function. And if in some context it is appropriate to restrict values to a subseries, a tool-box of list-trimming functions is readily available.

Example 2 (revisited) Both techniques can be illustrated using the `Prop` data-type. Here is a `series` generator adapted so that `Or` constructors have a depth cost of two, and propositions are restricted to two variables only.

```
instance Serial Prop where
  series = take 2 . cons1 Var
         \/         cons1 Not
         \/         cons2 Or . depth 2
```

By adapting a `series` definition in this way, rather than arbitrarily reprogramming it, we can still see easily which `Prop`s are included in the test set. Table 1 shows the effect on the number of tests. □

Depth	Adjustments made			
	None	2-Var	Or-2	Both
1	3	2	3	2
2	15	8	6	4
3	243	74	18	10
4	59295	5552	57	28
5	—	30830258	384	130
6	—	—	3636	916
7	—	—	151095	17818

Table 1. The numbers of `Prop` test cases for depth limits from 1 to 7: the standard default, with at most 2 variables, with `Or` adjusted to depth 2, and with both adjustments.

Constrained Newtypes and Bijective Representations

Frequently, there are restrictions such as data invariants on the appropriate domain of test-cases. As we have seen, such restrictions can often be expressed in antecedent conditions of properties, but that solution does not prevent the generation of useless tests. A natural alternative in a type-driven framework, and one familiar to QuickCheck users, is to define a distinct `newtype`, of tagged values satisfying a restriction, and a *custom generator* for those values.

Example 8 Perhaps the most frequently occurring simple example is that integers must often be limited to the natural numbers. Rather than defining a conditional property such as

```
prop_takeLength :: Int -> [()] -> Property
prop_takeLength i xs =
 i >= 0 ==> length (take i xs) == min i (length xs)
```

assuming a suitable `Serial` instance for a `Nat` type

```
newtype Integral a => N a = N a
type Nat = N Int
```

we can instead define:

```
prop_takeLength' :: Nat -> [()] -> Property
prop_takeLength' (N n) xs =
 length (take n xs) == min n (length xs)
```

Note the appropriate use of unit list elements. □

Section 2.3 pointed out drawbacks of custom generators, such as the difficulty of ensuring correspondence with a restricting predicate. In SmallCheck there is also the requirement for a consistent view of depth. One way to address these issues is to find an auxiliary *representation type* from which there is a bijective mapping to the restricted values. Values of this representation type are generated in the standard way, and used to measure depth. The required correspondence can be expressed as testable properties.

Example 8 (revisited) The `Nat` example is so simple that it hardly needs this technique, but let it serve as a simple first illustration, using a bijection between the natural numbers and unit lists.

```
instance Integral a => Serial (N a) where
  series = map (N . genericLength)
         . (series :: Series [()])
```

We verify that generated `Nats` uniquely represent all and only the non-negative `Int`s by testing the properties

```
prop_natSound :: Nat -> Bool
prop_natSound (N i) = i >= 0

prop_natComp1 :: Int -> Property
prop_natComp1 i =
  i >= 0 ==> exists1 $ \(N n) -> i == n
```

□

d	Number of tests at depth			
	1..5	6..10	11..15	16..20
3	13	1	0	0
4	25	38	0	0
5	30	212	82	0
6	30	511	1132	452

Table 2. The distribution of standard depths for ordered lists generated using a bijection from lists of naturals at depth d.

Example 3 (revisited) As a more typical example, consider a `newtype` for ordered lists of naturals.

```
newtype OrdNats = OrdNats [Nat]
```

We can exploit a bijection between ordered and unordered lists:

```
instance Serial OrdNats where
  series = map (OrdNats . scanl1 plus) . series
    where plus (N a) (N b) = N (a+b)
```

Given that `scanl1 plus` really is a bijection, the number of lists generated at each depth must, of course, be exactly the same as for the default series method. But now every one of them is an ordered list. In place of un-ordered lists, most test cases are now ordered lists that only occur at much greater depth in the default series, beyond feasible reach using brute enumeration. See Table 2. □

For some kinds of properties, testing by brute-enumeration of values using SmallCheck is only feasible with a shallow depth-limit. We have illustrated some customisation techniques that can enable deeper testing, but for a more powerful remedy across the general class of data-driven tests we now turn to *Lazy* SmallCheck.

4. Lazy SmallCheck

A consequence of lazy evaluation in Haskell is that functions can return results when applied to *partially-defined* inputs. To illustrate, consider the following Haskell function `ordered`.

```
ordered [] = True
ordered [x] = True
ordered (x:y:zs) = x <= y && ordered (y:zs)
```

When applied to `1:0:`$\perp$, where $\perp$ is is a call to Haskell's `error` function, `ordered` returns `False`. Indeed, `ordered (1:0:`xs`)` is `False` for *every* xs. Thus, by applying a function to a *single* partially-defined input, one can observe its result over *many* fully-defined ones.

This ability to see the result of a function on many inputs in one go is very attractive to property-based testing: if a property holds for a partially-defined input then it will also hold for all fully-defined refinements of that input. The aim of Lazy SmallCheck is to avoid generating such fruitless refinements.

Lazy SmallCheck is a compatible subset of SmallCheck, currently only capable of checking first-order properties with universal quantifiers. It requires no extensions to standard Haskell other than the ability to detect evaluation of `error`, and this facility is already supported by the main Haskell implementations through imprecise exceptions (Peyton Jones et al. 1999).

4.1 Implication

In SmallCheck and QuickCheck the `==>` operator returns a value of type `Property`, allowing tests falsifying the antecedent to be observed. This facility is less useful in Lazy SmallCheck which tends not to generate many tests falsifying the antecedent, so `==>` simply has the type `Bool -> Bool -> Bool`.

Example 3 (revisited) The property `prop_insertSet` states that `insert` should preserve the `ordered` invariant defined above.

```
prop_insertSet c s =
  ordered s ==> ordered (insert (c :: Char) s)
```

Using SmallCheck, the property can be tested for all inputs up to a given depth using the `depthCheck` function.

```
Main> depthCheck 7 prop_insertSet
Depth 7:
  Completed 109600 test(s) without failure.
  But 108576 did not meet ==> condition.
```

Passing the property to Lazy SmallCheck's `depthCheck` function instead yields

```
Main> depthCheck 7 prop_insertSet
OK, required 1716 tests at depth 7
```

□

Both testing libraries use the same definition of depth, so the input-space checked by each is identical. The difference is that by generating partially-defined inputs, Lazy SmallCheck is able to perform the check with fewer tests. To see why, observe that `prop_insertSet` applied to the partially-defined inputs $\bot$ and `'b':'a':`$\bot$ is `True`. Replacing each $\bot$ with more-defined values is unnecessary because `prop_insertSet` will not look them.

Example 3 will be used throughout the next three sections to illustrate further points about Lazy SmallCheck.

4.2 Laziness is Delicate

The set invariant in the example can be strengthened. Not only should the list representing a set be ordered, but it should also contain no duplicates, as expressed by the function `allDiff`.

```
allDiff [] = True
allDiff (x:xs) = x `notElem` xs && allDiff xs
```

The stronger invariant is

```
isSet s = ordered s && allDiff s
```

and `prop_insertSet` can be modified to use it.

```
prop_insertSet c s = isSet s ==> isSet (insert c s)
```

The `isSet` invariant reduces the number of tests generated by Lazy SmallCheck.

```
Main> depthCheck 7 prop_insertSet
OK, required 964 tests at depth 7
```

This is because some lists satisfy `ordered` but not `allDiff`, so there is increased scope for falsifying the condition without demanding the value of element being inserted.

However, now suppose that the conjuncts of `isSet` are reversed.

```
isSet s = allDiff s && ordered s
```

Checking `prop_insertSet` now requires some twenty times more tests than the version with the original conjunct ordering.

```
Main> depthCheck 7 prop_insertSet
OK, required 20408 tests at depth 7
```

The problem is that `&&` evaluates its left-hand argument first, and `allDiff` is less restrictive than `ordered` in this case.

4.3 Parallel Conjunction

When a property is composed of several sub constraints, like `isSet`, putting the most restrictive one first helps Lazy SmallCheck reduce the number of tests. But it is not always clear what the order should be. In fact, the best order may differ depending on the depth at which the property is checked.

Lazy SmallCheck provides the user with an alternative to normal conjunction called *parallel conjunction* and represented by `*&*`. A parallel conjunction is falsified if *any* of its conjuncts are. This is in contrast to a standard conjunction which returns $\bot$ if its first argument is $\bot$, even if its second is falsified. Replacing `&&` with `*&*` can reduce the need to place the conjuncts in a particular order, and can decrease the number of required tests.

The function `*&*` is defined in a datatype called `Property`, extending `Bool` to allow the distinction between sequential and parallel conjunction. Boolean values must be explicitly lifted to properties. After switching to `*&*` the example property becomes

```
isSet :: Ord a => Set a -> Property
isSet s = lift (ordered s) *&* lift (allDiff s)

prop_insertSet :: Char -> Set Char -> Property
prop_insertSet c s =
  isSet s *=>* isSet (insert c s)
```

(`Property` implication in Lazy SmallCheck is denoted `*=>*`.)

The parallel variant of `isSet` reduces the number of tests compared to either of the non-parallel ones.

```
Main> depthCheck 7 prop_insertSet
OK, required 653 tests at depth 7
```

This is because some lists falsify `ordered` but not `allDiff`, e.g. `1:0:`$\bot$, and vice-versa, some falsify `allDiff` but not `ordered`, e.g. `0:0:`$\bot$.

Now suppose again that the conjuncts are reversed.

```
isSet s = lift (allDiff s) *&* lift (ordered s)
```

This time the number of tests does not change, highlighting that parallel conjunction is not as sensitive to the order of the conjuncts.

```
Main> depthCheck 7 prop_insertSet
OK, required 653 tests at depth 7
```

Despite the advantages of parallel conjunction, it must be introduced manually, with care. An automatic rewrite is not possible, since switching to `*&*` may expose intended partiality in the second conjunct. The first conjunct of `&&` can be used as a guard which assures that the input has a certain property before evaluating the second one. With `*&*` such guards disappear and the property may crash unfairly.

Having to lift Booleans to properties does introduce an unfortunate notational burden. Overloaded Booleans (Augustsson 2007) would be really helpful here.

4.4 Strict Properties

Not all properties are as lazy as `prop_insertSet`. To illustrate, consider the following function that turns a list into a set, throwing away duplicates.

```
set :: Ord a => [a] -> Set a
set = foldr insert []
```

We might like to verify that `set` always returns valid sets.

```
prop_set :: [Char] -> Bool
prop_set cs = isSet (set cs)
```

To return `True`, `prop_set` demands the entire input, so there is no scope for the property to be satisfied by a partially-defined input. Checking with SmallCheck yields

```
Main> depthCheck 6 prop_set
Depth 6:
  Completed 1957 test(s) without failure.
```

and with Lazy SmallCheck:

```
Main> depthCheck 6 prop_set
OK, required 2378 tests at depth 6
```

Not only is Lazy SmallCheck of no benefit in this case, but it is worse than SmallCheck because it fruitlessly generates some partially-defined inputs as well as all the totally-defined ones.

4.5 Serial Types

Lazy SmallCheck also provides a `Serial` class with a `series` method. But now a series has the type

```
type Series a = Int -> Cons a
```

From the users perspective, `Cons a` is an abstract type storing instructions on how to construct values of type `a`. It has the following operations.

```
cons  :: a -> Series a
empty :: Series a
(\/)  :: Series a -> Series a -> Series a
(><)  :: Series (a -> b) -> Series a -> Series b
```

Unlike in SmallCheck, the `><` operator represents application rather than cross product. To illustrate, SmallCheck's `cons<N>` family of operators is defined in Lazy SmallCheck in the following fashion.

```
cons0 f = cons f
cons1 f = cons f >< series
cons2 f = cons f >< series >< series
```

So SmallCheck `Serial` instances defined using the standard pattern are written identically in Lazy SmallCheck.

Depth Customisation

The left-associative `><` combinator implicitly takes a depth `d`, and passes `d` to its left argument and `d-1` to its right argument. The result is that each child of a constructor is given depth `d-1`, like in SmallCheck. If the depth argument to `><` is zero, then no values can be constructed.

Example 9 Suppose a generator for rose trees (Bird 1998) is to be written.

```
data Rose a = Node a [Rose a]
```

The standard list generator might be deemed inappropriate to generate the children of a node, because each child would be generated to a different depth. Instead, the programmer might write

```
instance Serial a => Serial (Rose a) where
  series = cons Node >< series >< children
```

where `children` generates a list of values, each of which is bounded by the same depth parameter.

```
children d = list d
  where
    list = cons []
        \/ cons (:) >< const (series (d-1)) >< list
```

□

Primitive Types

Like in SmallCheck, a series can be defined as a finite list of finite *fully-defined* candidate values. This is achieved using the `drawnFrom` combinator.

```
drawnFrom :: [a] -> Cons a
drawnFrom xs = foldr (\/) empty (map cons xs) 0
```

The depth parameter `0` is irrelevant in the above definition, as it is not inspected by any of the combinators used.

Example 10 Here is the `Serial` instance for `Int`.

```
instance Serial Int where
  series d = drawnFrom [-d..d]
```

Using `drawnFrom`, primitive values of type `Integer`, `Char`, `Float` and `Double` are generated just as they are in SmallCheck. □

4.6 Implementation

This section presents the Lazy SmallCheck implementation. Only code for parallel conjunction, the `Testable` class, and for displaying counter-examples and counting tests is omitted.

Partially-defined Inputs

The central idea of Lazy SmallCheck is to generate *partially-defined* inputs, that is, inputs containing some calls to `error`. An example of a partially-defined input of type `Prop` is

```
Or (Or (Var Q) (Not (error "_|_"))) (error "_|_")
```

Using imprecise exceptions (Peyton Jones et al. 1999), one can apply a property to the above term and observe whether it evaluates to `True`, `False`, or `error "_|_"`. However, since the input contains several calls to `error "_|_"`, it cannot be determined which one was demanded by the program. This is the motivation for tagging each `error` with its *position* in the tree-shaped term. A position is a list of integers, uniquely describing the path from the root of the term to a particular sub-term.

```
type Pos = [Int]
```

For example, the position `[1,0]` refers to the 0^{th} child of the root constructor's 1^{st} child. Lazy SmallCheck encodes such positions in the string passed to `error`. Using the helper function

```
hole :: Pos -> a
hole p = error (sentinel : map toEnum p)
```

the above example term of type `Prop` is now represented as follows.

```
Or (Or (Var Q) (Not (hole [0,1,0]))) (hole [1])
```

Each argument to `error` is prefixed with a `sentinel` character, allowing holes to be distinguished from possible calls to `error` occurring in the property.

```
sentinel :: Char
sentinel = '\0'
```

Answers

The data type `Answer` is used to represent the result of a property applied to a partially-defined input.

```
data Answer = Known Bool | Unknown Pos
```

Using imprecise exceptions, the following function turns a `Bool` into an `Answer`.

```
answer :: Bool -> IO Answer
answer a =
  do res <- try (evaluate a)
     case res of
       Right b -> return (Known b)
       Left (ErrorCall (c:cs)) | c==sentinel ->
         return (Unknown (map fromEnum cs))
       Left e -> throw e
```

The functions `try`, `evaluate`, and `throw` are all exported by Haskell's `Control.Exception` library: `evaluate` forces evaluation of the Boolean value passed to it, before returning it in an `IO` action, and `try` runs the given `IO` action, and returns a `Right` constructor containing the action's result if no exception was raised, otherwise it returns a `Left` constructor containing the exception. If the exception represents a hole, then the position of demand is extracted and returned. Otherwise the exception is re-thrown.

When a property applied to a term yields `Unknown pos`, Lazy SmallCheck *refines* the term by defining it at position `pos`.

Refinement

Looking under the hood, the `Cons` data type is a little more complicated than the simple list it replaces in SmallCheck. Lazy SmallCheck must not only generate inputs but also take an existing input and refine it at a particular position.

```
data Cons a = Type :*: [[Term] -> a]
```

This data type can be read as follows: to construct a value of type `a`, one must have a sum-of-products representation of the type,

```
data Type = SumOfProd [[Type]]
```

and a list of conversion functions (one for each constructor) from a list of *universal* terms (representing the arguments to the constructor) to an actual value of type `a`. A universal term is either a constructor with an identifier and a list of arguments, or a hole representing an undefined part of the input.

```
data Term = Ctr Int [Term] | Hole Pos Type
```

Working with universal terms, the refinement operation can be defined generically, once and for all: it walks down a term following the route specified by the position of demand,

```
refine :: Term -> Pos -> [Term]
refine (Ctr c xs) (i:is) =
    map (Ctr c) [ls ++ y:rs | y <- refine x is]
  where (ls, x:rs) = splitAt i xs
refine (Hole p (SumOfProd sop)) [] = new p sop
```

and when it reaches the desired position, a list of constructors of the right type is inserted, each of which is applied to undefined arguments.

```
new :: Pos -> [[Type]] -> [Term]
new p sop =
  [ Ctr c (zipWith (\i -> Hole (p++[i])) [0..] ts)
  | (c, ts) <- zip [0..] sop ]
```

Series Combinators

The `Series` combinators `cons`, `empty`, `\/` and `><` are defined in Figure 1, along with two auxiliary functions. The `conv` auxiliary allows a conversion function of type `Term -> a` to be obtained from the second component of a `Cons a` value. The `nonEmpty` auxiliary is used to ensure that a partially-defined value is not generated when there is no fully-defined refinement of that value within the depth limit.

```
cons :: a -> Series a
cons a d = SumOfProd [[]] :*: [const a]

empty :: Series a
empty d = SumOfProd [] :*: []

(\/) :: Series a -> Series a -> Series a
(a \/ b) d = SumOfProd (psa ++ psb) :*: (ca ++ cb)
  where SumOfProd psa :*: ca = a d
        SumOfProd psb :*: cb = b d

(><) :: Series (a -> b) -> Series a -> Series b
(f >< a) d =
    SumOfProd [ta:p | notTooDeep, p <- ps] :*: cs
  where SumOfProd ps :*: cfs = f d
        ta :*: cas = a (d-1)
        cs = [ \(x:xs) -> cf xs (conv cas x)
             | notTooDeep, cf <- cfs ]
        notTooDeep = d > 0 && nonEmpty ta

nonEmpty :: Type -> Bool
nonEmpty (SumOfProd ps) = not (null ps)

conv :: [[Term] -> a] -> Term -> a
conv cs (Hole p _) = hole p
conv cs (Ctr i xs) = (cs !! i) xs
```

Figure 1. Lazy SmallCheck's `Series` combinators.

Refutation Algorithm

The algorithm to refute a property takes two parameters, the property to refute and an input term, and behaves as follows.

```
refute :: (Term -> Bool) -> Term -> IO ()
refute p x = do
  ans <- answer (p x)
  case ans of
    Known True  -> return ()
    Known False -> putStrLn "Counter example found"
                >> exitWith ExitSuccess
    Unknown pos -> mapM_ (refute p) (refine x pos)
```

A simple variant of Lazy SmallCheck's `depthCheck` function can be now be defined.

```
check :: Serial a => Int -> (a -> Bool) -> IO ()
check d p = refute (p . conv cs) (Hole [] t)
  where t :*: cs = series d
```

For simplicity of presentation, these two definitions do not attempt to print counter examples, count the number of tests performed, or support checking of multi-argument properties.

Parallel Conjunction

Parallel conjunction is a straightforward extension to the refutation algorithm. The main difference is that answers contain values of type `Property` rather than `Bool`. Internally, a `Property` is just a representation of a logical formula. To evaluate a `Property` of the form `p *&* q`, `p` is evaluated first and if it is unknown, then `q` is also evaluated, *without* refining the input as demanded by `p`. If `p` or `q` evaluates to `False` then the value of the whole conjunction is taken to be `False`. If both `p` and `q` are unknown, then the input is refined at the position demanded by `p`. This means that the number of tests generated can decrease when switching from `&&` to `*&*`, but never increase. There is however an evaluation overhead when `p` is unknown, because `*&*` will evaluate `q` in this case and `&&` will not.

Variations

Two alternative implementations of Lazy SmallCheck have been explored, both avoiding repeated conversion of universal terms to Haskell values of a particular type. One uses `Data.Generics` and only works in GHC, while the other requires an extra method in the `Serial` class so that refinement can be defined on a per-type basis. These variants are more efficient, but by no more than a factor of three in our experience. The implementation presented here has the advantage of giving depth and generation control to the programmer in a simple manner that is largely compatible with the core SmallCheck subset.

5. Comparative Evaluation

Previous sections have included some in-principle comparisons between the three libraries. This section presents some quantitative results. Table 3 shows the runtimes of several example properties tested to varying depths with SmallCheck and Lazy SmallCheck. QuickCheck is not represented in this table because it does not have the same notion of a depth bound. However, the time taken by QuickCheck to refute an invalid property can be meaningfully compared with that taken by SmallCheck and Lazy SmallCheck; such timings are noted in the discussion.

All the example properties are first-order and universally-quantified. All test generators are written using the simple standard pattern, with no customisation. The following paragraphs discuss the results, focusing on some of the more interesting examples.

RedBlack The RedBlack program is an implementation of sets using ordered Red-Black trees, taken from (Okasaki 1999). A fault was fabricated in the rebalancing function by swapping two subtrees. This is where most of the complexity in the implementation lies and is a likely source of a programming mistake. Okasaki's tree representation is as follows.

```
data Colour = R | B
data Tree a = E | T Colour (Tree a) a (Tree a)
```

A predicate defining ordered Red-Black trees was added, capturing three things: that trees are ordered (the `ord` invariant), that no red node has a red parent (the `red` invariant), and that every path from the root to an empty node contains the same number of black nodes (the `black` invariant).

```
redBlack t = ord t && black t && red t
```

The following property was also added.

```
prop_insertRB :: Int -> Tree Int -> Bool
prop_insertRB x t =
  redBlack t ==> redBlack (insert x t)
```

No counter example was found within 20 minutes of testing at depth 4 using SmallCheck. QuickCheck, with simple random generation of trees, did not find a counter example after 100,000 batches of 1000 tests (amounting to 32 minutes of testing). Testing with Lazy SmallCheck revealed the fault in a fraction of a second at depth 4, and with the fault removed, verified the property at depth 4 within 7 seconds.

The number of tests is a few times lower when using parallel conjunction inside the `redBlack` invariant. However, in this case the evaluation overhead of using `*&*` is substantial and cancels the benefit of fewer tests.

Huffman The Huffman program is taken from (Bird 1998). It contains functions for both compression and decompression of strings, along with a function for building Huffman trees. A Huffman tree is a binary tree with symbols at its leaves, and the path from the root to any symbol describes the unique, variable-length sequence of zeros and ones representing that symbol.

Property		**Depth**				
		3	**4**	**5**	**6**	**7**
RedBlack	L	0.03	⋆0.15			
	S	0.20	×			
Turner	L	0.01	0.47	×		
	S	0.01	0.07	×		
SumPuz	L	0.05	3.68	421.80	×	
	S	0.05	4.48	682.86	×	
Huffman_2	L	0	0	0.63	22.9	×
	S	0	0.01	7.65	×	
Countdown_1	L	0.01	0.14	2.27	39.3	800.4
	S	0.05	17.43	×		
Countdown_2	L	0.01	1.23	666.95	×	
	S	0.01	1.44	737.10	×	
Circuits_2	L	0	0.01	0.01	0.03	0.06
	S	0	0.01	0.52	63.80	×
Circuits_3	L	0.06	13.28	×		
	S	0.02	5.08	×		
Catch	L	0.07	6.22	830.02	×	
	S	0.02	88.23	×		
Mate	L	0	0.37	⋆29.87		
	S	0.06	×			
		Depth				
		7	**8**	**9**	**10**	**11**
ListSet	L	0.01	0.02	0.03	0.06	0.13
	S	0.05	0.39	4.06	694.10	×
Huffman_1	L	0.27	2.76	27.57	315.81	×
	S	0.08	0.73	7.69	90.38	×
Circuits_1	L	0.06	0.29	1.62	10.06	70.44
	S	0.04	0.20	1.21	8.38	65.88

Key: ⋆ Counter example found; × Longer than 20 minutes; L Lazy SmallCheck; S SmallCheck

Table 3. Times to check benchmark properties using SmallCheck and Lazy SmallCheck at various depths.

Two properties were added to Bird's program. The first states that the decompresser (`decode`) is the inverse of the compressor (`encode`).

```
prop_decEnc cs =
    length ft > 1 ==> decode t (encode t cs) == cs
  where ft = collate cs
        t  = mkHuff ft
```

Here, `collate` builds a frequency table (`ft`) for an input string, and `mkHuff` builds a Huffman tree from a frequency table.

The second property asserts that `mkHuff` produces *optimal* Huffman trees, that is, for all binary trees t, if t is a Huffman tree then it has a *cost* no less than that produced by `mkHuff`. A binary tree is only a Huffman tree (as determined by `isHuff`) if it contains every symbol in the source text exactly once. The cost of a Huffman tree is defined as the sum of each symbol's frequency multiplied by its depth in the Huffman tree.

```
prop_optimal cs t =
   isHuff t cs ==> cost ft t >= cost ft (mkHuff ft)
  where ft = collate cs
```

In checking the first property, SmallCheck was more efficient than Lazy SmallCheck by a constant factor of 3. This property is hyper-strict for most inputs. On the second property, due to the

condition that input trees must be Huffman trees, Lazy SmallCheck allowed testing to one level deeper within the 20 minute cut-off.

Turner The Turner program is a compiler from lambda expressions to Turner's combinators, as defined by in (Peyton Jones 1987). In particular, it provides a function (`abstr`) to abstract a free variable from an expression by introducing combinators from a known, fixed set. The property of interest is Turner's law of abstraction (Turner 1979), stating that if a variable is abstracted from an expression, and the resulting expression is applied to that variable, then one ends up with the original expression again.

```
prop_abstr v e = reduce (abstr v e :@ V v) == e
```

Here, `:@` is function application and `reduce` applies combinator reduction rules to a given expression. This property can only return `True` after demanding the whole input so, due to strictness, SmallCheck has an advantage, this time by a factor of 7 at depth 4.

Mate The Mate program solves mate-in-N chess problems. It represents a chess board as two lists, the first containing white's piece-position pairs and second containing black's.

```
data Board  = Board [(Kind,Square)] [(Kind,Square)]
data Kind   = King   | Queen  | Rook
            | Bishop | Knight | Pawn
type Square = (Int,Int)
data Colour = Black | White
```

It includes a function `checkmate` returning whether or not a given colour is checkmated on a given board. A property was added stating that for all chess boards *b*, if *b* is a *valid board* and white has only a king and a pawn, then black cannot be in checkmate.

```
prop_checkmate b@(Board ws bs) =
   ( length ws == 2
  && Pawn `elem` map fst ws
  && validBoard b
   ) ==> not (checkmate Black b)
```

A valid board is one satisfying a number of healthiness criteria, such as each side has exactly one king, kings cannot be placed on touching squares, and no two pieces can occur on the same square.

Neither SmallCheck at depth 4 after 20 minutes, nor QuickCheck with a 100,000 batches of 1000 random tests after 18 minutes, revealed a counter example. Lazy SmallCheck within 30 seconds at depth 5 produces

```
Counter example found:
Board [(King,(3,2)),(Pawn,(2,1))]
      [(Queen,(1,3)),(King,(1,2)),(Bishop,(1,1))]
```

The order of conjuncts in the property has a significant impact on performance, and a lot of experimentation was required to find the best order. The time taken to find a counterexample was more than 20 minutes if the order was unfortunately chosen. However, using parallel conjunction, no ordering required more than 22 seconds to find a counter example.

Other Examples The remaining examples follow a similar pattern. SmallCheck is more effective on strict properties and Lazy SmallCheck wins for lazy ones. Of these examples, ListSet is the set implementation using ordered lists (along with the insertion property) given earlier, Countdown is a solver for a popular numbers game (along with a lemma and a refinement theorem) taken from (Hutton 2002), SumPuz is a cryptarithmetic solver (with a soundness property) from (Claessen et al. 2002), Circuits is part of a library from the Reduceron (Naylor and Runciman 2008), and Catch is a specification (with a soundness property) for part of the Catch tool (Mitchell and Runciman 2007).

Summary of Results In two of the thirteen example properties, Lazy SmallCheck found a counter example in good time, and SmallCheck and QuickCheck did not. In five, Lazy SmallCheck permitted deeper checking than SmallCheck, and in five others, SmallCheck had a constant factor advantage over Lazy SmallCheck, ranging from a small 1.04 factor to a more significant 7.

Five of the example properties have an implication where the condition is composed of several conjuncts, and could potentially be improved by using parallel conjunction. In two of these, parallel conjunction had no impact on the number of tests, but neither did it introduce a significant evaluation overhead. In another, the number of tests was reduced, but this was cancelled out by the evaluation overhead. And in another, parallel conjunction reduced the runtime by up to a factor of three for some conjunct orderings, but had no effect on others. In the remaining example, the use of parallel conjunction eliminated the need to put a long series of conjuncts in a particular order for a counter example to be found.

6. Related work

Needed Narrowing Lazy SmallCheck's refutation algorithm is closely related to *needed narrowing* (Antoy et al. 1994), an evaluation strategy used by some functional-logic languages, including Curry (Hanus), and some Haskell analysis tools (Lindblad 2008) (Naylor and Runciman 2007). Like Lazy SmallCheck, needed narrowing allows functions to be applied to partially-defined inputs, but this is achieved using logical variables rather than calls to error. As needed narrowing is designed for functional-logic programs, it also deals with non-deterministic functions.

A typical implementation of needed narrowing stores the partially evaluated result after each test, and resumes the evaluation after refining the input. Lazy SmallCheck instead evaluates the property from scratch every time an undefined part of the input is demanded. This means that needed narrowing is more efficient. For *small inputs*, it would be interesting to explore just how big (or small) this benefit is.

Residuation Parallel conjunction is related to *residuation*, another evaluation strategy used by some functional-logic languages including Curry (Hanus) and Escher (Lloyd 1999). Under residuation, if the value of a logical variable is demanded by some logical conjunct in the system, then that conjunct *suspends* on the variable, and another conjunct is evaluated. If evaluation of this second conjunct happens to instantiate the variable suspended on by the first, then the first conjunct is resumed.

In parallel conjunction, when evaluation of the first conjunct calls error, the second conjunct is immediately evaluated on the *same* input. If the second conjunct also calls error then the input is refined. Therefore, a parallel conjunction of the form `p *&* q` is similar to evaluating `p` by residuation and `q` by narrowing. The end result in both cases is that if either conjunct is falsified, then so is the whole conjunction. There is no need for resumption and suspension mechanisms in Lazy SmallCheck because it evaluates the conjunction from scratch every time a refinement is made.

Gast Gast (Koopman et al. 2002) is a library for property-based testing in Clean. It exploits Clean's generic programming features to offer a default test-generator for all user-defined types. Like SmallCheck, it generates fully-defined and finite values. Unlike SmallCheck, it employs a blend of random and systematic generation. Constructors of an algebraic data type are selected at random, and duplicate tests are avoided by keeping a record of which inputs have been tried already. The authors also mention testing of existential properties, but without giving details.

EasyCheck EasyCheck (Christiansen and Fischer 2008) is another testing library, written in Curry. Like Lazy SmallCheck it employs narrowing to achieve property driven generation of data. The library makes use of the data refinement and narrowing mechanisms built into Curry. It provides a number of combinators for expressing properties about non-deterministic functions. Apart from this, the main difference is that EasyCheck uses *level diagonalisation*, which has the advantage that it allows systematic generation of deep and shallow inputs in a fair order. There are also some disadvantages of level diagonalisation: any counter examples produced are not necessarily minimal, and it is not clear to the programmer which inputs have been tested and which have not.

7. Future Work and Conclusions

Future Work It would be interesting to investigate higher-order properties and existential quantifiers in the context of Lazy SmallCheck. It would also be interesting to compare Lazy SmallCheck with a full-strength narrowing implementation, such as the Münster Curry Compiler (Lux 2003). This would help establish whether it is worth adding narrowing to an existing Haskell compiler to aid property-based testing, or whether lazy evaluation and imprecise exceptions already provide most of the benefit. Another avenue for investigation would be the ability to import QuickCheck, SmallCheck, and Lazy SmallCheck in a program and test the same properties using any tool.

Conclusions If a property is refuted by SmallCheck then a *simplest* counter example is reported, and such a counter example is usually the easiest to investigate. Alternatively, if a property is not refuted then a *clearly-defined* portion of the input space on which it holds is reported, and this knowledge is valuable in judging the effectiveness of testing. In each case the SmallCheck user learns something useful that the QuickCheck user would not. Furthermore, the SmallCheck user can (1) write data generators easily using a simple standard pattern; (2) view counter examples of higher-order properties; and (3) enjoy a richer specification language supporting (unique) existential quantification.

Using Lazy SmallCheck, the programmer can specify conditional properties as simple logical implications and typically have a plentiful supply of condition-satisfying inputs generated automatically. This is thanks not just to Haskell's lazy evaluation strategy, which can compute well-defined outputs for partially-defined inputs, but also to parallel conjunction. Parallel conjunction reduces the need for programmers to tweak conjunct orderings in properties in order to obtain the maximum benefit of Lazy SmallCheck. Of course, it is very difficult to say how often conditional properties occur in general, but they arose quite readily in our thirteen benchmark properties, the majority of which were taken from existing programs described in the literature. In seven of the thirteen properties, Lazy SmallCheck allowed deeper testing than SmallCheck, and in two of these, counter examples were revealed that were simply infeasible to find using QuickCheck and SmallCheck, at least without writing a custom generator.

Although SmallCheck and Lazy SmallCheck are sometimes more effective than QuickCheck, the reverse is also true. For example, as part of his ICFP'07 invited talk, Hughes tested an SMS message-packing program using QuickCheck. QuickCheck uncovered a bug when packing messages of multiple-of-eight length. Such large, strictly-demanded messages would be outside the reach of SmallCheck and Lazy SmallCheck.

Put simply: SmallCheck, Lazy SmallCheck and QuickCheck are complementary approaches to property-based testing in Haskell.

Availability

SmallCheck and Lazy SmallCheck are freely available from `http://hackage.haskell.org/`.

Acknowledgments

We thank the reviewers for their helpful feedback. Colin Runciman wrote the SmallCheck prototype during a visit to Galois in 2006. Matthew Naylor was supported by an EPSRC studentship.

References

Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *POPL'94*, pages 268–279, 1994.

Lennart Augustsson. Overloaded Booleans. `http://augustss.blogspot.com/`, 2007.

Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.

Jan Christiansen and Sebastian Fischer. Easycheck – test data for free. In *FLOPS'08*, pages 322–336. LNCS 4989, 2008.

K. Claessen, C. Runciman, O. Chitil, R. J. M. Hughes, and M. Wallace. Testing and tracing lazy functional programs using QuickCheck and Hat. In *AFP'02* , pages 59–99. Springer LNCS 2638, 2002.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP'00*, pages 268–279. ACM SIGPLAN, 2000.

Andy Gill and Colin Runciman. Haskell program coverage. In *Haskell'07*, pages 1–12. ACM, 2007.

Michael Hanus. Curry: An Integrated Functional Logic Language. Language report (March 2006), available online at `http://www.informatik.uni-kiel.de/~curry,/report.html`.

Graham Hutton. The countdown problem. *Journal of Functional Programming*, 12(6):609–616, November 2002.

Daniel Jackson. *Software abstractions: logic, language and analysis*. The MIT Press, 2006.

Pieter W. M. Koopman, Artem Alimarine, Jan Tretmans, and Marinus J. Plasmeijer. Gast: Generic automated software testing. In *IFL'02*, pages 84–100. LNCS 2670, 2002.

Fredrik Lindblad. Property directed generation of first-order test data. In *TFP'07*, volume 8, pages 105–123. Intellect, 2008.

John W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3), 1999.

Wolfgang Lux. The Münster Curry Compiler. `http://danae.uni-muenster.de/~lux/curry/`, 2003.

Neil Mitchell and Stefan O'Rear. Derive - project home page. `http://www.cs.york.ac.uk/~ndm/derive/`, March 2007.

Neil Mitchell and Colin Runciman. A static checker for safe pattern matching in Haskell. In *TFP'05*, volume 6, pages 15–30. Intellect, 2007.

Matthew Naylor and Colin Runciman. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. In *IFL'07*, page to appear. LNCS, 2008.

Matthew Naylor and Colin Runciman. Finding inputs that reach a target expression. In *SCAM'07*, pages 133–142. IEEE Computer Society, 2007.

Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Computer Science. Prentice-Hall, 1987.

Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *PLDI'99*, pages 25–36. ACM, 1999.

D. A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9(1):31–49, 1979.