SmallCheck and Lazy SmallCheck
automatic exhaustive testing for small values

Colin Runciman\textsuperscript{1}   Matthew Naylor\textsuperscript{1}   Fredrik Lindblad\textsuperscript{2}

\textsuperscript{1}University of York, UK

\textsuperscript{2}Chalmers University / University of Gothenburg, Sweden
Motivation
Small Scope Hypothesis

Common Observation
If a program fails to meet its specification in some cases, it *almost always* fails in some *simple* case.
Small Scope Hypothesis

Common Observation
If a program fails to meet its specification in some cases, it *almost always* fails in some *simple* case.

Contrapositive Corollary
If a program does not fail in any simple case, it *hardly ever* fails in *any* case.
Success of QuickCheck

QuickCheck (Claessen & Hughes, ICFP’00):

- A combinator library for *random testing*.
- Exploits *type classes* to generate test values.
- Checks *universally quantified* properties.
- Reports *counter-example* if found, or N tests OK.
- Widely used; *often reported effective.*
Drawbacks of QuickCheck

Principally:

- If failing cases are rare, none may be tested even though some of them are very simple.

Also:

- Counter-examples are random not minimal.
- Some properties have conditions hard to satisfy.
- Writing good custom generators can be tricky.
- No assurance of test-space coverage.
- No support for existential properties.
- Counter-examples that are functions are not displayed.
Property-based Testing and QuickCheck

- Arbitrary types have random-value generators.
- Testable types represent properties.

```haskell
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

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

a class-polymorphic test-driver.
Consider a function:

\[
is\text{Prefix} :: \text{Eq } a \Rightarrow [a] \to [a] \to \text{Bool}
\]
Example

- Consider a function:
  
  ```haskell
  isPrefix :: Eq a => [a] -> [a] -> Bool
  ```

- Specify an expected property:
  
  ```haskell
  prop_isPrefix :: [Int] -> [Int] -> Bool
  prop_isPrefix xs xs' = isPrefix xs (xs++xs')
  ```
Consider a function:

\[
\text{isPrefix} :: \text{Eq } a \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}
\]

Specify an expected property:

\[
\text{prop_isPrefix} :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool}
\]

\[
\text{prop_isPrefix } xs \; xs' = \text{isPrefix } xs \; (xs ++ xs')
\]

Test it automatically:

> quickCheck prop_isPrefix

OK, passed 100 tests.

Or if \text{isPrefix} interprets arguments the other way round:

Falsifiable, after 1 tests:

\[
[1]
\]

\[
[2]
\]
Arbitrary User-defined Types

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

Defining a generator for such a recursive data type requires careful use of controlling numeric parameters.

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)))]
Conditional Properties and Custom Generators

- QuickCheck defines an implication operator
  \((\Rightarrow) : \text{Testable } a \Rightarrow \text{Bool } \rightarrow a \rightarrow \text{Property}\)
  where Property is a new Testable type.

- For example:
  ```haskell
  type Set a = [a]
  insert :: Ord a => a -> Set a -> Set a

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

- To avoid useless *unordered* lists, use a *custom generator*. But there are drawbacks: (1) defining it; (2) verifying it.
Small Data Values

Algebraic data types

- Small bound on the \textit{depth of constructor nesting}.
  Eg. $\text{Or} \ (\text{Not} \ (\text{Var} \ P)) \ (\text{Var} \ Q)$ has depth 3.

Tuples

- Depth is the maximum component depth.

Numbers

- The depth of an \textit{integer} $i$ is its absolute value.
  (cf. $\text{Succ}^i \ \text{Zero}$).
- The depth of a \textit{floating point} number $s \times 2^e$ is the depth of the integer pair $(s,e)$. Eg. the floating point numbers of depth $\leq 2$ 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 with data arguments

- Bound the depth of the body — treating case like a constructor with its alternatives as components.

Eg. The Bool→Bool functions of depth 1 are:

\[ \begin{align*}
\lambda b & \rightarrow \text{case } b \text{ of } \{ \text{True } \rightarrow \text{ True } \ ; \ \text{False } \rightarrow \text{ True } \} \\
\lambda b & \rightarrow \text{case } b \text{ of } \{ \text{True } \rightarrow \text{ True } \ ; \ \text{False } \rightarrow \text{ False } \} \\
\lambda b & \rightarrow \text{case } b \text{ of } \{ \text{True } \rightarrow \text{ False } \ ; \ \text{False } \rightarrow \text{ True } \} \\
\lambda b & \rightarrow \text{case } b \text{ of } \{ \text{True } \rightarrow \text{ False } \ ; \ \text{False } \rightarrow \text{ False } \}
\end{align*} \]

Functions with functional arguments

- Defined generically — thank you Ralf!
Serial Types

A series is a function from depths to finite value-lists.

\[
\text{type Series } a = \text{Int } \rightarrow \ [a] \\
\]

A Serial type is one with a series method.

\[
\text{class Serial } a \text{ where} \\
\text{series } :: \text{Series } a \\
\]

Sums and products are simply defined (no diagonalisation):

\[
(\bigvee) :: \text{Series } a \rightarrow \text{Series } a \rightarrow \text{Series } a \\
\text{s1 } \bigvee \text{ s2 } = \text{\ d } \rightarrow \text{ s1 } \text{ d } ++ \text{ s2 } \text{ d} \\
\]

\[
(\bigotimes) :: \text{Series } a \rightarrow \text{Series } b \rightarrow \text{Series } (a, b) \\
\text{s1 } \bigotimes \text{ s2 } = \text{\ d } \rightarrow \ [(x,y) \mid x \leftarrow \text{ s1 } \text{ d}, y \leftarrow \text{ s2 } \text{ d}] \\
\]
Defining Serial Instances

- Instances are predefined for Prelude types.
- Instances for new algebraic types follow a simple pattern. The series method uses generic \( \vee \) and \( \text{cons}<N> \) combinators.

```haskell
instance Serial Prop where
  series = \text{cons1} \text{Var} \vee \text{cons1} \text{Not} \vee \text{cons2} \text{Or}
```

- The coseries method, generating functions, uses generic \( \text{alts}<N> \) combinators to generate case alternatives.
- The Derive tool automates instance definition — thank you Neil and Stefan!
Are all binary operations on `Bool` associative?

```haskell
prop_assoc op = \x y z ->
  (x 'op' y) 'op' z == x 'op' (y 'op' z)
where typeInfo = op :: Bool -> Bool -> Bool
```
Partial Extensions of Functional Values

- Are all binary operations on Bool associative?

  \[
  \text{propassoc } \text{op} = \lambda x \ y \ z \rightarrow \\
  (x \ '\text{op}' \ y) \ '\text{op}' \ z = x \ '\text{op}' \ (y \ '\text{op}' \ z) \\
  \text{where } \text{typeInfo} = \text{op} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
  \]

- Testing finds and displays a failing case:

  \[
  \text{Main> smallCheckI prop_assoc} \\
  \text{Depth 0:} \\
  \text{Failed test no. 22. Test values follow.} \\
  \{\text{True->\{True->True;False->True\};} \\
  \text{True->\{True->False;False->True\}}\} \\
  \text{False} \\
  \text{True} \\
  \text{False}
  \]
Existential Properties

▶ Testing exists \( f \) succeeds if for some small argument \( x \) testing \( f \ x \) succeeds.

\[ \text{exists :: (Show a, Serial a, Testable b) => (a -> b) -> Property} \]

Uniqueness

▶ Properties written using the translation

\[ (\exists! x(P \ x)) \iff (\exists x(P \ x)) \land (\forall y(P \ y \Rightarrow y = x)) \]

are awkward to write & read, inefficient to test and limited to \( \text{Eq} \) types. A variant \text{exists1} requires a \text{unique witness}.

Depth

▶ A universal property may pass shallow tests but fail deeper ones. An existential property may fail shallow tests but pass deeper ones. The variant \text{existsDeeperBy dt} specifies in \( \text{dt::Int->Int} \) a \text{depth transformer}.
Consider the isPrefix specification:
\[ \forall xs \forall ys (\text{isPrefix } xs \ ys \iff \exists xs' (xs++xs' = ys)) \]

prop_isPrefix captures the \(\iff\) direction, but what about the \(\Rightarrow\) direction?

prop_isPrefixSound \(xs \ ys =\)
\[ \text{isPrefix } xs \ ys \Rightarrow \exists \ xs' \ (xs++xs' = ys) \]

A QuickCheck user could write

prop_isPrefixSound' \(xs \ ys =\)
\[ \text{isPrefix } xs \ ys \Rightarrow xs ++ \text{skolem } xs \ ys = ys \]
where \(\text{skolem} = \text{drop} \ . \ \text{length}\)

but skolem has to be invented and defined — rarely so simple.
Dealing with Large Test Spaces

Depth-Adjustment and Filtering

- Generators of type \( \text{Int} \rightarrow [t] \) compose with depth adjustment functions of type \( \text{Int} \rightarrow \text{Int} \), or with filtering functions of type \([t] \rightarrow [t]\). Eg:

```haskell
instance Serial Prop where
    series = take 2 . cons1 Var
             \/
             cons1 Not
             \/
             cons2 Or . depth 2
```
Dealing with Large Test Spaces

Depth-Adjustment and Filtering

- Generators of type \( \text{Int} \rightarrow [t] \) compose with depth adjustment functions of type \( \text{Int} \rightarrow \text{Int} \), or with filtering functions of type \([t] \rightarrow [t] \). Eg:

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

Bijective Representations

- Impose data invariants by using testable bijections from a shallower representation. Eg:

  ```haskell
  instance Serial OrdNats where
    series = map (OrdNats . scanl1 plus) . series
  ```
Lazy SmallCheck
Partial Values and Refinements

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

- If we evaluate ordered 1:0:⊥ it reduces to False. We conclude that ordered 1:0:xs is False for every xs.
- By applying a function to a single partially-defined input, we deduce its result over many fully-defined ones.
- If a property holds for some a partially-defined argument value then it holds for all refinements of it.
- Lazy SmallCheck uses this fact to prune the test space for first-order, universal properties.
Example Revisited

\[
\text{prop\_insertSet } c \ s = \\
\text{ordered } s \Rightarrow \text{ordered (insert } c \ s) \\
\]

- Testing with SmallCheck:

  Main> depthCheck 7 prop\_insertSet

  Depth 7:
  
  Completed 109600 test(s) without failure.
  
  But 108576 did not meet \( \Rightarrow \) condition.

- Testing with Lazy SmallCheck

  Main> depthCheck 7 prop\_insertSet

  OK, required 1716 tests at depth 7
Laziness is Delicate

▶ A stronger invariant for ordered lists as sets:

\[ \text{isSet } s = \text{ordered } s \land \text{allDiff } s \]
Laziness is Delicate

- A stronger invariant for ordered lists as sets:

  isSet s = ordered s && allDiff s

- Redefining prop_insertSetSet accordingly, the number of tests almost halves:

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

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

  But if isSet conjuncts are switched, the number of tests increases 20-fold:

  isSet s = allDiff s && ordered s

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

- Standard && evaluates its left-hand argument first, and allDiff is less restrictive than ordered.
Laziness is Delicate

- A stronger invariant for ordered lists as sets:
  \[
  \text{isSet } s = \text{ordered } s \land \text{allDiff } s
  \]

- Redefining prop_insertSet accordingly, the number of tests *almost halves*:
  \[
  \text{prop_insertSet } c \ s = \text{isSet } s \implies \text{isSet } (\text{insert } c \ s)
  \]

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

- *But* if isSet conjuncts are switched, the number of tests *increases 20-fold*:
  \[
  \text{isSet } s = \text{allDiff } s \land \text{ordered } s
  \]

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

- A stronger invariant for ordered lists as sets:
  \[ \text{isSet } s = \text{ordered } s \land \text{allDiff } s \]

- Redefining \text{prop\_insertSet} accordingly, the number of tests \textit{almost halves}:
  \[
  \text{prop\_insertSet } c \ s = \text{isSet } s \Rightarrow \text{isSet } (\text{insert } c \ s)
  \]
  Main> depthCheck 7 prop_insertSet
  OK, required \textbf{964} tests at depth 7

- \textit{But} if \text{isSet} conjuncts are switched, the number of tests \textit{increases} 20-fold:
  \[
  \text{isSet } s = \text{allDiff } s \land \text{ordered } s
  \]
  Main> depthCheck 7 prop_insertSet
  OK, required \textbf{20408} tests at depth 7

- Standard \&\& evaluates its left-hand argument first, and \text{allDiff} is less restrictive than \text{ordered}. 
Parallel Conjunction

- The solution is *parallel refinement of conjuncts*.

\[
\text{isSet} :: \text{Ord } a \Rightarrow \text{Set } a \rightarrow \text{Property} \\
\text{isSet } s = \text{lift (ordered } s) \land \text{lift (allDiff } s) \\
\]

\[
\text{prop_insertSet} :: \text{Char} \rightarrow \text{Set Char} \rightarrow \text{Property} \\
\text{prop_insertSet } c \ s = \\
\text{isSet } s \Rightarrow \text{isSet (insert } c \ s) \\
\]

Testing this version of the property requires fewer tests than either of the sequential ones.

Main\textgreater depthCheck 7 prop_insertSet

OK, required 653 tests at depth 7.

Lists such as \(1:0: \bot\) falsify ordered but not allDiff; lists such as \(0:0: \bot\) falsify allDiff but not ordered.
Parallel Conjunction

- The solution is *parallel refinement of conjuncts.*

```haskell
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)
```

- Testing this version of the property requires *fewer tests than either* of the sequential ones.

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

- The solution is *parallel refinement of conjuncts*.
  
  \[
  \text{isSet} :: \text{Ord } a \Rightarrow \text{Set } a \rightarrow \text{Property} \\
  \text{isSet } s = \text{lift} \ (\text{ordered } s) \ \&\& \ \text{lift} \ (\text{allDiff } s) \\
  \]

  \[
  \text{prop_insertSet} :: \text{Char } \rightarrow \text{Set } \text{Char} \rightarrow \text{Property} \\
  \text{prop_insertSet } c \ s = \\
  \qquad \text{isSet } s \implies \text{isSet} \ (\text{insert } c \ s) \\
  \]

- Testing this version of the property requires *fewer tests than either* of the sequential ones

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

- Lists such as \(1:0:⊥\) falsify ordered but not allDiff; lists such as \(0:0:⊥\) falsify allDiff but not ordered.
Serial Types Redefined

- Standard instances of a Serial class can be written *just as in SmallCheck*, using \( \backslash / \) and the \texttt{cons<\textsc{N}>} family.
Serial Types Redefined

- Standard instances of a Serial class can be written *just as in SmallCheck*, using \(\backslash\) and the cons\(<\mathbb{N}>\) family.

- Underneath, the implementation is quite different.

  ```haskell
  type Series a = Int -> Cons a
  ```
Serial Types Redefined

- Standard instances of a Serial class can be written *just as in SmallCheck*, using `\` and the `cons<\>` family.
- Underneath, the implementation is quite different.

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

- Values of type `Cons a` describe how to construct and refine (partial) values of type `a`.

```haskell
data Cons a = Type :*: [[[Term] -> a]
data Type = SumOfProd [[[Type]]]
data Term = Ctr Int [Term] | Hole [Int] Type
```

- If a test evaluation reaches a `Hole`, a position-carrying exception is raised.

- By using a universal `Term` type, machinery such as refinement can be defined generically:

```haskell
refine :: Term -> Pos -> [[[Term]]]
```
Serial Types Redefined

- Standard instances of a Serial class can be written *just as in SmallCheck*, using \(\setminus\) and the \(\text{cons}^\mathbb{N}\) family.
- Underneath, the implementation is quite different.

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

- Values of type \(\text{Cons}~a\) describe how to construct and refine (partial) values of type \(a\).

```
data Cons a = Type :*: [[[Term] -> a] data Type  = SumOfProd [[[Type]]] data Term  = Ctr Int [Term] | Hole [Int] Type
```

- If a test evaluation reaches a Hole, a *position-carrying exception* is raised.
Serial Types Redefined

- Standard instances of a Serial class can be written \textit{just as in SmallCheck}, using \\ and the \texttt{cons<N>} family.

- Underneath, the implementation is quite different.

  \begin{verbatim}
  type Series a = Int -> Cons a
  \end{verbatim}

- Values of type \texttt{Cons a} describe how to construct and refine (partial) values of type \texttt{a}.

  \begin{verbatim}
  data Cons a = Type :*: [[Term] -> a]
  data Type   = SumOfProd [[Type]]
  data Term   = Ctr Int [Term] | Hole [Int] Type
  \end{verbatim}

- If a test evaluation reaches a Hole, a \textit{position-carrying exception} is raised.

- By using a \textit{universal} \texttt{Term} type, machinery such as refinement can be defined generically:

  \begin{verbatim}
  refine :: Term -> Pos -> [Term]
  \end{verbatim}
Comparative Evaluation
Red-black Trees (Okasaki)

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

redBlack :: Ord a => Tree a -> Bool
redBlack t = ord t && black t && red t

With a fault injected into rebalancing, we test whether insertion preserves the redBlack data invariant:

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

QC no counter-example after 100,000 batches of 1000 tests.
SC still testing at depth 4 after 20 minutes.
LSC level 4 counter-example after a fraction of a second.
Red-black Trees (Okasaki)

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

redBlack :: Ord a => Tree a -> Bool
redBlack t = ord t && black t && red t
```

With a fault injected into rebalancing, we test whether insertion preserves the redBlack data invariant:

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

QC no counter-example after 100,000 batches of 1000 tests.
SC still testing at depth 4 after 20 minutes.
LSC level 4 counter-example after a fraction of a second.
Huffman Compression (Bird)

\[
\text{prop\_decEnc } cs = \\
\quad \text{length } ft > 1 \implies \text{decode } t \ (\text{encode } t \ cs) = cs \\
\quad \text{where } ft = \text{collate } cs; \ t = \text{mkHuff } ft
\]
prop_decEnc cs =
    length ft > 1 ==> decode t (encode t cs) == cs
    where ft = collate cs; t = mkHuff ft

This property is hyperstrict.

SC Verifies to depth 10 in 1 min 30 sec.
LSC Verifies to depth 10 in 5 min 16 sec.
Huffman Compression (Bird)

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

This property is *hyperstrict*.

**SC** Verifies to depth 10 in 1 min 30 sec.

**LSC** Verifies to depth 10 in 5 min 16 sec.

prop_optimal cs t =
  isHuff t cs ==> cost ft t >= cost ft (mkHuff ft)
  where ft = collate cs
Huffman Compression (Bird)

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

This property is *hyperstrict*.

**SC** Verifies to depth 10 in 1 min 30 sec.

**LSC** Verifies to depth 10 in 5 min 16 sec.

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

Condition can be falsified for *partially-defined* arguments.

**SC** Verifies to depth 5 in 8 sec; still testing depth 6 after 20 min.

**LSC** Verifies to depth 6 in 23 sec.
Mate Chess Solver

Conjecture: king and pawn alone cannot give checkmate

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

Conjecture: king and pawn alone cannot give checkmate

module prop_checkmate =

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

QC finds no counter-example after 100,000 batches of 1000 random tests.

SC is still searching at depth 4 after 20 minutes.

LSC in under 30 seconds finds a counter-example at depth 5:
Mate Chess Solver

Conjecture: king and pawn alone cannot give checkmate

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

QC finds no counter-example after 100,000 batches of 1000 random tests.

SC is still searching at depth 4 after 20 minutes.

LSC in under 30 seconds finds a counter-example at depth 5:
Conclusions and Future Work

Overall Conclusions

- SmallCheck, Lazy SmallCheck and QuickCheck are *complementary* approaches to property-based testing in Haskell.
- Each tool has strengths and weaknesses making it effective for *some kinds of properties* but ineffective for others.

To-do List Top Three

- Refine SmallCheck’s treatment of functional values.
- Extend Lazy SmallCheck for higher-order and existential properties.
- Increase the *genericity of the property language* to enable free combinations of testing by different methods.
Availability

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

Support Acknowledged

- Galois
- EPSRC