Advances in Lazy SmallCheck

Jason S. Reich[1(✉)], Matthew Naylor[2], and Colin Runciman[1]

[1] Department of Computer Science, University of York, York, UK
{jason, colin}@cs.york.ac.uk
[2] Computer Laboratory, University of Cambridge, Cambridge, UK
matthew.naylor@cl.cam.ac.uk

Abstract. A property-based testing library enables users to perform lightweight verification of software. This paper presents improvements to the *Lazy SmallCheck* property-based testing library. Users can now test properties that quantify over *first-order functional values* and *nest universal and existential* quantifiers in properties. When a property fails, Lazy SmallCheck now accurately *expresses the partiality of the counter-example*. These improvements are demonstrated through several practical examples.

Keywords: Automated testing · Lazy SmallCheck · Functional values · Existential quantification · Search-based software engineering

1 Introduction

Property-based testing is a lightweight approach to verification where expected or conjectured program properties are defined in the source programming language. For example, consider the following conjectured property[1] that in Haskell every function with a list of Boolean values as an argument, and a single Boolean value as result, can be expressed as a *foldr* application.

```
prop_ ReduceFold :: ([Bool] → Bool) → Property
prop_ ReduceFold r = exists $ λf z → forAll $ λxs → r xs ≡ foldr f z xs
```

When this property is tested using our advanced version of *Lazy SmallCheck*, a small counterexample is found for *r*.

```
>>> test prop_ReduceFold
... Depth 2: Var 0: { [] -> False
      ; _:[] -> False
      ; _:_:_ -> True }
```

---

[1] Like all other properties used as examples in this paper, this property does not hold; our goal is to find a counterexample.

The counterexample is a function that tests for a multi-item list. It is expressed in the style of Haskell's case-expression syntax. Several new features of Lazy SmallCheck are demonstrated by this example. (1) Two of the quantified variables, *r* and *f*, are *functional values.* (2) An *existential quantifier* is used in the property definition. (3) The counterexample found for *r* is *concise* and understandable.

Previous property-based testing libraries struggle with such a property. The QuickCheck [2] library does not support existentials as random testing '*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* [14]'. QuickCheck also requires that functional values be wrapped in a *modifier* [1] for shrinking and showing purposes.

The original Lazy SmallCheck [14] supports neither existentials nor functional values. SmallCheck [14] supports all the necessary features of the property. However, it takes longer to produce a more complicated looking counterexample. This is because SmallCheck enumerates only fully defined test data and shows functions only in part, by systematically enumerating small arguments and corresponding results.

1.1   Contributions

This paper discusses the design, implementation[2] and use of new features in Lazy SmallCheck. We present several contributions:

- – An algorithm for checking properties that may contain universal and existential quantifiers in a Lazy SmallCheck-style testing library.
- – A method of lazily generating and displaying *functional values*, enabling the testing of higher-order properties.
- – An evaluation of these additions with respect to *functionality and run-time performance.*

1.2   Roadmap

Section 2 is a brief reminder of the Lazy SmallCheck approach to property-based testing. Section 3 demonstrates the new features of the Lazy SmallCheck through several examples. Section 4 describes architectural changes that enable these new features. Section 5 presents the formulation of functional values. Section 6 evaluates the new Lazy SmallCheck in comparison to other Haskell property-based testing libraries. Section 7 offers conclusions and suggestions for further work.

2   The Lazy SmallCheck Search Strategy

A property-based testing library uses a *strategy* to search the test data space for counterexamples to a given property. For example, QuickCheck [2] randomly

---

Table 1. Values of *xs* used by Lazy SmallCheck when testing *prop_ListSizes xs*.

	Test-data	Result		Test-data	Result
(1)	$\bot$	*Refine test-data*	(5)	$\bot : \bot : \bot$	*Refine test-data*
(2)	[]	*Property satisfied*	(6)	$\bot : \bot : []$	*Property satisfied*
(3)	$\bot : \bot$	*Refine test-data*	(7)	$\bot : \bot : \bot : \bot$	*Refine test-data*
(4)	$\bot : []$	*Property satisfied*	(8)	$\bot : \bot : \bot : []$	*Counterexample*

selects a fixed number of test-data values. SmallCheck [14], on the other hand, exhaustively constructs all possible values of a particular type, bounded by the depth of construction (or some appropriate metric for non-algebraic types).

Lazy SmallCheck instead begins by testing *undefined* — $\bot$ — as the value and *refines it by need*. The demands of the test property guide the exploration of the test-data space. When evaluation of a property depends on an *undefined* component of the test-data, *exactly* that component is refined. For algebraic datatypes, *undefined* is refined to all possible constructions, each with *undefined* arguments. To ensure termination, when Lazy SmallCheck is run, a bound is set on the depth of possible refinements.

Consider the illustrative property *prop_ ListSize*. It asserts that all lists with *Bool*-typed elements have lengths less than three.

```
prop_ ListSize :: [Bool] → Bool
prop_ ListSize xs = length xs < 3
```

Clearly this property is false. Lazy SmallCheck finds the following counterexample where each occurrence of _ means *any value*.

```
>>> test prop_ListSize
... Depth 3: Var 0: _:_:_:[]
```

As Lazy SmallCheck searches for this counterexample, it refines the test values bound to *xs* as shown in Table 1. Notice that the elements of the list *xs* are *never refined* as their values are *never needed* by the property. This pruning effect is the key benefit of Lazy SmallCheck over eager SmallCheck.

3   New Features in Action

The following examples further illustrate the new features in Lazy SmallCheck. The first generates *functional values* and displays *partial counterexamples*. The second shows the benefits of generating *small, partial functional values*. The final example demonstrates *existential quantification*.

3.1   Left and Right Folds

Let us look for a counterexample of another conjectured property. This property states that *foldl1 f* gives the same result as *foldr1 f* for non-empty list arguments with natural numbers as the element type.

$$prop\_ foldlr1 :: (Peano \rightarrow Peano \rightarrow Peano) \rightarrow [Peano] \rightarrow Property$$
$$prop\_ foldlr1\ f\ xs = (\neg \circ null)\ xs \implies foldl1\ f\ xs \equiv foldr1\ f\ xs$$

As in the original Lazy SmallCheck [14], testing this property requires a *Serial* instance for the *Peano* datatype. Additionally, an *Argument* instance must be defined so that Lazy SmallCheck can produce functional values with *Peano* arguments. We have defined a *Template Haskell* function [15] — *deriveArgument* — that automatically derives a suitable *Argument* instance. Section 5.2 discusses this in more detail.

data *Peano* = *Zero* | *Succ Peano* **deriving** (*Eq*, *Ord*, *Show*, *Data*, *Typeable*)

instance *Serial Peano* **where** *series* = *cons0 Zero* <|> *cons1 Succ*

deriveArgument "*Peano*

Lazy SmallCheck finds a counterexample at depth 3. The function *f* returns *Succ Zero* if its input is *Zero* and returns *Zero* in all other cases. The list *xs* is of length three where the last element is *Zero*.

```
 >>> test prop_foldlr1
Depth 3: ... Var 0: { _ -> { Zero -> Succ _
             ; Succ _ -> Zero } }
Var 1: _:_:Zero:[]
```

3.2   Generating Predicates

Our next example is based on *prop_ PredicateStrings* from Claessen [1].

$$prop\_ PredStrings :: (String \rightarrow Bool) \rightarrow Property$$
$$prop\_ PredStrings\ p = p\ \texttt{"Lazy SmallCheck"} \implies p\ \texttt{"SmallCheck"}$$

Lazy SmallCheck finds as a counterexample the function *p* that returns *True* when the second character in its argument is 'a' and *False* when any other character occurs in the second position. The function is *undefined* for strings of length less than two.

```
>>> test prop_PredStrings
...
Depth 4:
Var 0: { _:'a':_ -> True
       ; _:_:_ -> False }
```

Why is this the first counterexample found? We might expect a function that distinguishes an initial 'L' from an initial 'S'. As the depth-bound for testing increases, the extent to which the spines of list arguments can be refined increases. But also the range of character values used in refinements increases and the smallest non-empty range contains just 'a'.

QuickCheck also finds counterexamples for this property but the functions are stricter. They test equality with one of whole strings `"Lazy SmallCheck"` or `"SmallCheck"`.

3.3   Prefix of a List

This example is taken from Runciman et al. [14]. We assert that a (flawed) definition of *isPrefix* satisfies a soundness specification of the function.

$$isPrefix :: Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool$$
$$isPrefix\ [\,]\qquad \_\qquad\ = True$$
$$isPrefix\ (x : xs)\ (y : ys) = x \equiv y \vee isPrefix\ xs\ ys$$
$$isPrefix\ \_\qquad \_\qquad\ = False$$
$$prop\_isPrefixSound\ xs\ ys = isPrefix\ (xs :: [Peano])\ ys \implies$$
$$\quad (exists\ \$\ \lambda xs' \rightarrow xs + xs' \equiv ys)$$

In Runciman et al. [14], this property could only be checked by SmallCheck as Lazy SmallCheck did not support existential properties. Running it through the new Lazy SmallCheck gives another concise counterexample: if the first argument of *isPrefix* is a multi-item list with first element `Zero`, and the second argument is `[Zero]`; then *isPrefix* incorrectly returns `True`.

```
>>> test prop_isPrefixSound
... Depth 2: Var 0: Zero:_:_ Var 1: Zero:[]
```

A smallest counterexample with both *xs* and *ys* non-empty suggests an error in the second equation defining *isPrefix*. Indeed, a disjunction has been used in place of a conjunction.

4   Implementation of New Lazy SmallCheck

This section describes in detail how *new Lazy SmallCheck* achieves the process outlined in Sect. 2. We shall return to the *prop_ ListSize* example discussed in Sect. 2 to illustrate the data-types used in the implementation.

```
class Functor f where
    fmap :: (a → b) → f a → f b
infixl 3 <|>
infixl 4 <*>, <$>
(<$>) = fmap
class Functor f ⇒ Applicative f where
    pure  :: a → f a
    (<*>) :: f (a → b) → f a → f b
class Applicative f ⇒ Alternative f where
    empty :: f a
    (<|>) :: f a → f a → f a
```

Fig. 1. Definition of *Functor*, *Applicative* and *Alternative* type-classes.

In places, instead of the actual definitions used in the implementation, we give simpler versions that are less efficient but easier to read. These differences will be summarised in Sect. 4.5.

Abstractions We will make extensive use of the *Functor*, *Applicative* and *Alternative* type-classes. All are defined in Fig. 1. Functors are *containers* with an associated *fmap* operation that applies functions to each contained element. Lists, for example, are functors under the *map* function.

Applicative functors [12] extend this by viewing containers as *contexts* from which values may be obtained. Any ordinary value can be wrapped up in a context using *pure*. A function-in-context can be applied to a value-in-context using the (<*>) operator. Returning to the lists example, *pure* places the value into a singleton list and *fs* <*> *xs* applies every function in the collection *fs* to every argument in collection *xs* to obtain a collection of results.

Alternative functors are an extension of applicative functors by the addition of an *empty* container and an operation, (<|>), to merge containers. For lists, *empty* is the empty list and (<|>) is list concatenation.

4.1   Partial Values

Refinement exceptions As highlighted in Sect. 2, the test-data space includes partial values that are refined by need during the search for a counterexample. When the value of an *undefined* is needed, an exception tagged with the location of the *undefined* is raised and caught by the testing algorithm. The implementation uses GHC's *user-defined exceptions*. [11] The definition of Lazy SmallCheck's *refinement exceptions* can be found in Fig. 2.

The *Location* information uniquely identifies the component of a partial test-data value that is needed by a property under test. The *Path* in a *Location* gives directions from the root of a binary-tree representation to some specific

```
type Location = (Nesting, Path)
type Nesting = Int
type Path = [Bool]

data Refine = RefineAt Location deriving (Show, Typeable)
instance Exception Refine
```

Fig. 2. Definition of *Location* carrying exceptions.

subtree. The **Nesting** in a **Location** is akin to a *de Bruijn* [4] *level*: it identifies the quantifier for the test-data variable that needs refining.

Partial values functor A functor of **Partial** values is defined in Fig. 3. The only method of accessing the value inside the **Partial** functor is through *runPartial*. It forces the result of a computation using partial values and catches any refinement exception that may be raised.

A **Show** instance is defined so that **Partial** values can be printed. The definition is omitted here but it follows the *'Chasing Bottoms'* [3] technique. This is what allows the display of *wildcard patterns* in counterexamples.

Running example Consider the third value, $\perp : \perp$, tested in Table 1 from Sect. 2. Here is its simplified representation and the results of two small computations using it.

```
newtype Partial a = Partial {unsafePartial :: a}

instance Functor Partial where
    fmap f (Partial x) = Partial (f x)

instance Applicative Partial where
    pure = Partial
    Partial f <*> Partial x = Partial $ f x

runPartial :: (NFData a) ⇒ Partial a → Either Refine a
runPartial value = unsafePerformIO $
    (Right <$> evaluate (force (unsafePartial value)))
        `catch` (return ∘ Left)

refineAt :: Location → Partial a
refineAt = Partial ∘ throw ∘ RefineAt
```

Fig. 3. Definition of the **Partial** values functor.

```
  >>> let step3 = (:) <$> refineAt (0, [False, True])
                      <*> refineAt (0, [True])  :: Partial [a]

  >>> runPartial (prop_ListSize <$> step3)
 Left (RefineAt (0,[True]))

  >>> print (step3 :: Partial [Bool])
 _:_
```

The *undefined* arguments of the list-*cons* are uniquely tagged by locations. The result of applying *prop_ ListSize* shows that the second argument is needed. Pretty-printing this partial value hides the complexity underneath.
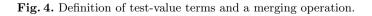
4.2   Test-Value Terms

The representation of a test-value term contains *tValue*, the information needed to obtain a partial test-data value, and *tRefine*, its possible refinements. The *Term* datatype is defined in Fig. 4.

The *Applicative* instance for terms shows how: (1) the *Path* component of a location is extended through the argument of *tValue* and (2) the *tRefine* uses this information to pass the rest of the path to the relevant subterm.
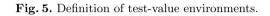
The *mergeTerms* function demonstrates how a collection of terms can be turned into a single *undefined* value paired with the ability to obtain the collection when required. This is key to the strategy illustrated in Sect. 2.

Test-value environments After test data is generated but before a property is applied to it, a pretty-printed representation of the partial value is recorded. The benefit of this technique is that we need not record a pretty-printing that

```
  data Term a = Term { tValue  :: (Location → TVE (Partial a))
                     , tRefine ::  (Path      → [ Term a]) }
  instance Functor  Term where
     fmap f ( Term v es) = Term ((fmap ∘ fmap ∘ fmap $ f ) v)
                                ((fmap ∘ fmap ∘ fmap $ f ) es)
  instance Applicative  Term where
     pure x = Term (pure ∘ pure ∘ pure $ x) (pure [])
     fs <⊛> xs = Term
       (λ(n, ps) → (<⊛>) <$> tValue fs (n, ps ++ [False])
                         <⊛> tValue xs (n, ps ++ [ True]))
       (λ(p : ps) → if p then fmap (fs <⊛>) (tRefine xs ps)
                    else  fmap (<⊛> xs) (tRefine fs ps))
  mergeTerms :: [ Term a] → Term a
  mergeTerms xs = Term ( TVE [string "_"] ∘ refineAt) (const xs)
```

Fig. 4. Definition of test-value terms and a merging operation.

```
data TVE a = TVE { tveEnv :: TVInfo, tveVal :: a }
type TVInfo = [AlignedString]
instance Functor TVE where
    fmap f (TVE ctx val) = TVE ctx (f val)
instance Applicative TVE where
    pure = TVE []
    TVE ctx0 f <*> TVE ctx1 x = TVE (ctx0 ++ ctx1) (f x)
```

Fig. 5. Definition of test-value environments.

could be obtained from the *final test-value* derived from the term. This will be especially useful for the display of functional values in Sect. 5.

The *test-value environments* type is shown in Fig. 5. We omit *AlignedString* in this paper but it follows established pretty-printing techniques, such as that used by Hughes [7].

4.3 Test-Value Series Generators

Series functor Properties are tested against a series of depth-bounded test-data terms. The Lazy SmallCheck library defines instances for the test-data *Series* functor that implicitly enforces depth-bounding and the introduction of partial test-data values. These definitions are in Fig. 6.

As with the original Lazy SmallCheck, a depth-cost is only introduced on the right-hand side of binary applications so that each child of a constructor is bounded by the same depth.

```
type Depth = Int
newtype Series a = Series { runSeries :: Depth → [Term a] }
instance Functor Series where
    fmap f xs = pure f <*> xs
instance Applicative Series where
    pure = Series ∘ pure ∘ pure ∘ pure
    Series fs <*> Series xs = Series $ λd →
        [ f <*> mergeTerms x | d > 0, f ← fs d
                             , let x = xs (d − 1), (¬ ∘ null) x ]
instance Alternative Series where
    empty = Series $ pure []
    Series xs <|> Series ys = Series $ (++) <$> xs <*> ys
```

Fig. 6. Definition of *Series* generators.

```
class (Data a, Typeable a) ⇒ Serial a where
    series :: Series a

    seriesWithEnv :: Series a
    seriesWithEnv = Series $ fmap storeShow <$> runSeries series
storeShow :: (Data a, Typeable a) ⇒ Term a → Term a
storeShow (Term v es) = Term
    ((fmap $ λ(TVE _ x) → TVE [string $ show x] x) v)
    (fmap storeShow <$> es)
```

Fig. 7. Definition of the *Serial* type-class.

Running example The following are definitions for depth-bounded values of Booleans, polymorphic lists and Boolean lists.

```
>>> let boolSeries = pure False <|> pure True
>>> let listSeries elem = pure []
                   <|> (:) <$> elem  <*> listSeries elem
>>> let listBoolSeries = listSeries boolSeries
```

Serial class A class of *Serial* types is defined in Fig. 7. Lazy SmallCheck uses *Serial* instances to automatically generate test values for argument variables in properties. Using the generic *Series* operators of Fig. 6, a family of $cons_n$ combinators can be defined exactly as described by Runciman et al. [14].

Running example again The library defines the series generators for many datatypes. The *Serial* instances for *Bool* and lists are as below. Notice that we no longer explicitly define how the arguments of list-*cons* are instantiated. It is automatically handled by the type system.

instance *Serial Bool* **where**
    series = cons0 False <|> cons0 True
instance *Serial a* ⇒ *Serial* [a] **where**
    series = cons0 [] <|> cons2 (:)

4.4   Properties and Their Refutation

Properties The *Property* data-type in Fig. 8 defines the abstract syntax of a domain-specific language. It includes standard Boolean operators. Crucially, it also provides a representation of universal and existential quantifiers that supports searches for counterexamples and witnesses.

Though not defined here, smart wrappers are provided for all six *Property* constructions. These automatically lift *Bool*-typed expressions to *Property* and

```
data Property = Lift Bool | Not Property
              | And Property Property  | Implies Property Property
              | ForAll (Series Property) | Exists (Series Property)
```

Fig. 8. The underlying representation of the *Property* DSL.

instantiate free variables in properties with appropriate series from *Serial* instances.

Refutation of properties The *depthCheck* function takes as arguments an integer depth-bound and a *Testable* property that may contain free variables of types of any *Serial* type. The *counterexample* and *refute* functions given in Fig. 9 search for a failing example.

A key point to observe is that *refute* recurses when it encounters a nested quantification. All refinement requests must therefore be tagged with the *Nesting level* for the associated quantifier. The *RefineAt* information can then be passed onto the relevant *tRefine* function. Those refined terms are then prepended onto the list of terms left to test.

```
counterexample :: Depth → Series Property → Maybe TVInfo
counterexample d xs = either ⊥ id $ refute 0 d xs

refute :: Nesting → Depth → Series Property → Either Refine (Maybe TVInfo)
refute n d xs = terms (runSeries xs d)
  where
    terms :: [Term Property] → Either Refine (Maybe TVInfo)
    terms []               = Right Nothing
    terms (Term v es : ts) = case (join ∘ runPartial ∘ fmap prop) <$> v (n, []) of
        TVE _    (Left (RefineAt (m, ps))) | m ≡ n     → terms $ es ps ++ ts
                                           | otherwise → Left   $ RefineAt (m, ps)
        TVE info (Right False)                         → Right $ Just info
        TVE _    (Right True)                          → terms $ ts
    prop :: Property → Either Refine Bool
    prop (Lift    v)  = pure v
    prop (Not     p)  = ¬          <$> prop p
    prop (And     p q) = (∧)        <$> prop p <*> prop q
    prop (Implies p q) = ( ⟹ )      <$> prop p <*> prop q
    prop (ForAll  xs)  = isNothing <$> refute (succ n) d xs
    prop (Exists  xs)  = isJust     <$> refute (succ n) (succ d) (fmap Not xs)
```

Fig. 9. Definition of the refutation algorithm.

4.5   Differences Between Versions of Lazy SmallCheck

The main differences between the new Lazy SmallCheck and the original Lazy SmallCheck described in [14] are as follows. In the new implementation:

– Terms are always represented in a type-specific way. Previously they were generated from a generic description.
– Terms can carry a *test-value environment* enabling the display of test-data types (such as functions) that cannot be directly pretty-printed.
– The testing algorithm calls itself recursively, refining information about enclosing quantifiers.

The main differences between real implementation of the new Lazy SmallCheck and the slightly simplified variant described in this paper are as follows. In the real implementation:

– The *Path* datatype is a *difference list* to optimise the list-*snoc* operation.
– Terms representing total and partial values are distinguished to optimise performance and to allow the use of existing *Show* instances for total terms.
– Terms representing partial values record the total number of potential refined values they represent up to the depth bound. The refutation algorithm counts the actual number of refinements performed. *(This is useful for performance measurements and comparison with other approaches.)*

5   Implementing Functional Values

The key to generating functional values is the ability to represent them as tries, also known as prefix trees. New Lazy SmallCheck supports the derivation of appropriate tries for given argument types, and the conversion of tries into functions to be used as test values.

The use of test-value environments allows a trie to be pretty-printed *before* it is converted into a Haskell function. This removes the need for the kind of modifier used by Claessen [1].

5.1   Trie Representations of Functions

We define a generic trie datatype in Fig. 10. It is expressed as a two-level, mutually recursive GADT. Level one describes functions that either ignore their argument — *Wild*, or perform a case inspection of it — *Case*.

Level two represents details of a case inspection. The *Valu* construction occurs when the argument is of unit type and therefore returns the single result. The *Sum* construction represents functions with a tagged union as argument type, performing further inspection on their constituent types. The *Prod* construction represents functions with arguments of a product type, producing a trie that first inspects the left component of the product, then the right to return a value.

A construction *Natu vs v* represents a function with a natural number argument. If an argument *n* is less than the length of *vs*, the value of *vs* !! *n* is

```
type (:→:) = Level1
data Level1 k v where
   Wild :: v → Level1 k v
   Case :: Level2 k v → Level1 k v
data Level2 k v where
   Valu :: v                              → Level2 () v
   Sum :: Level2 j v → Level2 k v         → Level2 (Either j k) v
   Prod :: Level2 j (Level2 k v)          → Level2 (j, k) v
   Natu :: [v] → v                        → Level2 Nat v
   Cast :: Argument k ⇒ Level1 (Base k) v → Level2 (BaseCast k) v
applyT :: (k :→: v) → k → v
applyT (Wild v) = const v
applyT (Case t) = applyL2 t
applyL2 :: Level2 k v → k → v
applyL2 (Valu v)    _            = v
applyL2 (Sum t _)   (Left k)     = t `applyL2` k
applyL2 (Sum _ t)   (Right k)    = t `applyL2` k
applyL2 (Prod t)    (j, k)       = t `applyL2` j `applyL2` k
applyL2 (Natu m d)  (Nat k)      = foldr const d $ drop k m
applyL2 (Cast t)    (BaseCast k) = t `applyT` k
```

Fig. 10. Definition of the two-level trie data structure.

returned. Otherwise *v* is returned as default. The *Cast* construction is used in all other cases. We shall say more about it in Sect. 5.2. The function *applyT* converts a trie into a Haskell function.

5.2   Custom Data-Types for Functional Value Arguments

The *Argument* class is defined in Fig. 11. Users supply an instance *Argument t* to enable generated functional test values with an argument of type *t*. Each instance defines a *base type representation* and an *isomorphism* between the argument type and the base type. This is a variation of the generic trie technique used by Hinze [6]. The *Cast* construction of the trie datatype performs the necessary type conversions using the *Argument* instances.

The *BaseCast* functor is used at recursive points to prevent infinite representations of recursive datatypes. It is a type-level thunk indicating that an arbitrary type can be translated into a *Base* type. For example, Fig. 12 shows the *Argument Peano* instance. The Template Haskell function *deriveArgument* automatically produces *Argument* instances for any Haskell 98 type.

5.3   Serial Instances of Functional Values

Functional values have been reified through the trie datatype, so we first need to define series of types. The *Serial* instances are defined in Fig. 13. A special

```
class (SerialL2 (Base k), Typeable k, Data k) ⇒ Argument k where
   type Base k
   toBase    :: k → Base k
   fromBase :: Base k → k
data BaseCast a = BaseCast {forceBase :: Base a}
toBaseCast :: Argument k ⇒ k → BaseCast k
toBaseCast = BaseCast ∘ toBase
fromBaseCast :: Argument k ⇒ BaseCast k → k
fromBaseCast = fromBase ∘ forceBase
```

Fig. 11. Definition of the *Argument* type-class.

```
instance Argument Peano where
   type Base Peano = Either () (BaseCast Peano)
   toBase Zero       = Left ()
   toBase (Succ n)   = Right $ toBaseCast n
   fromBase (Left _)  = Zero
   fromBase (Right n) = Succ $ fromBaseCast n
```

Fig. 12. The *Argument* instance for *Peano*.

```
seriesT :: (SerialL2 k) ⇒ Series v → Series (k :→: v)
seriesT srs = (Wild <$>ˆ srs) <|> (Case <$> seriesL2 srs)
class SerialL2 k where
   seriesL2 :: Series v → Series (Level2 k v)
instance SerialL2 () where
   seriesL2 srs = Valu <$>ˆ srs
instance (SerialL2 j, SerialL2 k) ⇒ SerialL2 (Either j k) where
   seriesL2 srs = Sum <$>ˆ seriesL2 srs <∗>ˆ seriesL2 srs
instance (SerialL2 j, SerialL2 k) ⇒ SerialL2 (j, k) where
   seriesL2 srs = Prod <$>ˆ seriesL2 (seriesL2 srs)
instance SerialL2 Nat where
   seriesL2 srs = Natu <$>ˆ fullSizeList srs <∗>ˆ srs
instance Argument k ⇒ SerialL2 (BaseCast k) where
   seriesL2 srs = Cast <$>ˆ seriesT srs
```

Fig. 13. Definition of *Series* generators for tries and functions.

Table 2. Comparision of property-based testing library features.

Feature	QuickCheck	SmallCheck	Original LSC	New LSC
Test strategy	Random	Bounded exhaustive	Bounded exhaustive	Bounded exhaustive
Test-space pruning	N/A	N/A	Lazy generation	Lazy generation
Minimal result	Shrinking	Natural	Natural	Natural
Functional values	Yes[a]	Yes	No	Yes
Existentials	No	Yes	No	Yes
Nested quantification	Yes	Yes	No	Yes
Displays partial counterexamples	N/A	N/A	No	Yes
Haskell 98/2010	Partial[b]	Compatible	Compatible	No[c]

[a] Functional value is wrapped in a modifier at its quantification binding if showing or shrinking is required.

[b] Originally Haskell 98 compatible but functional values modifier requires GADTs.

[c] Requires Haskell extensions: GADTs, type families and flexible contexts.

type-class *SerialL2* is defined. It represents types that can be represented as trie constructions. The applicative operators with a *carret suffix* introduce *no depth cost*, as opposed to those defined in Sect. 4.3. These specialist operators have been carefully placed to give a natural depth metric for functions while keeping the series finite.

Using these definitions, a *Serial* instance for functional values is defined. The default definition of *seriesWithEnv* is overridden to store the pretty-printed form of the trie before it is converted into a Haskell function. This instance definition is omitted here due to lack of space.

6   Discussion and Related Work

A feature comparison of several Haskell property-based testing libraries can be found in Table 2. The test-space exploration strategy is the main distinction between the QuickCheck library and SmallCheck family of libraries. QuickCheck assumes that test data detecting a failure is likely within some probability distribution. SmallCheck, on the other hand, appeals to the *Small Scope hypothesis* [8] — programming errors are likely to appear for small test data.

6.1   Runtime Performance

The repository includes performance benchmarks to compare this implementation with the previously published Lazy SmallCheck. Experiments performed using GHC 7.6.1 with `-O2` optimisation on a 2GHz quad-core PC with 16GB of RAM show very little difference in execution times between the two encodings.

6.2   Functional Values

The original QuickCheck paper [2] explains how functional test values can be generated through the *Arbitrary* instance of functions with a *Coarbitrary* instance of argument types. At this stage, QuickCheck could not display the failing example without bespoke use of the *whenFail* property combinator.

QuickCheck has since gained the ability not only to display functional counterexamples but also to reduce their complexity through *shrinking*. Claessen [1] achieves this by transforming functions generated using the existing *Coarbitrary* technique into tries.

Claessen's formulation of tries slightly differs from ours. Existential types are used in place of type families and there is no provision for non-strict functions. Partiality of functions is explicitly expressed instead of being a result of partially defined tries. Claessen also requires that functions are wrapped in a *'modifier'* at quantification binding. This *Fun* modifier retains information for showing and shrinking at the expense of a slightly more complex interface presented to users.

In Lazy SmallCheck, on the other hand, we directly generate a trie and then convert it into a Haskell function. A pretty-printed representation of the trie is stored at the time of generation and retrieved for counterexample output.

The SmallCheck representation of functional values uses a *coseries* approach, analogous to QuickCheck's *Coarbitrary*. However, functional values are displayed by systematically enumerating arguments.

6.3   Existential and Nested Quantification

As previously discussed in Sect. 1, it does not make sense to use QuickCheck for existential quantification. The previous design of Lazy SmallCheck made it difficult to conceive of a refutation algorithm that could handle the nested quantification required to make existential properties useful.

The use of the *Partial* values functor in this implementation gives statically typed guarantees that term refinements are performed at the correct quantifier nesting.

6.4   Benefits of Laziness

Runciman et al. [14] discussed the benefits and fragility of exploiting the laziness of the host language to prune the test-data search space. When applied to functional values, we see further benefits. The partiality of a trie representation corresponds directly with the partiality of the function it represents. Whereas Claessen [1] needs to shrink total function to partial functions, the latest Lazy SmallCheck has partial functions as a natural result of its construction.

7   Conclusions and Further Work

This paper has described the extension of Lazy SmallCheck with several new features; (1) quantification over functional values, (2) existential and nested quantification in properties and (3) the display of partial counterexamples.

Properties that quantify over functional values occur often in higher-order functional programming. Similarly, many properties may involve existential quantification and even nesting of quantification within property definitions. The examples in this paper have demonstrated the power of a tool that can find counterexamples for such properties.

This paper takes an *extensional* view of functional values, characterising them as mappings from input to output. An alternative would be to characterise functions *intensionally* as lambda abstractions or other defining expressions, perhaps allowing recursion [9,10]. We would expect the generic machinery for typed functional series to be more complex. Also, when functions are needed as test values, alternative definitions of the same extensional function are not interesting [13].

Parallelisation of the refutation algorithm is a current area of investigation. A prototype implementation shows near-linear speedups, in multicore shared-memory environments, for benchmarks in which no counterexample is found. This benefit is derived from the tree structure of the Lazy SmallCheck test-value search space. However, in some benchmarks where a counterexample is found the overheads of continued searches in other threads can cause slowdowns rather than speedups.

References

1. Claessen, K.: Shrinking and showing functions: (functional pearl). In: Proceedings of the 2012 Symposium on Haskell, pp. 73–80. Haskell '12, ACM (2012)
2. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, pp. 268–279. ICFP '00. ACM (2000)
3. Danielsson, N.A., Jansson, P.: Chasing bottoms. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 85–109. Springer, Heidelberg (2004)
4. de Bruijn, N.G.: Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Math. **34**, 381–392 (1972)
5. Elliott, C.: Elegant memoization with functional memo tries. http://conal.net/blog/posts/elegant-memoization-with-functional-memo-tries (October 2008). Accessed 26 July 2012
6. Hinze, R.: Generalizing generalized tries. J. Funct. Program. **10**(04), 327–351 (2000)
7. Hughes, J.: The design of a pretty-printing library. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 53–96. Springer, Heidelberg (1995)

8. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2012). Revised edn

9. Katayama, S.: Systematic search for lambda expressions. In: Trends in Functional Programming, TFP2005, vol. 6, pp. 111–126. Intellect Books (2007)

10. Koopman, P., Plasmeijer, R.: Synthesis of functions using generic programming. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) AAIP 2009. LNCS, vol. 5812, pp. 25–49. Springer, Heidelberg (2010)

11. Marlow, S.: An extensible dynamically-typed hierarchy of exceptions. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell, pp. 96–106. Haskell '06. ACM (2006)

12. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (2008)

13. Reich, J.S., Naylor, M., Runciman, C.: Lazy generation of canonical test programs. In: Gill, A., Hage, J. (eds.) IFL 2011. LNCS, vol. 7257, pp. 69–84. Springer, Heidelberg (2012)

14. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell, pp. 37–48. Haskell '08, ACM (2008)

15. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, pp. 1–16. Haskell '02. ACM (2002)