QuickSpec—
Formal Specifications for Free!

Beijing 2023 Autumn School

John Hughes

CHALMERS QuviQ

A simple example

```
import Test.QuickCheck
import QuickSpec

intSig = [
  con "+"      ((+) :: Int -> Int -> Int),
  con "negate" (negate :: Int -> Int)
  ]
```

Define a signature containing named operations and their types

```
*Int> quickSpec intSig
== Functions ==
   (+) :: Int -> Int -> Int
negate :: Int -> Int

== Laws ==
  1. x + y = y + x                                Commutativity
  2. negate (negate x) = x                        Double negation
  3. x + negate x = y + negate y                  ???
  4. (x + y) + z = x + (y + z)                    Associativity
  5. negate x + negate y = negate (x + y)         Distribution
  6. negate x + (x + y) = y                       ???
```

- All the laws are *true*, but some are unexpected
- No equation is printed *if it follows from the previous ones*

Adding 0 to the vocabulary

```
intSig0 = intSig ++ [
  con "0" (0 :: Int)
  ]
```

```
*Int> quickSpec intSig0
== Functions ==
   (+) :: Int -> Int -> Int
negate :: Int -> Int
     0 :: Int

== Laws ==
  1. negate 0 = 0
  2. x + y = y + x
  3. x + 0 = x
  4. negate (negate x) = x
  5. x + negate x = 0
  6. (x + y) + z = x + (y + z)
  7. negate x + negate y = negate (x + y)

  6. negate x + (x + y) = y ???
```

```
negate x + (x + y) = y
                    by 6. (x + y) + z = x + (y + z)
(negate x + x) + y = y
                    by 2. x + y = y + x
(x + negate x) + y = y
                    by 5. x + negate x = 0
0 + y = y
                    by 2. x + y = y + x
y + 0 = y
                    by 3. x + 0 = x
y = y
```

Let’s try it for Float

```
floatSig = [
  con "+"       ((+) :: Float -> Float -> Float),
  con "negate" (negate :: Float -> Float),
  con "0.0" (0 :: Float)
  ]
```

```
*Float> quickSpec floatSig
== Functions ==
   (+) :: Float -> Float -> Float
negate :: Float -> Float
   0.0 :: Float

WARNING: The following types have no 'Arbitrary'
instance declared.
You will not get any variables of the following types:
  Float

WARNING: The following types have no 'Ord' or
'Observe' instance declared.
You will not get any equations about the following
types:
  Float

== Laws ==
```

No laws!

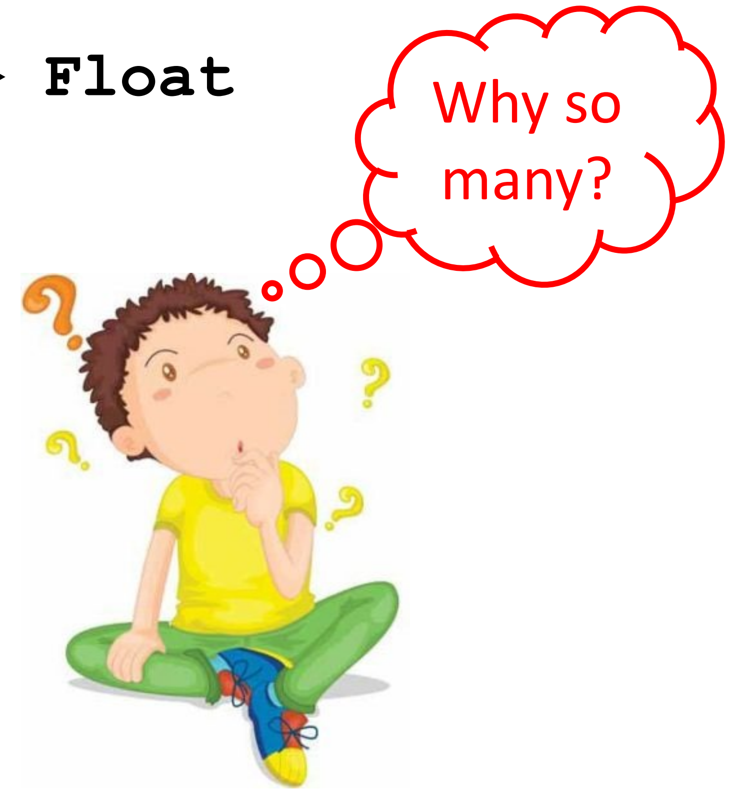
```
*Float> quickSpec floatSig
== Functions ==
   (+) :: Float -> Float -> Float
negate :: Float -> Float
   0.0 :: Float

WARNING: The following types have no 'Arbitrary'
instance declared.
You will not get any variables of the following types:
  Float

WARNING: The following types have no 'Ord' or
'Observe' instance declared.
You will not get any equations about the following
types:
  Float

== Laws ==
```

No laws!

Let’s try it for Float, the right way

```
floatSig = [
  monoType (Proxy :: Proxy Float),
  con "+"      ((+) :: Float -> Float -> Float),
  con "negate" (negate :: Float -> Float),
  con "0.0" (0 :: Float)
  ]
```

```
*Float> quickSpec floatSig
== Functions ==
   (+) :: Float -> Float -> Float
negate :: Float -> Float
   0.0 :: Float

== Laws ==
  1. negate 0.0 = 0.0
  2. x + y = y + x
  3. x + 0.0 = x
  4. negate (negate x) = x
  5. x + negate x = 0.0
  6. negate x + negate y = negate (x + y)
  7. negate x + (x + x) = x
  8. (x + y) + (x + y) = (x + x) + (y + y)
  9. x + (x + (x + x)) = (x + x) + (x + x)
```



Comparing with Int

```
1. negate 0 = 0
2. x + y = y + x
3. x + 0 = x
4. negate (negate x) = x
5. x + negate x = 0
6. (x + y) + z = x + (y + z)
7. negate x + negate y =
     negate (x + y)
```

```
1. negate 0.0 = 0.0
2. x + y = y + x
3. x + 0.0 = x
4. negate (negate x) = x
5. x + negate x = 0.0
6. negate x + negate y =
     negate (x + y)
7. negate x + (x + x) = x
8. (x + y) + (x + y) =
     (x + x) + (y + y)
9. x + (x + (x + x)) =
     (x + x) + (x + x)
```

Is Associativity true?

```
prop_Assoc :: Float -> Float -> Float -> Property
prop_Assoc x y z =
   (x + y) + z === x + (y + z)
```

```
*Float> quickCheck prop_Assoc
*** Failed! Falsified (after 3 tests and 6 shrinks):
1.0
0.7
0.1
1.8000001 /= 1.8
```

QuickSpec is complete!

Further investigations…

- Haskell has *exact rationals*

```
rationalSig = [
  monoType (Proxy :: Proxy Rational),
  con "+"      ((+) :: Rational -> Rational -> Rational),
  con "negate" (negate :: Rational -> Rational),
  con "0" (0 :: Rational)
  ]
```

```
== Laws ==
  1. negate 0 = 0
  2. x + y = y + x
  3. x + 0 = x
  4. negate (negate x) = x
  5. x + negate x = 0
  6. (x + y) + z = x + (y + z)
  7. negate x + negate y = negate (x + y)
```

Generate terms and laws,
but ***don't print them***

Exact

```
jointSig = [
  background floatSig,
  background rationalSig,
  con "toRational"   (toRational   :: Float -> Rational),
  con "fromRational" (fromRational :: Rational -> Float)
  ]
```

Rounding

```
== Laws ==
  1. toRational 0.0 = 0
  2. fromRational 0 = 0.0
  3. fromRational (toRational x) = x
  4. fromRational (negate x) = negate (fromRational x)
  5. negate (toRational x) = toRational (negate x)
  6. fromRational x + fromRational x = fromRational (x + x)
  7. toRational x + toRational x = toRational (x + x)
  8. fromRational (toRational x + toRational y) = x + y
  9. fromRational (x + toRational (fromRational x)) =
       fromRational (x + x)
```

```
prop_Inverse r =
  toRational (fromRational r) === r
```

```
*Float> quickCheck prop_Inverse
*** Failed! Falsified (after 2 tests and 38 shrinks):
1 % 653954092621
757203022813914 9 % 4951760157141521099596496896 /= 1 % 653954092621
```

Binary Search Trees again

```
type Tree = BST Int Integer

treeSig = [
  monoType (Proxy :: Proxy Tree),
  con "nil"    (nil    :: Tree),
  con "find"   (find   :: Int -> Tree -> Maybe Integer),
  con "insert" (insert :: Int -> Integer -> Tree -> Tree)
  ]
```

```
== Functions ==
   nil :: BST Int Integer
  find :: Int -> BST Int Integer -> Maybe Integer
insert :: Int -> Integer -> BST Int Integer -> BST Int Integer

== Laws ==
  1. find x nil = find y nil
  2. find x (insert x y z) = find x (insert x y w)
  3. find x (insert x y z) = find w (insert w y z)
  4. find x (insert y z nil) = find y (insert x z nil)
  5. insert x y (insert x z w) = insert x y w
```

Doesn't matter—we always get **`Nothing`**

`Just y`

Binary Search Trees again

```
treeSig = [
  monoType       (Proxy :: Proxy Tree),
  con "nil"      (nil    :: Tree),
  con "find"     (find   :: Int -> Tree -> Maybe Integer),
  con "insert"   (insert :: Int -> Integer -> Tree -> Tree),
  con "Nothing" (Nothing :: Maybe Integer),
  con "Just"     (Just    :: Integer -> Maybe Integer)
  ]
```

```
*BSTSpec> quickSpec treeSig
== Functions ==
    nil :: BST Int Integer
   find :: Int -> BST Int Integer -> Maybe Integer
 insert :: Int -> Integer -> BST Int Integer -> BST Int Integer
Nothing :: Maybe Integer
   Just :: Integer -> Maybe Integer

== Laws ==
  1. find x nil = Nothing
  2. find x (insert x y z) = Just y
  3. find x (insert y z nil) = find y (insert x z nil)
  4. insert x y (insert x z w) = insert x y w
```

- x == y && (2) ➔ (3)
- x /= y ➔ both sides are `Nothing`

Adding preconditions

```
neTreeSig = [
  background treeSig,
  predicate "/=" ((/=) :: Int -> Int -> Bool)
  ]
```

```
== Laws ==
  1. x /= y = y /= x
  2. x /= x = y /= y
  3. x /= y => find x (insert y z w) = find x w
```

Equivalence

Why don't we get

```
4. x /= z => insert x y (insert z w t) =
                 insert z w (insert x y t)
```

—one of the properties we wrote on Tuesday?

The trees are *equivalent*, but not *equal*

Recall `t1 =~= t2 = toList t1 === toList t2`

QuickSpec uses == to compare, but we can change that!

Observing a type

- We need to tell QuickSpec *how* to observe trees

```
instance (Ord k, Ord v) =>
            Observe () [(k,v)] (BST k v) where
  observe () = toList
```

- We need to turn on language extensions…

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
```

- We need to tell QuickSpec to *use* the **`observe`** function for this type

```
treeEquivSig = treeSig ++ [
  monoTypeObserve (Proxy :: Proxy Tree)
  ]
```

```
== Laws ==
  1. find x nil = Nothing
  2. find x (insert x y z) = Just y
  3. find x (insert y z nil) = find y (insert x z nil)
  4. insert x y (insert x z w) = insert x y w
  5. insert x y (insert z y w) = insert z y (insert x y w)
```

*Swap insertions with different **keys**, but the same **value***

```
neTreeEquivSig = [
  background treeEquivSig,
  predicate "/=" ((/=) :: Int -> Int -> Bool)
  ]
```

```
== Laws ==
  1. x /= y = y /= x
  2. x /= x = y /= y
  3. x /= y => find x (insert y z w) = find x w
```

WHAT??? No law about swapping insert?

Sometimes equations are *too big* for QuickSpec to find

```
neTreeEquivSig = [
  background treeEquivSig,
  withMaxTermSize 8,
  predicate "/=" ((/=) :: Int -> Int -> Bool)
  ]
```

```
== Laws ==
  1. x /= y = y /= x
  2. x /= x = y /= y
  3. x /= y => find x (insert y z w) = find x w
  4. z /= x => insert x y (insert z w x2) =
                 insert z w (insert x y x2)
```

What if the code is buggy?

BST1

```
*BSTSpec> quickSpec $ treeEquivSig ++
  [background $ predicate "/=" ((/=) :: Int -> Int -> Bool)]
…

== Laws ==
  1. find x nil = Nothing
  2. insert x y z = insert x y w
  3. find x (insert y z w) = find y (insert x z w)
  4. find x (insert x y z) = Just y
  5. x /= y => find x (insert y z w) = Nothing
```

We get the wrong laws… which make the buggy behaviour pretty clear!

BST2

```
*BSTSpec> quickSpec $ treeEquivSig ++
  [background $ predicate "/=" ((/=) :: Int -> Int -> Bool)]
…

== Laws ==
  1. find x nil = Nothing
  2. find x (insert y z nil) = find y (insert x z nil)
  3. find x (insert x y nil) = Just y
  4. x /= y => find x (insert y z w) = find x w
```

All the laws are true, but…

- A law to swap insertions is missing

BST2

```
*BSTSpec> quickSpec $ treeEquivSig ++
  [background $ predicate "/=" ((/=) :: Int -> Int -> Bool)]
…

== Laws ==
  1. find x nil = Nothing
  2. find x (insert y z nil) = find y (insert x z nil)
  3. find x (insert x y nil) = Just y
  4. x /= y => find x (insert y z w) = find x w
```

All the laws are true, but…

- A law to swap insertions is missing
- We expect (3) to hold *in general*, not just for nil

```
*BSTSpec> quickCheck $ \x y t ->
  find (x :: Int) (insert x (y :: Integer) t) === Just y
*** Failed! Falsified (after 15 tests and 9 shrinks):
12
1
Branch Leaf 12 0 Leaf
Just 0 /= Just 1

*BSTSpec> insert 12 1 (Branch Leaf 12 0 Leaf)
Branch Leaf 12 0 (Branch Leaf 12 1 Leaf)
```

BST3

```
*BSTSpec> quickSpec $ treeEquivSig ++
  [background $ predicate "/=" ((/=) :: Int -> Int -> Bool)]
…

== Laws ==
  1. find x nil = Nothing
  2. find x (insert y z nil) = find y (insert x z nil)
  3. find x (insert x y nil) = Just y
  4. x /= y => find x (insert y z w) = find x w
  5. insert x y (insert x z w) = insert x z w
  6. insert x y (insert z y w) = insert z y (insert x y w)
```

What use is QuickSpec?

- The laws give insight—a good design should satisfy a nice algebra
- *Missing* laws reveal bugs or infelicities
- Laws make great *regression tests*
- A tool for *design*?

```
listsSig = [
  con "[]" ([] :: [Int]),
  con "++" ((++) :: [Int] -> [Int] -> [Int])
  ]
```

```
*Lists> quickSpec listsSig
== Functions ==
  [] :: [Int]
(++) :: [Int] -> [Int] -> [Int]

== Laws ==
  1. xs ++ [] = xs
  2. [] ++ xs = xs
  3. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

```haskell
consSig = [
  background listsSig,
  con ":" ((:) :: Int -> [Int] -> [Int])
  ]
```

```
*Lists> quickSpec consSig
== Functions ==
  [] :: [Int]
(++) :: [Int] -> [Int] -> [Int]

== Functions ==
(:) :: Int -> [Int] -> [Int]

== Laws ==
  1. (x : xs) ++ ys = x : (xs ++ ys)
  2. [] ++ xs = xs
```

Part of the ***definition***

```
revSig = [
  background consSig,
  con "reverse" (reverse :: [Int] -> [Int])
  ]
```

```
*Lists> quickSpec revSig
…

== Laws ==
  1. reverse [] = []
  2. reverse (reverse xs) = xs
  3. reverse (x : []) = x : []
  4. reverse xs ++ reverse ys = reverse (ys ++ xs)
  5. xs ++ (x : (y : [])) = reverse (y : (x : reverse
xs))
```

```
revSig = [
  background consSig,
  con "reverse" (reverse :: [Int] -> [Int]),
  maxPruningDepth 0
  ]
```

```
*Lists> quickSpec revSig
…

== Laws ==
  1. reverse [] = []
  2. reverse (reverse xs) = xs
  3. reverse (x : []) = x : []
  4. reverse xs ++ reverse ys = reverse (ys ++ xs)
  5. reverse (x : reverse xs) = xs ++ (x : [])
  6. reverse (xs                    ys ++ reverse xs
                xs = reverse ys
  7. reverse (                      ys ++ xs
  8. reverse (x : xs) ++ ys = reverse xs ++ (x : ys)
     reverse (x : ys) = reverse ys ++ (x : [])
  9. re
 10. reverse (xs ++ (x : ys)) = reverse ys ++ (x : reverse xs)
 11. reverse xs ++ (reverse ys ++ zs) = reverse (ys ++ xs) ++ zs
 12. reverse (x : reverse (xs ++ ys)) = (xs ++ ys) ++ (x : [])
 13. xs ++ (x : (y : [])) = reverse (y : (x : reverse xs))
 14. xs ++ (x : reverse ys) = reverse (ys ++ (x : reverse xs))
 15. reverse xs ++ (ys ++ reverse zs) = reverse (zs ++ (reverse ys ++ xs))
```

Eureka! `rev' xs ys = reverse xs ++ ys`

```
rev'Sig = [
  background revSig,
  con "rev'" (rev' :: [Int] -> [Int] -> [Int])
  ]
```

```
*Lists> quickSpec rev'Sig
…
== Laws ==
  1. rev' xs [] = reverse xs
  2. rev' [] xs = xs
  3. reverse (rev' xs ys) = rev' ys xs
  4. rev' xs ys = reverse xs ++ ys
  5. rev' (reverse xs) ys = xs ++ ys
  6.
  7. rev' (rev' xs ys) zs =                xs ++ zs
  8. rev' xs (x : []) =                    xs)
  9. reverse (rev' xs y                ' zs (rev' ys xs)
 10. rev' xs (x : ys) ++ zs = rev' (x : xs) (ys ++ zs)
 11. rev' (x : (y : xs)) [] = reverse (x : (y : xs))
```

`rev' xs (x:ys) = rev' (x:xs) ys`

Take `zs = []`
and simplify

Changing representations

```
data List a = Nil | Snoc (List a) a
```

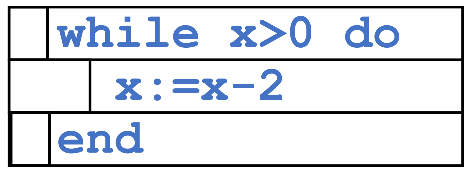
```
snoc xs x = xs ++ [x]

snocSig = [
  con "Nil"  ([]   :: [Int]),
  con "Snoc" (snoc :: [Int] -> Int -> [Int]),
  background [
    con "++" ((++) :: [Int] -> [Int] -> [Int]),
    con "reverse" (reverse :: [Int] -> [Int])
    ],
  withPruningDepth 0
  ]
```

```
== Laws ==
  1. reverse Nil = Nil
  2. xs ++ Nil = xs
  3. Nil ++ xs = xs
  4. reverse (Snoc Nil x) = Snoc Nil x
  5. xs ++ Snoc ys x = Snoc (xs ++ ys) x
  6. reverse (Snoc (reverse xs) x) = Snoc Nil x ++ xs
  7. xs ++ reve ... c xs x ++ reverse ys
  8. reverse (S ... Snoc (Snoc Nil y) x
  9. Snoc (xs + ... (Snoc ys x ++ zs)
 10. xs ++ (Sno ... xs x ++ ys
 11. S ... c (Snoc (reverse xs)
y) x)
 12. Snoc (reverse ... everse (Snoc (reverse ys) x ++
xs)
```

Let `xs = reverse ys`

```
reverse (Snoc ys x) = Snoc Nil x ++ reverse ys
```

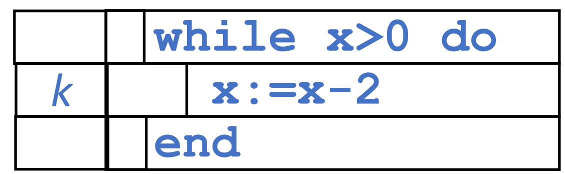
Pretty-printing



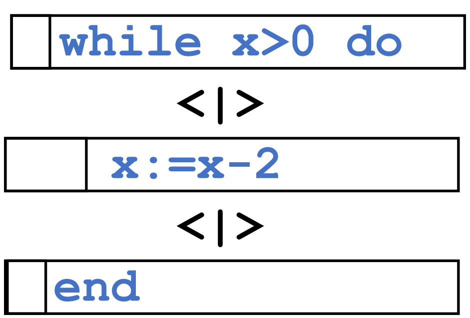
a **`Layout`** `[(Int,String)]`

`text s`

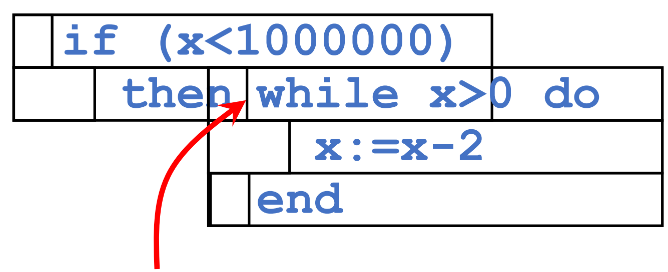
`nest k l`



`l <|> l'`



`l <-> l'`



Position **`l'`** *at the end of the last line of* **`l`**, *preserving layout*

Semantics

```
newtype Layout = Layout [(Int,String)]
  deriving (Eq, Ord)

text s = Layout [(0,s)]

nest k (Layout nxs) =
  Layout [(n+k,x) | (n,x) <- nxs]

Layout nxs <|> Layout nxs' = Layout $ nxs ++ nxs'

Layout nxs <-> Layout ((n2,x2):nxs') = Layout $
    init nxs ++ [(n1,x1++x2)] ++
      [(n1+length x1+n'-n2,x') | (n',x') <- nxs']
    where (n1,x1) = last nxs
```

Two key operations

$\texttt{sep}\ [l_1,\ l_2,\ \ldots\ l_n]$

$l_1\ \texttt{<->}\ \texttt{sp}\ \texttt{<->}\ l_2\ \texttt{<->}\ \ldots\ \texttt{<->}\ l_n$

$\quad\texttt{where sp = text " "}$

OR

$$
\begin{array}{ll}
 & l_1 \\
\texttt{<|>} & l_2 \\
\texttt{<|>} & \ldots \\
\texttt{<|>} & l_n
\end{array}
$$

$\texttt{best}\ l$

```
nestSig = [
  monoType (Proxy :: Proxy Layout),
  con "nest" (nest :: Int -> Layout -> Layout
  ]
```

```
*PP> quickSpec nestSig
== Functions ==
nest :: Int -> Layout -> Layout

== Laws ==
  1. nest x (nest y z) = nest y (nest x z)
```

```
nestSig = [
  monoType (Proxy :: Proxy Layout),
  con "nest" (nest :: Int -> Layout -> Layout),
  background [
    con "+"    ((+)    :: Int -> Int -> Int),
    con "0"    (0      :: Int)
    ]
  ]
```

```
*PP> quickSpec nestSig
…
== Laws ==
  1. nest 0 x = x
  2. nest x (nest y z) = nest y (nest x z)
  3. nest (x + y) z = nest x (nest y z)
```

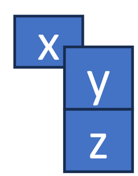
```
aboveSig = [
  background nestSig,
  con "<|>"  ((<|>) :: Layout -> Layout -> Layout)
  ]
```

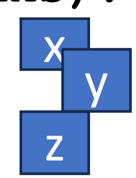
```
*PP> quickSpec aboveSig
…
== Laws ==
  1. (x <|> y) <|> z = x <|> (y <|> z)
  2. nest x y <|> nest x z = nest x (y <|> z)
```

```
besideSig = [
  background aboveSig,
  con "<->"  ((<->) :: Layout -> Layout -> Layout)
  ]
```

```
*PP> quickSpec besideSig
…
== Laws ==
  1. x <-> nest y z = x <-> z
  2. nest x y <-> z = nest x (y <-> z)
  3. (x <-> y) <-> z = x <-> (y <-> z)
  4. x <|> (y <-> z) = (x <|> y) <-> z
```

```
*PP> quickCheck $ \x y z -> x<->(y<|>z) === (x<->y)<|>z
*** Failed! Falsified (after 3 tests and 5 shrinks):
Layout [(0,"a")]
Layout [(0,"")]
Layout [(0,"")]
Layout [(0,"a"),(1,"")] /= Layout [(0,"a"),(0,"")]
```





```
textSig = [
  background besideSig,
  con "text" (text :: String -> Layout),
  background [
    con "++" ((++) :: String -> String -> String),
    con "\"\"" ("" :: String)
    ],
  withMaxTermSize 10
  ]
```

```
*PP> quickSpec textSig
…
== Laws ==
  1. x <-> text "" = x
  2. text xs <-> text ys = text (xs ++ ys)
  3. text "" <-> (text xs <|> x) = text xs <|> x
  4. (text "" <-> x) <|> (text "" <-> x) =
       text "" <-> (x <|> x)
  5. text "" <-> ((text xs <-> x) <|> y) =
       (text xs <-> x) <|> y
```

```
lengthSig = [
  background textSig,
  con "length" (length :: String -> Int)
  ]
```

```
*PP> quickSpec lengthSig
…
== Laws ==
  1. length "" = 0
  2. length (xs ++ ys) = length (ys ++ xs)
  3. length xs + length ys = length (xs ++ ys)
  4. text xs <|> nest (length xs) x =
       text xs <-> (text "" <|> x)
  5. text (xs ++ ys) <|> nest (length xs) x =
       text xs <-> (text ys <|> x)
  6. text xs <-> (nest (length xs) x <|> x) =
       (text xs <-> x) <|> (text "" <-> x)
  7. (text xs <-> x) <|> nest (length xs) y =
       text xs <-> ((text "" <-> x) <|> y)
  8. text xs <-> (nest (length xs) (text ys) <|> x) =
       text (xs ++ ys) <|> x
```

From the pretty-printer paper…

```
data Doc =
    Text String
  | TextAbove String Doc
  | Nest Int Doc
  | Union Doc Doc
  | Empty
```

The implementation was *derived* by applying the algebraic laws we've seen.

JFP **27**, e18, 49 pages, 2017.
doi:10.1017/S0956796817000090

Quick specifications for the busy programmer

NICHOLAS SMALLBONE, MOA JOHANSSON,
KOEN CLAESSEN and MAXIMILIAN ALGEDHED
Chalmers University of Technology, Gothenburg, Sweden
(*e-mails:* nicsma@chalmers.se, moa.johansson@chalmers.se, koen@chalmers.se,
algehed@chalmers.se)

Abstract

QuickSpec is a theory exploration system which tests a Haskell program to find equational properties of it, automatically. The equations can be used to help understand the program, or as lemmas to help prove the program correct. QuickSpec is largely automatic: the user just supplies the functions to be tested and QuickCheck data generators. Previous theory exploration systems, including earlier versions of QuickSpec itself, scaled poorly. This paper describes a new architecture for theory exploration with which we can find vastly more complex laws than before, and much faster. We demonstrate theory exploration in QuickSpec on problems both from functional programming and mathematics.

1 Introduction

Formal specifications are a powerful tool for understanding programs. For example, 1995

The Design of a Pretty-printing Library

John Hughes

Chalmers Tekniska Högskola, Göteborg, Sweden.

1 Introduction

On what does the power of functional programming depend? Why are functional programs so often a fraction of the size of equivalent programs in other languages? Why are they so easy to write? I claim: because functional languages support software reuse extremely well.

Programs are constructed by putting program components together. When we discuss reuse, we should ask

- What kind of components can be given a name and reused, rather than reconstructed at each use?
- How flexibly can each component be used?

Every programming language worthy of the name allows sections of a program with identical control flow to be shared, by defining and reusing a procedure. But 'programming idioms' — for example looping over an array — often cannot be defined as procedures because the repeated part (the loop construct) contains a varying part (the loop body) which is different at every instance. In a functional language there

Property-based testing

- Is an effective practical technique *and* an active research area.
- There are tools for practically every programming language.
- There is much more to discover!
- Above all, it's great *fun!*

Don’t write tests!

Generate them