Extrapolate: generalizing counterexamples of functional test properties

Rudy Braquehais
University of York, UK
rudy@braquehais.org

Colin Runciman
University of York, UK
colin.runciman@york.ac.uk

ABSTRACT

This paper presents a new tool called Extrapolate that automatically generalizes counterexamples found by property-based testing in Haskell. Example applications show that generalized counterexamples can inform the programmer more fully and more immediately what characterises failures. Extrapolate is able to produce more general results than similar tools. Although it is intrinsically unsound, as reported generalizations are based on testing, it works well for examples drawn from previous published work in this area.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

enumerative property-based testing, systematic testing, functional programming, Haskell.

ACM Reference Format:
Rudy Braquehais and Colin Runciman. 2017. Extrapolate: generalizing counterexamples of functional test properties. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages, August 30-September 1, 2017, Bristol, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205368.3205371

1 INTRODUCTION

Most programmers are familiar with the following situation: a failing test case has been discovered during testing; but it is not immediately apparent what more general class of tests would trigger the same failure. The programmer may resort to painstaking step-by-step reevaluation of the reported failure in the hope of realizing where a fault lies. In this paper, we examine a less explored approach: the *generalization* of failing cases informs the programmer more fully and more immediately what characterises such failures. This information helps the programmer to locate more confidently and more rapidly the causes of failure in their program. We present Extrapolate, a tool to generalize counterexamples of test properties in Haskell. Several example applications demonstrate the effectiveness of Extrapolate.

IFL 2017, August 30-September 1, 2017, Bristol, United Kingdom

Example 1.1. Consider the following faulty `sort` function:

```
sort :: Ord a => [a] -> [a]
sort []     =  []
sort (x:xs) =  sort (filter (< x) xs)
            ++ [x]
            ++ sort (filter (> x) xs)
```

The function `sort` should have the following properties

```
prop_sortOrdered :: Ord a => [a] -> Bool
prop_sortOrdered xs = ordered (sort xs)

prop_sortCount :: Ord a => a -> [a] -> Bool
prop_sortCount x xs = count x (sort xs) == count x xs
```

that together completely specify `sort`.

If we pass both properties to an established property-testing library, such as QuickCheck [11] or SmallCheck [33], we get something like:

```
> check (prop_sortOrdered :: [Int] -> Bool)
+++ OK, passed 500 tests.

> check (prop_sortCount :: Int -> [Int] -> Bool)
*** Failed! Falsifiable (after 4 tests):
0 [0,0]
```

That is, `prop_sortCount 0 [0,0] = False`. If instead we test using Extrapolate, then for the failing property, in addition to a specific counterexample, Extrapolate prints:

```
Generalization:
x (x:x:_)
```

Some values have been generalized: the specific value `0` does not matter, `prop_sortCount x (x:x:_) = False` for any integer `x`; the tail value `_` does not affect the result. Extrapolate also prints:

```
Conditional Generalization:
x (x:xs)  when  elem x xs
```

This hints that our faulty `sort` function fails for lists with repeated elements. We return to this example in §4.1. □

In general, if Extrapolate finds a counterexample, then the property is definitely false. Any specific counterexample displayed by Extrapolate is valid. A displayed generalization may or may not be valid. However, a generalization is only displayed if a specific counterexample has first been found and displayed. So Extrapolate never incorrectly reports a property to be false, even though a generalized counterexample may be invalid (or *loose*, cf. §6).

1.1 Contributions

The main contributions of this paper are:

(1) methods using automated black-box testing to generalize counterexamples of functional test properties by replacing constructors with variables, where these variables may be repeated or subject to side-conditions;
(2) the design of the Extrapolate library, which implements these methods in Haskell and for Haskell test properties;
(3) a selection of small case-studies, investigating the effectiveness of Extrapolate;
(4) a comparative evaluation of generalizations performed by Extrapolate and similar tools for Haskell.

Despite the Haskell setting of the implementation and experiments, we expect similar techniques to be applicable in other functional programming languages.

1.2 Road-map

This paper is organized as follows:
§2 describes how to use Extrapolate;
§3 describes how Extrapolate works internally;
§4 presents example applications and results;
§5 discusses related work;
§6 evaluates Extrapolate in comparison with similar tools;
§7 draws conclusions and suggests future work.

2 HOW EXTRAPOLATE IS USED

Extrapolate is a library (loaded by "`import Test.Extrapolate`"). Unless they already exist, instances of the `Listable` (§3.1) and `Generalizable` (§3.2) typeclasses are declared for needed user-defined datatypes (step 1). The `check` function is then applied to each test property (step 2).

Step 1. Provide class instances for used-defined types. Extrapolate needs to know how to generate test values for property arguments — this capability is provided by instances of the `Listable` typeclass (§3.1). Extrapolate also needs to manipulate the structure of test values so that it can perform its generalization procedure — this capability is provided by instances of the `Generalizable` typeclass (§3.2). Extrapolate provides instances for most standard Haskell types and a facility to derive instances for user-defined data types using Template Haskell [34]. For applications in which values of algebraic datatypes can be freely constructed, as there are no constraining data invariants, writing `deriveGeneralizable ''<Type>` is enough to create the necessary instances. Extrapolate's online documentation explains how to define these instances manually.

Step 2. Call the property-checking function. The function `check` tests properties. If counterexamples are found, it reports them and any candidate generalizations.

Example 1.1 (revisited). Figure 1 shows the program used to obtain the results in §1. □

When exploring conditional generalizations, Extrapolate limits conditions by size. Size is defined as the number of symbols in an expression added to the sizes of constants as defined by LeanCheck's

```haskell
import Test.Extrapolate

sort :: Ord a => [a] -> [a]
sort []     =  []
sort (x:xs)  =  sort (filter (< x) xs)
             ++ [x]
             ++ sort (filter (> x) xs)

prop_sortOrdered :: [Int] -> Bool
prop_sortOrdered xs  =  ordered (sort xs)
  where
  ordered :: [Int] -> Bool
  ordered (x:y:xs)  =  x <= y && ordered (y:xs)
  ordered _         =  True

prop_sortCount :: Int -> [Int] -> Bool
prop_sortCount x xs  =  count x (sort xs)
                     == count x xs
  where
  count :: Int -> [Int] -> Int
  count x  =  length . filter (== x)

main :: IO ()
main = do
  check prop_sortOrdered
  check prop_sortCount
```

Figure 1: Full program applying Extrapolate to properties of `sort` used to obtain the results in §1.

`Listable` enumeration (§3.1). This definition is similar to the one used in our previous work on Speculate [5]. By default, Extrapolate:

- tests properties for up to 500 value assignments;
- considers side conditions up to size 4;
- uses Speculate to find equivalences between expressions of up to size 4 avoiding many redundant equivalent conditions (§3.3).

Extrapolate allows variations of these default settings. The number of configured tests affects the runtime linearly so long as the underlying `Listable` enumeration has linear runtime. The size limit of side conditions affects runtime exponentially so it should be adjusted with caution. Our choice of default parameters aims for a runtime of a few seconds. For all the examples in §4 Extrapolate terminates in less than 10 seconds.

3 HOW EXTRAPOLATE WORKS

Extrapolate works in three steps:

(1) it tests properties searching for counterexamples, and if any is found, steps 2 and 3 are performed (§3.1);
(2) it tries to generalize counterexamples by substituting variables for constants (§3.2);
(3) it tries to generalize counterexamples by introducing variables subject to side conditions (§3.3).

Throughout this section we shall use the following example to illustrate how each step of Extrapolate works.

Example 3.1. In Haskell, the `Data.List` module declares the function `nub`, that removes duplicate elements from a list, keeping only their first occurrences. Consider the following *incorrect* property about `nub`:

```
prop_nubid :: [Int] -> Bool
prop_nubid xs  =  nub xs == xs
```

When Extrapolate's `check` function is applied to the above property, it produces the following output:

```
> check prop_nubid
*** Failed!  Falsifiable (after 3 tests):
[0,0]

Generalization:
x:x:_

Conditional Generalization:
x:xs  when  elem x xs
```

Coincidentally, the result for this example is similar to the result we saw in §1 — but it is simpler. □

3.1 Searching for counterexamples

To test properties searching for counterexamples, Extrapolate uses LeanCheck [2] which defines the `Listable` typeclass used to generate test values *enumeratively*.

We have used LeanCheck previously in other tools including Speculate [5], a tool we use when generating conditional generalizations (§3.3). More information on LeanCheck and the `Listable` typeclass can be found in its Haddock documentation [3] and in §4.1 and §4.2 of [4].

Example 3.1 (1st revisit). When Extrapolate's `check` function is applied to `prop_nubid`, it is tested for each of the list arguments: `[]`, `[0]`, `[0,0]`. The property fails for the third list `[0,0]` and Extrapolate produces its first two lines of output:

```
*** Failed!  Falsifiable (after 3 tests):
[0,0]
```
□

3.2 Unconditional generalization

Listing generalizations. After finding a counterexample, Extrapolate lazily lists all of its possible generalizations, from most general to least general, formed by replacing one or more subexpressions with variables. Generality order is not total as some generalizations are incomparable. So we consider candidate replacements left-to-right.

Example 3.1 (2nd revisit). Recall the reported counterexample to `prop_nubid`: `[0,0]`, or, more verbosely, `0:0:[]`. Its generalizations from most general to least general are: `xs`, `x:xs`, `x:y:xs`, `x:x:xs`, `x:y:[]`, `x:x:[]`, `x:0:xs`, `x:0:[]`, `0:xs`, `0:x:xs`, `0:x:[]` and `0:0:xs`. □

Testing generalizations. For each of these generalizations, Extrapolate tests for a configured number of value assignments whether the property *always* fails. It reports the first generalization for which this test succeeds. Although this process is unsound, as it is based on testing, it works well for example applications drawn from previous published work (§4). Any variables that appear only once in generalized counterexamples are reported as "_".

Example 3.1 (3rd revisit). The first three generalizations are not counterexamples as there are possible assignments of values for which the property returns `True`:

- `xs` — `prop_nubid []` `= True`
- `x:xs` — `prop_nubid (0:[])` `= True`
- `x:y:xs` — `prop_nubid (0:1:[])` `= True`

The fourth generalization `x:x:xs` is tested and the property fails for all tested assignments of values to the variables `x` and `xs`. So, Extrapolate produces its third and fourth lines of output:

```
Generalization:
x:x:_
```

Since the variable `xs` appears only once in the generalized counterexample, it is reported as "_". □

3.3 Conditional generalization

Background functions. Before trying to discover conditional generalizations we must first decide which *background* functions are allowed to appear in conditions.

The larger the number of functions in the background, the longer Extrapolate will take to produce a conditional generalization. So, we refrain from including a large set such as the entire Haskell Prelude. If the algorithm used is ever refined to something faster we may be able to include a larger set by default (§7).

Each type has a default list of functions to be included in the background. These lists are declared as part of `Generalizable` typeclass instances:

- for `Int`s, `==`, `/=`, `<=`, `<`;
- for `Char`s, `==`, `/=`, `<=`, `<`;
- for `Bool`s, `==`, `/=`, `not`;
- for lists, `==`, `/=`, `<=`, `<`, `length` and `elem`;
- for `Maybe`s, `==`, `/=`, `<=`, `<`, `Just`;
- for tuples, `==`, `/=`, `<=`, `<`.

For user-defined datatypes, the implementor of instances of the `Generalizable` typeclass decides which functions to include in the background. When using `deriveGeneralizable`, by default, `Eq` instances have `==` and `/=` in the background and `Ord` instances have `<` and `<=` in the background. Additional background functions can be provided using the `` `withBackground` `` combinator. In our experiments (§4), we found it useful to include in the background functions already appearing in properties.

Whenever a property is tested, these background functions are gathered for all types and component types of arguments of the property being tested. The background of `Bool` is always included.

Background constants. Constants of the types being tested, obtained from their `Listable` instances, are also included in the background.

Example 3.1 (4^{th} revisit). The background functions used when testing `prop_nubid :: [Int] -> Bool` are

```
(==), (/=)             :: Bool -> Bool -> Bool
(==), (/=), (<=), (<)  :: Int -> Int -> Bool
(==), (/=), (<=), (<)  :: [Int] -> [Int] -> Bool
not                    :: Bool -> Bool
length                 :: [Int] -> Int
elem                   :: Int -> [Int] -> Bool
```

along with enumerated constants of `Bool`, `Int` and `[Int]` types. □

Enumerating expressions. Extrapolate uses Speculate [5] to recursively enumerate expressions formed by type-correct applications of background functions to background constants and variables. Speculate avoids generating many distinct but semantically equivalent expressions by using results of previous tests to determine equivalence classes of subexpressions, and term-rewriting to normalize subexpressions to canonical forms. For further details see [5]. Expressions are enumerated lazily and limited by the configured maximum size. This enumeration process is done only once for each property that has a counterexample.

Enumerating candidate conditions. Expressions of Boolean type are used as candidate side-conditions.

Example 3.1 (5^{th} revisit). With Extrapolate configured to consider conditions up to size 3, Speculate returns the following 24 conditions

(1) `p`	(9) `xs <= ys`	(17) `x /= y`
(2) `False`	(10) `xs < ys`	(18) `x /= 0`
(3) `True`	(11) `elem x xs`	(19) `x < y`
(4) `not p`	(12) `elem 0 xs`	(20) `x < 0`
(5) `xs == ys`	(13) `p == q`	(21) `0 < x`
(6) `xs == []`	(14) `p /= q`	(22) `x <= y`
(7) `xs /= ys`	(15) `x == y`	(23) `x <= 0`
(8) `xs /= []`	(16) `x == 0`	(24) `0 <= x`

where `p :: Bool`; `x,y :: Int`; `xs,ys :: [Int]`. In addition, repeated variable instances of these conditions are tested when searching for candidate conditional generalizations. □

Discarding neutral side-conditions. Neutral conditions are those yielding generalizations equivalent to a simpler counterexample. For example `[x,x] when x == 0` is equivalent to simply `[0,0]`. So any condition of the form `<var> == <value>` is discarded. So too is any condition containing at least one variable v and found true only for a single assignment of a value to v.

Turning off Speculate. The use of Speculate is optional and can be turned off by: `check ‘maxSpeculateSize‘ 0`. Using Speculate has two effects in most cases:

(1) runtime is significantly reduced;
(2) detection of neutral conditions is improved.

Listing conditional generalizations. For each generalization, Extrapolate produces all possible Boolean conditions involving its variables based on the list of candidate conditions — this includes all possible variable renamings.

Testing conditional generalizations. Extrapolate tests to find side conditions under which the property fails all tests. Of these conditions, Extrapolate selects the conditions satisfied in the most test cases. Remember different side conditions may hold for distinct numbers of test values.

This is similar to what we do in Speculate [5]: there we find side-conditions to properties, whereas here we apply side-conditions to generalized counterexamples.

Example 3.1 (6^{th} revisit). Suppose Extrapolate has been configured to explore conditions of up to size 3. For the first candidate generalization, replacing `[0,0]` by `xs`, two candidate conditions are listed:

(1) `xs /= []`
(2) `elem 0 xs`

Testing shows that the candidate generalization `xs` does *not* always falsify the property under either of these conditions. For example, the property does not fail for the list `[0]`.

For the second candidate generalization, replacing `0:0:[]` by `x:xs`, Extrapolate lazily lists eight candidate conditions:

(1) `xs /= []`	(4) `x /= 0`	(7) `x <= 0`
(2) `elem x xs`	(5) `x < 0`	(8) `0 <= x`
(3) `elem 0 xs`	(6) `0 < x`	

The property only fails for all test values of the form `x:xs` under the second condition, `elem x xs`. Extrapolate reports its last two lines of output:

```
Conditional Generalization:
x:xs  when  elem x xs
```
□

4 EXAMPLE APPLICATIONS AND RESULTS

In this section, we use Extrapolate to generalize counterexamples of properties about:

- a sorting function (§4.1);
- a calculator library (§4.2);
- a serializer and parser (§4.3);
- the XMonad window manager (§4.4).

These example applications are adapted from Pike [30]. We omit just two of Pike's examples: one concerns arithmetic overflow and for the other we have incomplete information. The example applications in §4.2, §4.3 and §4.4 are respectively of small, medium and large scale. In §4.5, we use generalizations as property refinements. In §4.6 we give a summary of performance results for all these applications.

It was during our experiments with these examples that we found a simple rule of thumb for improving the results of Extrapolate: to include in the background functions occurring in properties. Automatically including these functions is left as future work (§7).

The *complete* source code for all examples described here is included in the Extrapolate distribution package. The package includes other examples not discussed here for the sake of space.

In this section, we evaluate Extrapolate on its own. Comparison with related work can be found in §6.

4.1 A sorting function: exact generalization

Carrying on from the example described in §1 and Figure 1, if we include `count` as a *background function* (§3.3),

```
> let chk = check `withConditionSize` 6
>           `withBackground` [constant "count" count]
> chk prop_sortCount
```

Extrapolate prints:

```
Conditional Generalization:
x xs  when  count x xs > 1
```

This is an exact description of the test cases that fail.

4.2 A calculator language

In this section, we use Extrapolate to find and generalize counterexamples to a property about the simple calculator language described by Pike [30].

Expressions to be calculated are represented by the datatype `Exp` and may contain integer constants, addition and division.

```
data Exp = C Int
         | Add Exp Exp
         | Div Exp Exp
```

The function `eval` evaluates `Exp`s and returns a `Maybe` value:

- `Nothing` when the calculation involves a division by 0;
- `Just` an integer otherwise.

```
eval :: Exp -> Maybe Int
eval (C i)       = Just i
eval (Add e0 e1) = liftM2 (+) (eval e0) (eval e1)
eval (Div e0 e1) =
  let e = eval e1
  in if e == Just 0
       then Nothing
       else liftM2 div (eval e0) e
```

The following function `noDiv0`[1], returns `True` when no *literal* division by `0` occurs in an expression.

```
noDiv0 :: Exp -> Bool
noDiv0 (C _)          =  True
noDiv0 (Div _ (C 0))  =  False
noDiv0 (Add e0 e1)    =  noDiv0 e0 && noDiv0 e1
noDiv0 (Div e0 e1)    =  noDiv0 e0 && noDiv0 e1
```

Using `noDiv0`, we define the following test property:

```
\e -> noDiv0 e ==> eval e /= Nothing
```

That is, if an expression contains no literal division by 0, evaluating it returns a `Just` value.

Using Extrapolate, we find a counterexample and two generalizations:

```
> check $ \e -> noDiv0 e ==> eval e /= Nothing
*** Failed! Falsifiable (after 20 tests):
Div (C 0) (Add (C 0) (C 0))

Generalization:
Div (C _) (Add (C 0) (C 0))

Conditional Generalization:
Div e1 (Add (C 0) (C 0))  when  noDiv0 e1
```

The property fails because it is not enough to test whether any denominator is a literal zero constant, we should test whether any denominator *evaluates* to zero. The generalized counterexamples provide improved information for the programmer. Specifically, constructors unrelated to the fault are generalized to variables.

To generate the above conditional generalization we manually included `noDiv0` in the list of *background* functions.

The following maximal generalizations (§6) are out-of-reach for the current implementation of Extrapolate as the conditions are too large with 9 and 16 symbols respectively.

```
Div e1 e2  when noDiv0 (Div e1 e2)
             && eval e2 == Just 0

Div e1 e2  when noDiv0 e1 && noDiv0 e2
             && e2 /= (C 0) && eval e2 == Just 0
```

At least one reader has suggested smaller expressions for these conditions. They were mistaken! Machine checking is useful!

4.3 A serializer and parser

In this section, we apply Extrapolate to the parser and pretty printer for a toy language described by Pike [30]. Programs in this language are represented by the datatype `Prog`[2], which can contain modules, functions, statements, expressions, assignments, etc. For brevity, we omit details of the implementation here, but it is included in the Extrapolate distribution package. It has two main functions:

```
show' :: Prog -> String
read' :: String -> Prog
```

The serializer (`show'`) is defined in ≈ 100 lines of code. The parser (`read'`) is defined in ≈ 200 lines of code. The parser includes a bug that switches the arguments of conjunctions.

When we test the property that serializing followed by parsing is an identity, Extrapolate reports a counterexample along with generalizations:

```
> check $ \e -> read' (show' e) == e
*** Failed! Falsifiable (after 96 tests):
Prog [] [Func (Var "a")
             [And (Int 0) (Bool False)] []]

Generalization:
Prog _ (Func _ (And (Int _) (Bool _):_) _:_)

Conditional Generalization:
Prog _ (Func _ (And e f:_) _:_)  when  e /= f
```

The reported conditional generalization clearly characterizes a set of failing cases: the property fails whenever there is an `And` expression with different operands. This characterization strongly hints at a programming error failing to distinguish operands correctly.

[1] Originally called `divSubTerms` by Pike [30]. We find it clearer to call it `noDiv0`.

[2] Originally called `Lang` by Pike [30]. We find it clearer to call it `Prog`.

4.4 XMonad

XMonad [36] is a tiling window manager written in roughly 1700 lines of Haskell code. The XMonad developers defined over a hundred test properties.

In this section, we use Extrapolate to find an artificial bug introduced by Pike [30] in XMonad.

The function. XMonad has a function

```
removeFromWorkspace ws =
  ws { stack = stack ws >>= filter (/=w) }
```

which removes a given window from a given workspace.

The bug. As described by Pike [30], we introduce an artificial bug, replacing `/=` by `==` simulating a typo:

```
  ws { stack = stack ws >>= filter (==w) }
```

The property. The following property `prop_delete` is one of XMonad's original test properties.

```
prop_delete x = case peek x of
                Nothing -> True
                Just i -> not (member i (delete i x))
```

A counterexample. In a regular enumerative property-based testing tool, we would get the following minimal counterexample for `prop_delete` (indented to improve readability):

```
> check prop_delete
Failed! Falsifiable (after 15 tests):
StackSet { current = Screen
             { workspace = Workspace
                 { tag = 0
                 , layout = 0
                 , stack = Just ( Stack { focus = 'a'
                                        , up = ""
                                        , down = ""
                                        } ) }
             , screen = 0
             , screenDetail = 0 }
         , visible = []
         , hidden = []
         , floating = fromList [] }
```

A generalization. When we pass `prop_delete` to Extrapolate, we instead get the following output (again indented to improve readability):

```
> check prop_delete
*** Failed! Falsifiable (after 15 tests):
StackSet
  (Screen (Workspace 0 0 (Just (Stack 'a' "" "")))
          0 0)
  [] [] (fromList [])

Generalization:
StackSet (Screen (Workspace _ _ (Just _)) _ _) _ _ _
```

Compare the non-generalized counterexample with its generalization. We can see quite clearly which parts are actually related to the fault: what characterizes failing cases is the presence of an optional third argument (of type `Maybe Stack`) in the argument workspace. Or, as Pike [30] explained *"it turns out what matters is having a* `Just` *value, which is the stack field that deletion works on!"*

We found it easier to exclude the record notation in the output of Extrapolate. In principle, it could be added. Later, in Table 6, we compare the results uniformly avoiding the record notation although some other tools support it.

4.5 Generalizations as property refinements

Whenever Extrapolate finds a generalised condition `C` for a property `P` to fail and we suspect that the property is incorrect rather than the code under test, we can directly derive from it a candidate variant of that property: `not C ==> P`. In this way, Extrapolate is also a tool that assists in the *refinement* of initially conjectured properties, too wide in their scope to be generally true, into more precise variants with scopes defined by conditions.

The actual antecedent introduced in such a refinement may be a simplified equivalent of '`not C`'. Or it may be a different condition, prompted and informed by C, but which the programmer conjectures to be (closer to) the exact necessary and sufficient condition for the property to hold. A programmer using testing without any generalising extrapolation has far more need for such conjectures and has to work harder to find them.

The process of extrapolated checking followed by property refinement may be iterative, terminating when testing finds no counterexample at all. There may be intermediate steps as Extrapolate generalisations are often still approximations — either too general or not general enough. Adding or strengthening an antecedent condition allows us to make progress as further testing reveals new residual counterexamples and their generalisations.

Consider the following property about the `words` function defined in the Haskell prelude:

```
prop_lengthWords :: String -> Bool
prop_lengthWords s  =  s /= "" ==>
  length (words s) == length (filter isSpace s) + 1
```

When passed `prop_lengthWords`, Extrapolate reports:

```
*** Failed! Falsifiable (after 4 tests):
" "

Generalization:
' ':_

Conditional Generalization:
c:_  when  c <= ' '
```

We forgot to account for lists that start (or end) with spaces. The conditional generalization may be puzzling. Here is a clue:

```
> [c | c <- list, c <= ' ']
" \n\t"
```

The `c` values are drawn from the `list` enumeration of test characters. The function `words` considers not only `' '`, but also `'\n'` and `'\t'` to be spaces.

If we add isSpace to the background, Extrapolate reports:

```
> let chk = check `withBackground`
>             [constant "isSpace" isSpace]
> chk prop_lengthWords
...
Conditional Generalization:
c:_  when  isSpace c
```

Using this information, we refine our property:

```
prop_lengthWords s  =
  s /= "" && not (isSpace (head s))
          && not (isSpace (last s))
  ==>
  length (words s) == length (filter isSpace s) + 1
```

Extrapolate now reports:

```
> chk prop_lengthWords
*** Failed! Falsifiable (after 43 tests):
"a  a"

Conditional Generalization:
c:' ':' ':c:""  when  not (isSpace c)
```

We forgot to account for words separated by more than one space. This becomes even clearer if we add Data.List.isInfixOf and " " to the background and run Extrapolate again on the original prop_lengthWords:

```
Conditional Generalization:
cs  when  "  " `isInfixOf` cs
```

We finally reach a correct property of the function words:

```
prop_lengthWords s  = noDoubleSpace (" " ++ s ++ " ")
  ==>
  length (words s) == length (filter isSpace s) + 1
  where
  noDoubleSpace s = and [not (isSpace a && isSpace b)
                        |(a,b) <- zip s (tail s)]
```

The number of words in a string is given by the number of spaces plus one so long as there are no leading, trailing or double spaces.

4.6 Performance Summary

Performance results are summarized in Table 1. For all example applications, Extrapolate takes at most a second to produce unconditional generalizations. For the calculator and XMonad examples, Extrapolate also takes at most a second to produce conditional generalizations. For the sorting and parser applications, Extrapolate takes respectively 5 and 9 seconds to consider conditional generalizations. The Table also includes results for other tools, to be discussed in §6.

Our tool and examples were compiled using ghc -O2 (version 8.2.1) under Linux. The platform was a PC with a 2.2Ghz 4-core processor and 8GB of RAM.

5 RELATED WORK

Since the introduction of QuickCheck [10–12], other property-based testing libraries and techniques have been developed. For Haskell, we can cite SmallCheck, Lazy SmallCheck [32, 33] and Feat [15]. For Curry, there's EasyCheck [9]. For Erlang, there's PropEr [29] and QuviQ QuickCheck [1]. For CLEAN, there's GAST [25].

Tracing and step-by-step evaluation. A lot of research has been done on tracing and step-by-step evaluation. In the realm of Haskell, we can note tools such as Freja [27, 28], Hat [7, 8, 37] and Hood [16]. These tools facilitate the process of locating faults in programs. Extrapolate on the other hand does not directly improve this process, but rather gives the programmer improved results to inform it. Except when a generalized counterexample makes it very obvious where the fault is, Extrapolate does not replace tools like Freja, Hat and Hood, but complements them. Claessen et al. [13] explore the combined use of property-based testing and tracing.

Property discovery and refinement. QuickSpec [14, 35] and Speculate [5] are tools capable of automatically conjecturing properties given a collection of Haskell functions. Like Extrapolate, these tools rely mainly on testing to achieve their results. Extrapolate does not directly conjecture properties, but its generalized counterexamples can be *seen* as properties about faults. In Example 1.1, the counterexample x (x:x:_) can be seen as the following property:

```
\x xs -> not $ prop_sortCount x (x:x:xs)
```

Extrapolate's generalization aims to find the largest test space in which the negation of the property under test succeeds. As stated in §3, Extrapolate uses Speculate internally.

FitSpec [4] is a tool capable of guiding refinements of Haskell test properties. FitSpec generates mutant variations of functions under test against a given property set, recording any surviving mutants that pass all tests. Reported surviving mutants prompt the user to amend or to add new properties making the property set stronger. FitSpec can guide refinements of properties more generally whereas Extrapolate can guide refinements of incorrect conditions of one property at a time (§4.5).

Program Synthesis. Magic Haskeller [19–22] and IGOR II [17, 18, 23] are systems for program synthesis using inductive functional programming techniques [24]. They are able to produce programs based on a limited list of input-output bindings and a set of background functions. Similarly, there are PROGOL [26], FOIL [31] and GOLEM [6] for logic programs. There is potential for the application of these systems and their techniques to generalize counterexamples: based on which test inputs pass or fail, generate a program to describe a set of counterexamples.

Lazy SmallCheck. Lazy SmallCheck [32, 33] is a property-based testing tool that uses laziness to prune the search space. Before testing fully defined test values, it tries partially defined test values: if a property fails or passes for a partially defined value, more defined variations of that value need not be tested. As a side-effect of this test strategy, Lazy SmallCheck is able to return partially defined values as counterexamples (see Table 4). These can be read as generalized counterexamples. In §6 we compare results of Lazy SmallCheck to those of Extrapolate.

Table 1: Summary of results for five different applications, testing properties with Extrapolate, SmartCheck and Lazy SmallCheck; #-symbols = #-constants + #-variables; #-constants = number of constants in the reported counterexamples; #-variables = number of variables in the reported counterexamples; generality = how general is the counterexample (○ = least general; ● = most general; × = invalid/loose); runtime = rounded elapsed time; space = peak memory residency. SmartCheck's result for the faulty XMonad example is presented as reported by Pike [30] and does not include runtime and memory figures.

Example & Property	Tool	#-symbols	#-constants	#-variables	generality	Runtime	Memory
Faulty `sort` (§1)	Ungeneralized	6	6	0	○	< 1s	23MB
`\x xs -> count x (sort xs)`	Lazy SmallCheck	6	6	0	○	< 1s	33MB
`== count x xs`	SmartCheck (min)	6	5	1	◔	< 1s	22MB
	SmartCheck (median)	12	11	1	◔	< 1s	22MB
	Extrapolate	6	2	4	◕	< 1s	26MB
	Extrapolate (side)	8	4	4	●	5s	34MB
Faulty `noDiv0` (§4.2)	Ungeneralized	8	8	0	○	< 1s	23MB
`\e -> noDiv0 e`	Lazy SmallCheck	8	7	1	◔	< 1s	33MB
`==> eval e /= Nothing`	SmartCheck (min/median)	7	6	1	×	< 1s	22MB
	Extrapolate	8	7	1	◔	< 1s	26MB
	Extrapolate (side)	10	8	2	◕	< 1s	28MB
Faulty parser (§4.3)	Ungeneralized	17	17	0	○	< 1s	25MB
`\e -> read' (show' e) == e`	Lazy SmallCheck	17	17	0	○	12s	36MB
	SmartCheck (min)	17	17	0	○	< 1s	23MB
	SmartCheck (median)	27	27	0	○	< 1s	23MB
	Extrapolate	14	7	7	◔	< 1s	27MB
	Extrapolate (side)	16	7	9	◕	9s	35MB
Faulty XMonad (§4.4)	Ungeneralized	16	16	0	○	< 1s	28MB
`prop_delete`	SmartCheck	13	9	4	◔	–	–
	Extrapolate	12	4	8	◕	< 1s	30MB
	Extrapolate (side)	15	7	8	◕	< 1s	31MB

Table 2: Extrapolate contrasted with Lazy SmallCheck and SmartCheck: ● = yes; ○ = no.

	SmartCheck	Extrapolate	Lazy SC
Random testing	●	○	○
Enumerative testing	○	●	●
Demand-driven testing	○	○	●
Generalized counterexamples	●	●	●
strict	●	●	○
partial (w/ undefined values)	○	○	●
functional generalizations	○	○	●
repeated variables	○	●	○
side conditions	○	●	○

SmartCheck. Because QuickCheck tests values randomly, it does not always return small counterexamples, but relies on *shrinking* [10] to derive smaller counterexamples from larger ones. SmartCheck [30] is an extension to QuickCheck that provides two improvements: a better algorithm for shrinking and generalization of counterexamples. SmartCheck's generalization algorithm performs both universal and existential quantification but does not allow repeated variables. SmartCheck is perhaps the most closely related tool to Extrapolate, and Pike's paper inspired the work reported here. In §6 we compare results of SmartCheck to those of Extrapolate.

Table 2 compares in summary three tools for Haskell able to report generalized counterexamples: Lazy SmallCheck, SmartCheck and Extrapolate. The key contribution of Extrapolate is allowing for repeated variables and side conditions in generalized counterexamples.

Table 3: Counterexamples for the `count` property of `sort` (Example 1.1 from §1) reported by Extrapolate, SmartCheck and Lazy SmallCheck.

Tool	Counterexample
Ungeneralized	`0 [0,0]`
Lazy SmallCheck	`0 [0,0]`
SmartCheck (min)	`4 (4:4:x0)`
SmartCheck	`9 (8:17:9:9:x0)`
Extrapolate	`x (x:x:_)`
Extrapolate (side)	`x xs when count x xs > 1`

Table 4: Counterexamples for the property involving `noDiv0` (§4.2) reported by Extrapolate, SmartCheck and Lazy SmallCheck. SmartCheck reports loose counterexamples.

Tool	Counterexample
Ungeneralized	`Div (C 0) (Add (C 0) (C 0))`
Lazy SmallCheck	`Div (C _) (Add (C 0) (C 0))`
SmartCheck	`Div x0 (Add (C (-5)) (C 5))`
Extrapolate	`Div (C _) (Add (C 0) (C 0))`
Extrapolate (side)	`Div e1 (Add (C 0) (C 0))` `when noDiv0 e1`

6 COMPARATIVE EVALUATION

We now revisit examples from §4 comparing results of Extrapolate with results of SmartCheck and Lazy SmallCheck.

Terms. The following paragraphs define some of the terms used in evaluating results of different tools: *generality* and *loose counterexamples*.

Generality. The more general the counterexample, the better it is. We consider a generalized test-case description more general than another if:

- it strictly subsumes another;
- it includes a larger set of failing test cases.

We say that a generalization is *maximal* when no more general description exists.

Loose counterexamples. When a reported generalized counterexample is *too general* and includes inputs that are *not* counterexamples, we say it is *loose*. Later in this section, we shall see examples of this.

Representatives and Median values. SmartCheck randomly tests values and does not usually report the same counterexamples. In Tables 3–6 counterexamples for SmartCheck are median representatives only. In Table 1, the numbers of symbols, constructors and variables are the median values from 1000 sample runs.

Versions used. We have used the following versions for each tool:

- Lazy SmallCheck: version of 2014-07-07 — compiled with GHC 7.8.4;
- SmartCheck: version 0.2.4 — compiled with GHC 8.0.2;
- Extrapolate: version 0.3.0 — compiled with GHC 8.2.1.

Summary Table. Table 1 summarizes all results. For *most* examples, Extrapolate gives *more general* results than either Lazy SmallCheck or SmartCheck. For *all* examples, Extrapolate gives results *at least as general* as Lazy SmallCheck and SmartCheck.

All tools usually report their results within a second, a reasonable time when testing a property. Extrapolate is slower to produce conditional generalizations for the `sort` and `parser` examples, taking respectively 5s and 9s. This increased runtime is still acceptable as generalization is not performed for every property under test but only for those that fail. There are no significant differences in memory use.

Faulty sort (§1). See Table 3. The counterexample reported by Extrapolate has the same number of symbols as the one reported by Lazy SmallCheck. Lazy SmallCheck is not able to report a generalization as the property being tested is not lazy. Extrapolate is able to generalize the tail and the initial elements of the list whereas SmartCheck[3] is only able to generalize the tail. With the function `count` in the background, Extrapolate reports a *maximal* generalization – there is no more general description of the failing cases.

The reported runtime of five seconds needed to reach the condition involving `count` is for Extrapolate with a maximum explored condition size of 6. With the default settings, Extrapolate reports the condition involving `elem` in two seconds (§1).

Calculator and faulty `noDiv0` *(§4.2).* See Table 4. Extrapolate and Lazy SmallCheck report the same generalization. The generalizations SmartCheck reports in 96% of runs are *too* general — as they include values that are *not* counterexamples if we read `==>` as a logical implication. Concerning the counterexample in Table 4:

```
> let prop e = noDiv0 e ==> eval e /= Nothing
> let x0 = (Div (C 0) (C 0))
> prop (Div x0 (Div (C (-5)) (C 5)))
True
```

With the precondition falsified, the property holds.

Faulty parser (§4.3). See Table 5. Neither Lazy SmallCheck nor SmartCheck is able to report a generalization. Extrapolate is able to report both an unconditional generalization and to improve it in a further conditional generalization. Extrapolate reports counterexamples that are smaller than those reported by other tools.

Faulty XMonad (§4.4). See Table 6. SmartCheck[4] and Extrapolate give counterexamples of almost the same number of symbols, but the Extrapolate counterexample has fewer constant constructors and more variables. SmartCheck always assigns variable names whereas Extrapolate uses "_" for unrepeated variables, making it more immediately apparent where component values do not matter. Extrapolate's conditional and unconditional generalizations are equivalent.

[3] Since SmartCheck only tries to generalize the first argument of properties (a design choice), the property had to be uncurried for it to report a generalization.
[4] Due to time constraints we have not tested Lazy SmallCheck or SmartCheck on this example. The SmartCheck counterexample is as reported by Pike [30] in the original paper about SmartCheck.

Table 5: Counterexamples for the parser property (§4.3) reported by Extrapolate, SmartCheck and Lazy SmallCheck.

Tool	Counterexample
Ungeneralized (full record syntax)	`Prog {modules = [], funcs = [Func {fnName = Var "a", args = [And (Int 0) (Bool False)], stmts = []}]}`
Ungeneralized (simplified)	`Prog [] [Func (Var "a") [And (Int 0) (Bool False)] []]`
Lazy SmallCheck	`Prog [] [Func (Var "a") [And (Int 0) (Bool False)] []]`
SmartCheck	`Prog [] [Func (Var "U") [] [Return (And (Bool False) (Int 0))]]`
Extrapolate	`Prog _ (Func _ (And (Int _) (Bool _):_) _:_)`
Extrapolate (with side condition)	`Prog _ (Func _ (And e f:_) _:_)  when  e /= f`

Table 6: Counterexamples for `prop_delete` from XMonad (§4.4) reported by Extrapolate and SmartCheck. The SmartCheck counterexample is the one reported by Pike [30] in the original SmartCheck paper.

Tool	Counterexample
Ungeneralized	`StackSet (Screen (Workspace 0 0 (Just (Stack 'a' "" ""))) 0 0) [] [] (fromList [])`
SmartCheck	`StackSet (Screen (Workspace x0 (-1) (Just x1)) 1 1) x2 x3 (fromList [])`
Extrapolate	`StackSet (Screen (Workspace _ _ (Just _)) _ _) _ _ _`
Extrapolate (side)	`StackSet (Screen (Workspace _ _ ms) _ _) _ _ _  when  Nothing /= ms`

7 CONCLUSIONS AND FUTURE WORK

Finally, we note some conclusions and avenues for future work.

7.1 Conclusions

In summary, we have presented a tool that is able to generalize counterexamples of functional test properties. As set out in §2 and §3, Extrapolate enumerates and tests generalizations, reporting the ones that fail all tests. We have demonstrated in §4 Extrapolate's applicability to a range of examples. After reviewing related work in §5, we have compared in §6 Extrapolate results with those reported by other tools.

Value of reported laws. The conjectured generalizations reported by Extrapolate are surprisingly accurate in practice, despite their inherent uncertainty in principle. They can provide helpful insights into the source of faults. Allowing repeated variables and side-conditions makes possible the discovery of generalizations previously unreachable by other tools that provide similar functionality, such as Lazy SmallCheck [32] and SmartCheck [30].

Ease of use. Extrapolate requires no more programming effort than a regular property-based testing tool such as QuickCheck [11] or SmallCheck [33]. If only standard Haskell datatypes are involved, no extra `Listable` or `Generalizable` instances are needed. If user-defined data types can be freely enumerated without a constraining data invariant, instances can be automatically derived.

7.2 Future Work

We note a number of avenues for further investigation that could lead to improved versions of Extrapolate or similar tools.

Type-by-type generalization. Although sufficiently fast for the examples we have tried, the current algorithm to find counterexamples is very naïve. It considers generalizations replacing subexpressions of several different types by variables, all in one step. We believe runtime could be reduced by switching to an iterative process where generalization happens one type at a time. First at outer types, then at inner types.

First generalizing with one variable per type. An interesting observation used in our previous work [5] and the work of Smallbone et al. [35] could be used to speed-up Extrapolate. For a property with several variables per type to be true, its one-variable-per-type instance should be true as well, for example:

$$\forall x\ y\ z.(x + y) + z = x + (y + z) \Rightarrow \forall x.(x + x) + x = x + (x + x)$$

The same is true for generalized counterexamples: if lists of the form `x:y:xs` always falsify a property, then lists of the form `x:x:xs` should as well. Based on this observation, we can test one-variable-per-type generalizations first, potentially reducing the search space.

Generalizing from several counterexamples. The current version of Extrapolate bases its generalizations on a single counterexample. As mentioned in §5, it may be possible to use techniques from inductive functional programming [24] to base its generalizations on several counterexamples. This might reduce the time needed to produce generalizations or increase their accuracy.

Multiple generalizations. Reported generalizations are derived from initial counterexamples. After finding a generalization, Extrapolate could search for other counterexamples and report additional generalizations. These might be of additional help in finding the source of faults.

Parallelism. As a way to improve performance, particularly when dealing with costly test functions, we could parallelize the testing of different enumerated generalizations among multiple processors.

Automatically include functions occurring in properties in the background. In several examples (§4), we provided functions present in the property as background functions for side conditions, improving generalized counterexamples. This could be done automatically to improve out-of-the-box results.

Alternative configuration parameters. Future versions of Extrapolate could offer improved control of configuration parameters. As briefly mentioned in §2, the size limit on side conditions is a very fragile parameter: incrementing it may greatly increase runtime. To make it easier to configure, future versions of Extrapolate could offer a time-limit parameter for when exploring conditions.

Custom generic derivation hierarchy. The improvement of counterexamples in Extrapolate by generalization can be compared with the improvement of counterexamples from QuickCheck by shrinking. In QuickCheck, though there are some proposed default generic derivation rules for shrinking, the shrinking methods are also exposed and custom shrinking functions can be declared explicitly. In Extrapolate, the background is a bit like that: there are defaults, and options to declare more. But the method of defining and searching the hierarchy of generalizations is a fixed default. Allowing it to be overridden could be one way to solve the problem of generalizations when algebraic datatype values cannot be freely generated as there are invariant constraints.

AVAILABILITY

Extrapolate is freely available with a BSD3-style license from:

- https://hackage.haskell.org/package/extrapolate
- https://github.com/rudymatela/extrapolate

This paper describes Extrapolate version 0.3.0.

ACKNOWLEDGEMENTS

We thank Rob Alexander, Tim Atkinson, John Hughes, Lee Pike and anonymous reviewers for their comments on earlier drafts.

Rudy Braquehais is supported by CAPES, Ministry of Education of Brazil (Grant BEX 9980-13-0).

REFERENCES

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with QuviQ QuickCheck. In *Erlang '06*. ACM, 2–10.
[2] Rudy Braquehais. 2015–2017. LeanCheck. https://github.com/rudymatela/leancheck. (2015–2017).
[3] Rudy Braquehais. 2016–2017. LeanCheck's Documentation. https://hackage.haskell.org/package/leancheck/docs/Test-LeanCheck.html. (2016–2017).
[4] Rudy Braquehais and Colin Runciman. 2016. FitSpec: refining property sets for functional testing. In *Haskell'16*. ACM, 1–12.
[5] Rudy Braquehais and Colin Runciman. 2017. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Haskell'17*. ACM, 40–51.
[6] R. Mike Cameron-Jones and J. Ross Quinlan. 1994. Efficient Top-down Induction of Logic Programs. *SIGART Bulletin* 5, 1 (Jan 1994), 33–42.
[7] Olaf Chitil, Maarten Faddegon, and Colin Runciman. 2016. A Lightweight Hat: Simple Type-Preserving Instrumentation for Self-Tracing Lazy Functional Programs. In *IFL 2016*. ACM, 1–14.
[8] Olaf Chitil, Colin Runciman, and Malcolm Wallace. 2001. Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *IFL 2000*. Springer, 176–193.
[9] Jan Christiansen and Sebastian Fischer. 2008. EasyCheck – Test Data for Free. In *Functional and Logic Programming*. Springer, 322–336.
[10] Koen Claessen. 2012. Shrinking and Showing Functions. In *Haskell'12*. ACM, 73–80.
[11] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP'00*. ACM, 268–279.
[12] Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. In *Haskell '02*. ACM, 65–77.
[13] Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. 2003. Testing and Tracing Lazy Functional Programs Using QuickCheck and Hat. In *AFP'03*. Springer, 59–99.
[14] Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *TAP 2010*. Springer, 6–21.
[15] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Haskell'12*. ACM, 61–72.
[16] Andy Gill. 2001. Debugging Haskell by Observing Intermediate Data Structures. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 1. Also in Haskell'00.
[17] Martin Hofmann and Emanuel Kitzelmann. 2010. I/O Guided Detection of List Catamorphisms: Towards Problem Specific Use of Program Templates in IP. In *PEPM'10*. ACM, 93–100.
[18] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. 2010. Porting Igor II from Maude to Haskell. In *Approaches and Applications of Inductive Programming (AAIP'09). Revised Papers*. Springer, 140–158.
[19] Susumu Katayama. 2004. Power of Brute-Force Search in Strongly-Typed Inductive Functional Programming Automation. In *PRICAI 2004: Trends in Artificial Intelligence (LNAI 3157)*. Springer, 75–84.
[20] Susumu Katayama. 2005. Systematic search for lambda expressions. In *TFP 2005*.
[21] Susumu Katayama. 2010. Recent Improvements of MagicHaskeller. In *AAIP 2009. Revised Papers (LNCS 5812)*. Springer, 174–193.
[22] Susumu Katayama. 2012. An Analytical Inductive Functional Programming System That Avoids Unintended Programs. In *PEPM'12*. ACM, 43–52.
[23] Emanuel Kitzelmann. 2007. Data-driven induction of recursive functions from input/output-examples. In *Approaches and Applications of Inductive Programming (AAIP'07)*. 15–26.
[24] Emanuel Kitzelmann. 2010. Inductive Programming: A Survey of Program Synthesis Techniques. In *AAIP'09*. Springer, 50–73.
[25] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. 2003. GAST: Generic Automated Software Testing. In *IFL 2003, LNCS 2670*. Springer, 84–100.
[26] Stephen Muggleton. 1995. Inverse entailment and PROGOL. *New Generation Computing* 13, 3 (Dec 1995), 245–286.
[27] Henrik Nilsson. 1998. *Declarative Debugging for Lazy Functional Languages*. Ph.D. Dissertation.
[28] Henrik Nilsson and Jan Sparud. 1997. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering* 4, 2 (Apr 1997), 121–150.
[29] Manolis Papadakis and Konstantinos Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Erlang '11*. ACM, 39–50.
[30] Lee Pike. 2014. SmartCheck: Automatic and Efficient Counterexample Reduction and Generalization. In *Haskell'14*. ACM, 59–70.
[31] J. R. Quinlan and R. M. Cameron-Jones. 1993. FOIL: A midterm report. In *ECML-93: European Conference on Machine Learning*. Springer, 1–20.
[32] Jason S. Reich, Matthew Naylor, and Colin Runciman. 2013. Advances in Lazy SmallCheck. In *IFL'13*. Springer, 53–70.
[33] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Haskell'08*. ACM, 37–48.
[34] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Haskell'02*. ACM, 1–16.
[35] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. 2017. Quick specifications for the busy programmer. *Journal of Functional Programming* 27 (2017).
[36] Don Stewart and Spencer Sjanssen. 2007. XMonad. In *Haskell '07*. ACM, 119–119.
[37] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. 2001. Multiple-View Tracing for Haskell: a New Hat. In *Haskell'01*. ACM, 182–196.