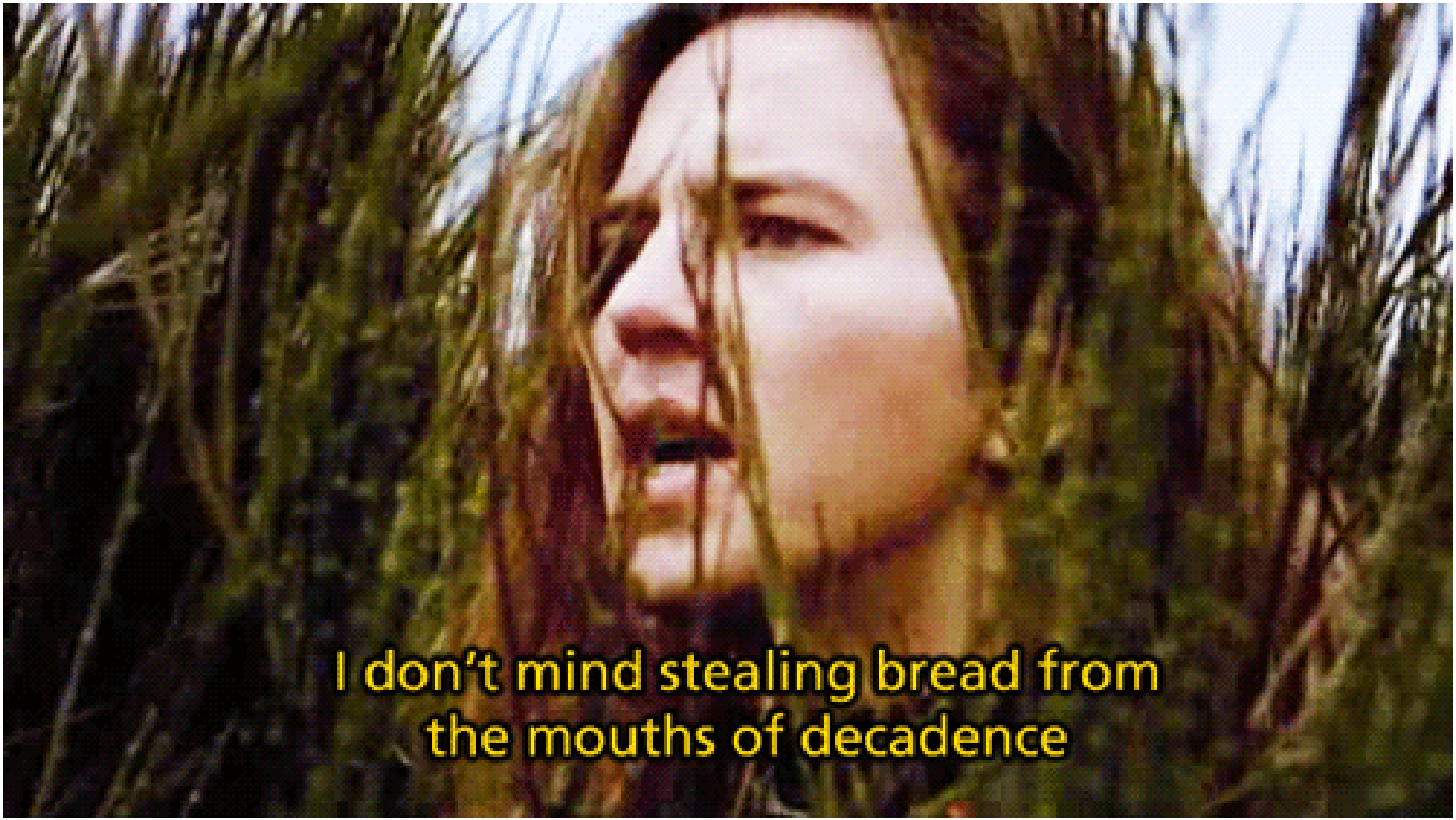
Generating Circumstances

Zeeshan Lakhani

Software Engineer at Basho Technologies,Inc | Founder/Organizer Papers We Love
@zeeshanlakhani

1-9-2015 (Codemash)

What I Listened To



#TDDDEADYO

CodeMash 2015

08 JAN 2015

Is TDD Dead?

By DHH FOWLER

Test-first fundamentalism is like abstinence-only sex ed An unrealistic, ineffective morality campaign for self-loathing and shaming.

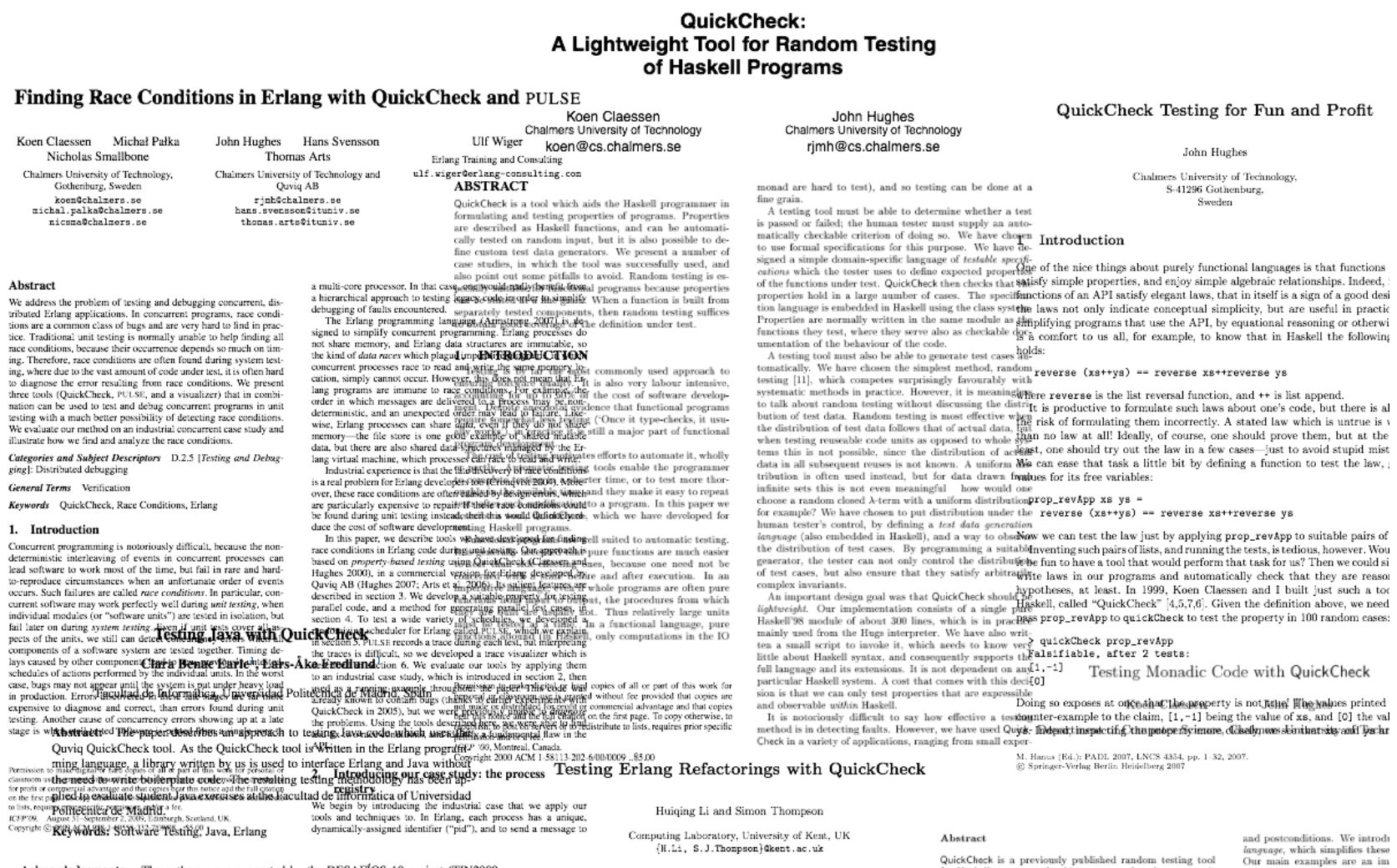
Long live testing.

tion Council, worldwide moose numbers are expected to grow markedly on last year due to the traditional moose strongholds of Canada and the United States, with the larger developing moose ecologies also poised to make gains. The largest percentagege increase in moose will likely come from China", says McRobson, The Chinese government has invested heavily in moose infrastructure over the past decade, and their committment to macrofauna is beginning to pay dividends". Since 2004 China has expanded moose pasture from 1.5% of arable land to nearly 3.648% and moose numbers are expected to rise to 60,000 making China a net moose dred million billion.

Europe's rise as an international moose power will slow slightly this year as a response to the European Union's move towards standardising the European moose. Stringent quality controls are holding back the development of the eastern european populations compared to last year when they contributed significantly to europe's strong growth figures. Norway, which is not an EU member but has observer status, strengthed in numbers relative to the Euro area with numbers of Norweigian moose, known locally as elk" expected to rise for the tenth consecutive year, particularly thanks to a strong showing in [1]

[1] http://bit.ly/1ALmSGC

In Papers



QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen, Chalmers University of Technology, koen@cs.chalmers.se
John Hughes, Chalmers University of Technology, rjmh@cs.chalmers.se

ABSTRACT

QuickCheck is a tool which aids the Haskell programmer in formulating and testing properties of programs. Properties are described as Haskell functions, and can be automatically tested on random input, but it is also possible to define custom test data generators. We present a number of case studies, in which the tool was successfully used, and also point out some pitfalls to avoid. Random testing is especially useful for functional programs because properties often have a simple structure. When a function is built from simpler functions, then random testing suffices if its properties are first tested for the definition under test.

Finding Race Conditions in Erlang with QuickCheck and PULSE

Koen Claessen, Michał Pałka, Nicholas Smallbone — Chalmers University of Technology, Gothenburg, Sweden — koen@chalmers.se, michal.palka@chalmers.se, nicsma@chalmers.se
John Hughes, Hans Svensson, Thomas Arts — Chalmers University of Technology and Quviq AB — rjmh@chalmers.se, hans.svensson@ituniv.se, thomas.arts@ituniv.se

Abstract

We address the problem of testing and debugging concurrent, distributed Erlang applications. In concurrent programs, race conditions are a common class of bugs and are very hard to find in practice. Traditional unit testing is normally unable to help finding all race conditions, because their occurrence depends so much on timing. Therefore, race conditions are often found during system testing, where due to the vast amount of code under test, it is often hard to diagnose the error resulting from race conditions. We present three tools (QuickCheck, PULSE, and a visualizer) that in combination can be used to test and debug concurrent programs in unit testing with a much better possibility of detecting race conditions. We evaluate our method on an industrial concurrent case study and illustrate how we find and analyze the race conditions.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Distributed debugging

General Terms Verification

Keywords QuickCheck, Race Conditions, Erlang

1. Introduction

Concurrent programming is notoriously difficult, because the nondeterministic interleaving of events in concurrent processes can lead software to work most of the time, but fail in rare and hard-to-reproduce circumstances when an unfortunate order of events occurs. Such failures are called *race conditions*. In particular, concurrent software may work perfectly well during *unit testing*, when individual modules (or "software units") are tested in isolation, but fail later on during *system testing*. Even if unit tests cover all aspects of the units, we still can detect failures caused by how different components of a software system are tested together. Timing delays caused by other components can change the scheduling and interleaving of actions performed by the individual units. In the worst case, bugs may not appear until the system is put under heavy load in production. Errors found during late testing are more expensive to diagnose and correct, than errors found during unit testing. Another cause of concurrency errors showing up at a late stage is when testing non-functional properties, such as performance.

Testing Java with QuickCheck

Clara Benac Earle, Lars-Åke Fredlund — Facultad de Informática, Universidad Politécnica de Madrid, Spain

Abstract

This paper describes an approach to testing Java code which uses the Quviq QuickCheck tool. As the QuickCheck tool is written in the Erlang programming language, a library written by us is used to interface Erlang and Java without the need to write boilerplate code. The resulting testing methodology has been applied to evaluate student Java exercises at the Facultad de Informática of Universidad Politécnica de Madrid.

Keywords: Software Testing, Java, Erlang

Acknowledgements: The authors were supported by the DESAFÍOS 10 project (TIN2009-14599-C03-03) financed by the (former) Spanish Ministry of Education and Science, and the EU FP7 project PROWESS (317820).

1 Introduction

This paper describes a methodology for testing Java code which uses the property-based random testing Quviq QuickCheck tool [AHJW06], in the following often abbreviated as QuickCheck. Rather than writing individual "low-level" test cases, a QuickCheck user writes "high-level"

Testing Erlang Refactorings with QuickCheck

Huiqing Li and Simon Thompson — Computing Laboratory, University of Kent, UK — {H.Li, S.J.Thompson}@kent.ac.uk

Abstract. Refactoring is a technique for improving the design of existing programs without changing their behaviour. Wrangler is a tool built at the University of Kent to support Erlang program refactoring; the Wrangler tool is written in Erlang.

In this paper we present the use of a novel testing tool, Quviq QuickCheck, for testing the implementation of Wrangler. QuickCheck is a specification-based testing tool for Erlang. With QuickCheck, programs are tested by writing properties in a restricted logic, and using the tool these properties are tested in randomly generated test cases.

This paper first gives overviews of Wrangler and Quviq QuickCheck, then discusses the various ways in which refactorings can be validated,

QuickCheck Testing for Fun and Profit

John Hughes — Chalmers University of Technology, S-41296 Gothenburg, Sweden

Introduction

One of the nice things about purely functional languages is that functions satisfy simple properties, and enjoy simple algebraic relationships. Indeed, functions of an API satisfy elegant laws, that in itself is a sign of a good desi the laws not only indicate conceptual simplicity, but are useful in practic simplifying programs that use the API, by equational reasoning or otherwi is a comfort to us all, for example, to know that in Haskell the following holds:

```
reverse (xs++ys) == reverse xs++reverse ys
```

where `reverse` is the list reversal function, and `++` is list append.

It is productive to formulate such laws about one's code, but there is al the risk of formulating them incorrectly. A stated law which is untrue is than no law at all! Ideally, of course, one should prove them, but at the least, one should try out the law in a few cases—just to avoid stupid mist We can ease that task a little bit by defining a function to test the law, values for its free variables:

```
prop_revApp xs ys =
  reverse (xs++ys) == reverse xs++reverse ys
```

Now we can test the law just by applying `prop_revApp` to suitable pairs of inventing such pairs of lists, and running the tests, is tedious, however. Wou be fun to have a tool that would perform that task for us? Then we could si write laws in our programs and automatically check that they are reaso hypotheses, at least. In 1999, Koen Claessen and I built just such a to Haskell, called "QuickCheck" [4,5,7,6]. Given the definition above, we need pass `prop_revApp` to `quickCheck` to test the property in 100 random cases:

```
> quickCheck prop_revApp
Falsifiable, after 2 tests:
[1,-1]
[0]
```

Testing Monadic Code with QuickCheck

Doing so exposes at once that the property is not true! The values printed counter-example to the claim, `[1,-1]` being the value of `xs`, and `[0]` the val

M. Hanus (Ed.): PADL 2007, LNCS 4354, pp. 1–32, 2007.
© Springer-Verlag Berlin Heidelberg 2007

In the Year 2000...

We have designed a simple domain-specific language of testable specifications which the tester uses to define expected properties of the functions under test.

We have chosen to put distribution under the human tester's control, by defining a test data generation language...

We have taken two relatively old ideas, namely specifications as oracles and random testing, and found ways to make them easily available ...

Don't Write Tests[2]

- One Feature - O(n)
- Pairs of Features - O(n2) - quadratic
- Triples of Features - O(n3) - cubic

[2] John Hughes - `http://bit.ly/1rDOGr3`

Thinking in Specifications[3]

- A "roundtrip", e.g encode/decode?
- An existing implementation with similar behavior
- A relationship between inputs/outputs?
- A set of client interactions with a platform

[3] the @seancribbs - `http://bit.ly/1wZm0I6`

Sample Properties[3]

- Reversing a list twice should equal the original list.
- Reversing and sorting a list should preserve its length.
- Popping an element from a queue should reduce its size by one

Defining Truth?[4]

- Invariant

?SOMETIMES(N,Prop)

A property which tests Prop repeatedly N times, failing only if all of the tests fail. In other words, the property passes if Prop sometimes passes. This is used in situations where test outcomes are non-deterministic, to search for test cases that consistently fail...

fails(Prop::property()) -> property()

A property which succeeds when its argument fails. Sometimes it is useful to write down properties which do not hold (even though one might expect them to). This can help prevent misconceptions.

[4] Notes on Erlang QC - `http://bit.ly/14CmDBF`

The Only Sure Thing in Computer Science

- Everything is a tradeoff.[5]
  - QC Tradeoffs: Duration | Assertion Power

[5] http://bit.ly/1xP7uIg

A Personal Story

- ClojureWest 2014
- John Hugues / Reid Draper

#BEZERKER

```
;; Release-Base and Release-Link are at the top of this file. This is
;; so Media-Link can correctly depend on Release-Link.
(def Release-New
  (assoc Release-Base
    ;; TODO: This should probably be optional, but I really want csync to send it
    ;; so leaving it as required just for now.
    :slug c/Slug
    :name c/NonEmptyString
    :artists (s/both [c/Simple-Account-Link] c/NonEmptyCollection)
    ;; TODO: This should probably be optional, but I really want csync to send it
    ;; so leaving it as required just for now.
    :created-by c/Simple-Account-Link
    (s/optional-key :copyright) (s/maybe c/Localized-Map)
    :media (s/both [(s/either c/Simple-Track-Link c/Simple-Video-Link)]
                   c/NonEmptyCollection)
    (s/optional-key :purchase) c/Simple-Link
    (s/optional-key :label) c/Simple-Account-Link
    (s/optional-key :images) c/Images
    (s/optional-key :description) c/Localized-Map
    (s/optional-key :created-with) c/Simple-Account-Link
    (s/optional-key :uploaded-with) c/Simple-Account-Link
    (s/optional-key :hearted) s/Bool
    (s/optional-key :pro-id) s/Str
    (s/optional-key :pro-slug) s/Str))

(def Release-Existing
  (assoc Release-Base
    :name c/NonEmptyString
    :slug c/Slug
    :prior-slugs [c/Slug]
    :created-at sc/ISO-Date-Time
    :created-by a/Account-Link
    :artists [a/Account-Link]
    :label (s/maybe a/Account-Link)
    :images c/Images
    :hearted s/Bool
    :copyright (s/maybe c/Localized-Map)
    :description c/Localized-Map
    :media [(s/either Track-Link Video-Link)]
    :sharing c/Simple-Link
    :total-hearts s/Int
    ;; :total-plays s/Int
    :created-with (s/maybe a/Account-Link)
    :uploaded-with (s/maybe a/Account-Link)
    :purchase (s/maybe c/Simple-Link)
    :pro-id (s/maybe s/Str)
    :pro-slug (s/maybe s/Str)
    ;; :license License-Link
    :hearted-by c/Simple-Link
    ;; :listened-to c/Simple-Link
    ))

(def Playlist-Base
  {:name c/NonEmptyString
   (s/optional-key :tags) c/Tags
   (s/optional-key :duration-seconds) (s/maybe s/Int)})

(def Playlist-Link
  (assoc Playlist-Base
    :url c/URL
    :created-by a/Account-Link
    :images c/Images
    :sharing c/Simple-Link
    :pro-id  (s/maybe s/Str)
    :pro-slug (s/maybe s/Str)
    ;; :total-hearts s/Int
    ;; :total-plays s/Int
    ))
```

```
(def Mix-Existing
  (assoc Mix-Base
    :slug c/Slug
    :prior-slugs [c/Slug]
    :created-at sc/ISO-Date-Time
    :created-by a/Account-Link
    :artists [a/Account-Link]
    :release (s/maybe Release-Link)
    :label (s/maybe a/Account-Link)
    :images c/Images
    :hearted s/Bool
    :copyright (s/maybe c/Localized-Map)
    ;; :license c/Simple-Link
    :description c/Localized-Map
    :source-tracks [{:track Track-Link
                     :start-time-seconds s/Int}]
    :sharing c/Simple-Link
    :total-hearts s/Int
    ;; :total-plays s/Int
    :purchase (s/maybe c/Simple-Link)
    :created-with (s/maybe a/Account-Link)
    :uploaded-with (s/maybe a/Account-Link)
    :recorded-date (s/maybe sc/ISO-Date-Time)
    :hearted-by c/Simple-Link
    ;; :listened-to c/Simple-Link
    ))
```

quickcheck in the wild

- test.check (clojure)
- Quickcheck - Haskell
- Erlang Quickcheck (from QuviQ)
- ScalaCheck
- JSVerify
- FsCheck (for .NET)
- ...

Specification - The Transpose of a Transposed Matrix is the Original Matrix

(A^T)^T = A

```
(def transpose-of-transpose-prop
  (prop/for-all [m matrix-gen]
  (= m (transpose (transpose m)))))

(quick-check 50 transpose-of-transpose-prop)

;; Results:
{:result true, :num-tests 50, :seed 1405444353915}
```

```clojure
(def matrix
  [[1 2]
   [3 4]])

(transpose matrix)

;; Results:
[[1 3]
 [2 4]]
```

```clojure
(def matrix-gen
  (gen/such-that
   not-empty
   (gen/vector
    (gen/tuple gen/int gen/int gen/int))))

(gen/sample matrix-gen 10)

;; Results:
([[0 0 1]]
 [[0 -1 0]]
 [[0 2 1] [0 1 -2]]
 [[1 0 1] [0 1 -3] [0 4 2] [2 -3 4]]
 [[-3 5 -4] [3 -3 -2] [-2 3 -2] [-3 -5 -3] [0 -3 -1]]
 [[2 -5 -3] [-4 -3 5] [-4 -3 -4] [-4 4 3]]
 [[-4 4 5] [-4 2 0] [5 -6 0] [2 -3 5] [-6 -3 -5]]
 [[-5 2 -3] [-2 -2 5]]
 [[-1 3 -5] [5 -3 -2] [7 4 -7] [7 -3 -2] [1 -3 8]]
 [[-5 -3 -3] [2 7 -3]])
```

gen/fmap

gen/fmap allows us to create a new generator by applying a function to the values generated by another generator[6]

```
(def gen1
  (gen/fmap (fn [n] (* n 2)) gen/nat))

(gen/sample gen1)

;; Results
(0 2 4 2 0 10 10 6 8 8)
```

[6] test.check docs - http://bit.ly/1xWxQ9R

gen/bind

gen/bind allows us to create a new generator based on the value of a previously created generator[6]

- Generator a -> (a -> Generator b) -> Generator b

```
(def gen2
  (gen/bind gen1 (fn [v]
                   (gen/hash-map :codemash
                                 (gen/return v)))))
(gen/sample gen2)

;; Results
({:codemash 0} {:codemash 2} {:codemash 0}
 {:codemash 4} {:codemash 6} {:codemash 10}
 {:codemash 2} {:codemash 14} {:codemash 12}
 {:codemash 16})
```

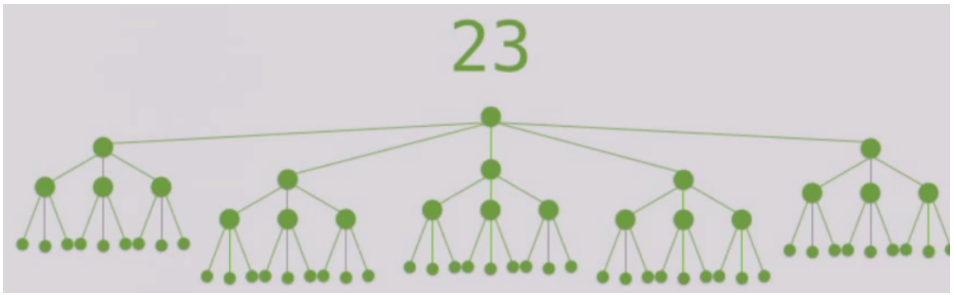
seed

Running Against the Same Set of Test Cases

```
(def prop-correct
  (prop/for-all [v gen1]
                (> 100 v)))

(tc/quick-check 100 prop-correct :seed 1420668333773)
```

Shrinking[7]

Shrink trees are lazily generated for each generated result value



Remember our property (> 100 v)?

```
{:result false, :seed 1420668333773, :failing-size 63,
 :num-tests 64, fail [108],
 :shrunk {:total-nodes-visited 17, :depth 2,
          :result false,
 :smallest [100]}}
```

[7] video - http://bit.ly/1zYwPwa

schema->gen[10] for reasons

- Turn types, generics, schemas into generated data.
- My use-case: Prismatic Schema[8, 9]
- Work with Properties Testing API Workflow
- Regression Testing Out of the Box

[8] `http://bit.ly/1BTaPW4`
[9] another example - Herbert - `http://bit.ly/1IxJs5V`
[10] `http://bit.ly/1tSakMP`

```
(def s-vector
  [(s/one s/Bool "first")
   (s/one s/Num "second")
   (s/one #"[a-z0-9]" "third")
   (s/optional s/Keyword "maybe")
   s/Int])

;; (true 3.0 "r"
;; :_1:r98l:Y!:npG-*:ZLyx4*+?:+I7:y08577B5D:392_:!1-+2:8-aMu7
;; 1 7 3 -4)

(def s-hashmap-with-hashmap
  {:foo s/Int
   :baz s/Str
   :bar {:foo s/Int}
   :far {(s/optional-key :bah) s/Bool}
   s/Keyword s/Num})

;; {:foo 0,
;;  :baz "%?I\"",
;;  :bar {:foo 9},
;;  :far {:bah false},
;;  :j 1.0,
;;  :O*37L:?K7+43:M?!?U_DIl*:GS90Ky**11 2.0}

(s/check s-hashmap-with-hashmap datum)
```

A more realistic shrink

- Trying to Model Recursive Data Types[11]

```
{:foo -1.53125, :baz {:foo -3.0,
 :baz {:baz {:baz {:baz {:baz {:foo -1.1891892}}}}}}}

;; :smallest [{:foo 0.0, :baz {:foo 0.0,
;;                             :baz {:baz {:foo 0.0}}}}]
```

[11] http://bit.ly/1sc221b

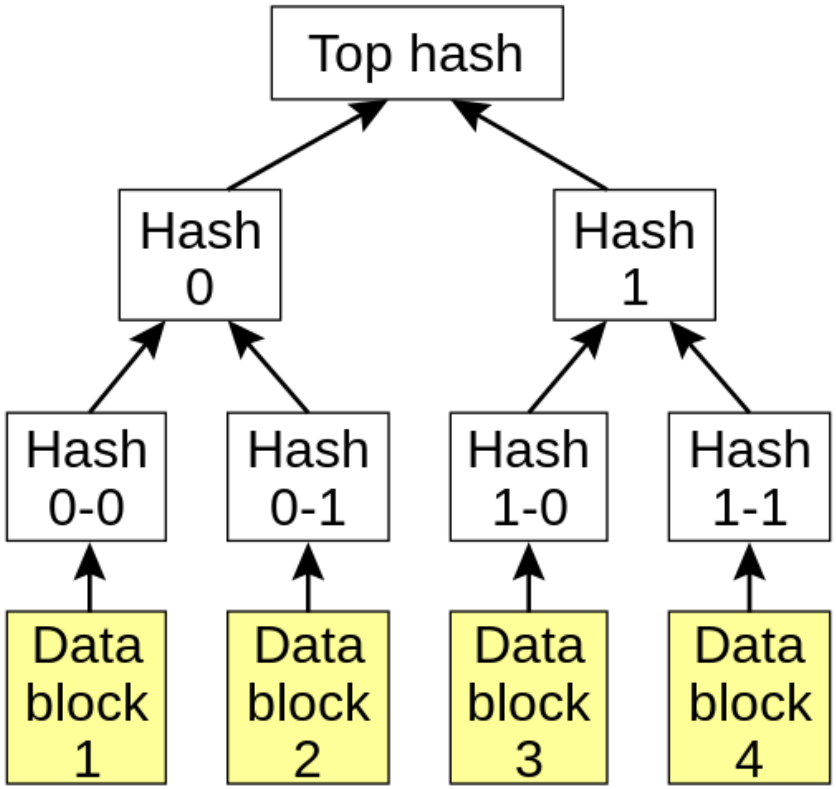
Multiple Dispatch

```
(defmethod schema->gen* schema.core.One
  [e]
  (schema->gen (:schema e)))

(defmethod schema->gen* schema.core.RequiredKey
  [e]
  (gen/return (:k e)))

(defmethod schema->gen* schema.core.OptionalKey
  [e]
  (gen/return (:k e)))

(defmethod schema->gen* schema.core.Maybe
  [e]
  (gen/one-of
   [(gen/return nil)
    (schema->gen (:schema e))]))
```

Composing Generators

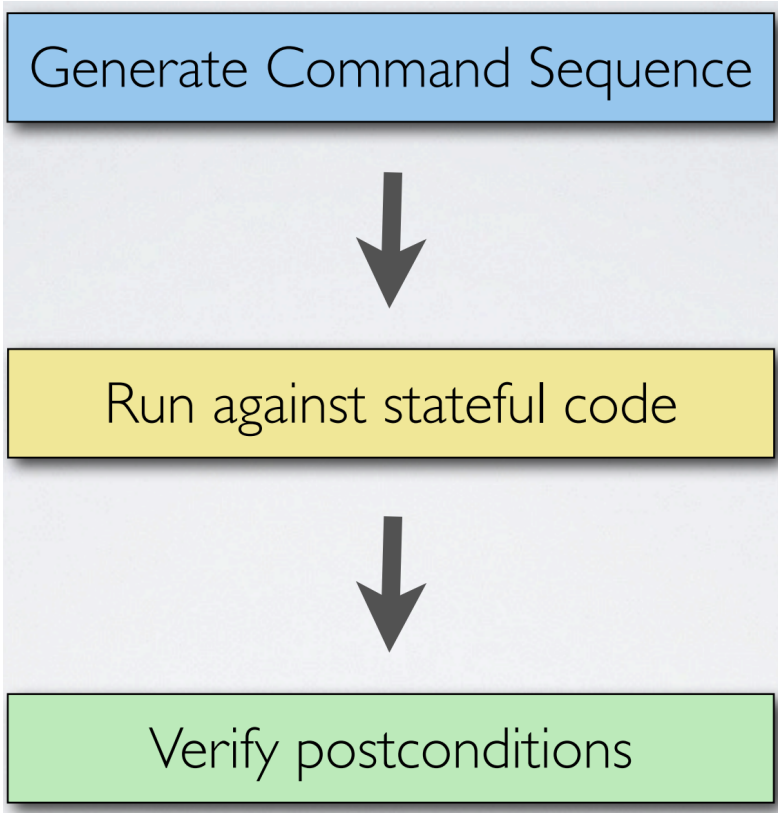
```
(defmethod schema->gen* clojure.lang.Sequential
  [e]
  (let [[ones [repeated]]
        (split-with #(instance? schema.core.One %) e)
        [required optional]
        (split-with (comp not :optional?) ones)]
    (g/apply-by
     (partial apply concat)
     (g/one-of
      (apply gen/tuple (map schema->gen required))
      (g/apply-by
       (partial apply concat)
       (apply gen/tuple
              (map schema->gen
                   (concat required optional)))
       (if repeated
         (gen/vector (schema->gen repeated))
         (gen/return []))))))))
```

Changing Gears



eqc_statem

Model State Transitions (as a FSM) -> Assert Against Implementation[12]



[12] @jtuple - http://bit.ly/1tOfzaR

Side Effects

- Requires knowledge about the context and its possible histories[13]
- Symbolic values are generated during test generation and dynamic values are computed during test execution
  - dynamic state is computed at runtime
- next_state callback operates during both test generation and test execution

[13] http://bit.ly/14CyorP

```
%% ====================================================================
%% Notes
%% ====================================================================

%% [1] Earle, Clara Benac, and Lars-Ake Fredlund."Testing Java with QuickCheck."

%% This is a very basic eqc_statem test that I've updated a bit, dealing with
%% adding|cons'ing to a list and making sure those added values are members of a
%% list.

%% ====================================================================
%% Code
%% ====================================================================

-module(stateful_sm).
-compile(export_all).

-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").

setup() ->
    io:format("Setup Components If Need Be.~n"),
    ok.

cleanup() ->
    io:format("TearDown Components if Need Be.~n"),
    ok.

test() ->
    test(100).

test(N) ->
    setup(),
    try eqc:quickcheck(numtests(N, prop_codemash()))
    after
        cleanup()
    end.
```

```erlang
%% Initialize State
initial_state() -> [].

%% ------ Grouped operator: add
%% @doc add_command - Command generator
-spec add_command(S :: eqc_statem:symbolic_state()) ->
        eqc_gen:gen(eqc_statem:call()).
add_command(S) ->
    {call, ?MODULE, add, [S, nat()]}.

%% @doc add_pre - Precondition for add
-spec add_pre(S :: eqc_statem:symbolic_state(),
              Args :: [term()]) -> boolean().
add_pre(S, _Args) ->
    S /= undefined.

%% @doc add_next - Next state function
-spec add_next(S :: eqc_statem:symbolic_state(),
               V :: eqc_statem:var(),
               Args :: [term()]) -> eqc_statem:symbolic_state().
add_next(S, _Value, [_, N]) ->
    [N|S].

%% @doc add_post - Postcondition for add
-spec add_post(S :: eqc_statem:dynamic_state(),
               Args :: [term()], R :: term()) -> true | term().
add_post(S, [_, N], Res) ->
    [N|S] =:= Res.

%% @doc - Perform add action
-spec add(list(), non_neg_integer()) -> list().
add(AList, N) ->
    [N|AList].
```

```erlang
%% ------ Grouped operator: is_member
%% @doc is_member_command - Command generator
-spec is_member_command(S :: eqc_statem:symbolic_state()) ->
        eqc_gen:gen(eqc_statem:call()).
is_member_command(S) ->
    {call, ?MODULE, is_member, [S, nat()]}.

%% @doc is_member_pre - Precondition for is_member
-spec is_member_pre(S :: eqc_statem:symbolic_state(),
                    Args :: [term()]) -> boolean().
is_member_pre(S, _Args) ->
    S /= undefined.

%% @doc is_member_next - Next state function
-spec is_member_next(S :: eqc_statem:symbolic_state(),
                     V :: eqc_statem:var(),
                     Args :: [term()]) -> eqc_statem:symbolic_state().
is_member_next(S, _Value, _Args) ->
    S.

%% @doc is_member_post - Postcondition for is_member
-spec is_member_post(S :: eqc_statem:dynamic_state(),
                     Args :: [term()], R :: term()) -> true | term().
is_member_post(S, [_, N], Res) ->
    Res == lists:member(N, S).

%% @doc - Perform is_member action
-spec is_member(list(), non_neg_integer()) -> boolean().
is_member(S, N) ->
    lists:member(N, S).
```

```
%% ====================================================================
%% Invariants
%% ====================================================================

-spec invariant(eqc_statem:dynamic_state()) -> boolean().
invariant(S) when length(S) >= 0 ->
    true;
invariant(S) when length(S) > 0 ->
    FirstNum = hd(S),
    is_number(FirstNum) andalso FirstNum >= 0;
invariant(_) ->
    false.

%% Property Test
prop_codemash() ->
    ?FORALL(Cmds,commands(?MODULE),
            aggregate(command_names(Cmds),
                      begin
                      {H, S, Res} = run_commands(?MODULE, Cmds),
                      pretty_commands(?MODULE,
                                      Cmds,
                                      {H, S, Res},
                                      Res =:= ok)
                      end)).
```

```
stateful_sm:test().

%% Results:

%% Licence for Basho reserved until
%% {{2015,1,8},{16,15,57}}
%% ..........................
%% ..........................
%% ..........................
%% ..........................
%% OK, passed 100 tests

%% 51.6% {stateful_sm,is_member,2}
%% 48.4% {stateful_sm,add,2}
%% true
```

WIP: Modeling KV-YZ HashTree in Riak

```
%% ------ Grouped operator: start_yz_tree
%% @doc start_yz_tree_command - Command generator
-spec start_yz_tree_command(S :: eqc_statem:symbolic_state()) ->
        eqc_gen:gen(eqc_statem:call()).
start_yz_tree_command(_S) ->
    {call, ?MODULE, start_yz_tree, []}.

%% @doc start_yz_tree_pre - Precondition for generation
-spec start_yz_tree_pre(S :: eqc_statem:symbolic_state()) -> boolean().
start_yz_tree_pre(S) ->
    S#state.yz_idx_tree == undefined.

%% ---------------------------------------------------------------------

-spec insert_kv_tree(sync|async, obj(), {ok, tree()}) -> ok.
insert_kv_tree(Method, RObj, {ok, TreePid}) ->
    {Bucket, Key} = eqc_util:get_bkey_from_object(RObj),
    Items = [{void, {Bucket, Key}, RObj}],
    case Method of
        sync ->
            riak_kv_index_hashtree:insert(Items, [], TreePid);
        async ->
            riak_kv_index_hashtree:async_insert(Items, [], TreePid)
    end.
```

We Talking About Tests? Tests?

- Automating Selenium Actions[14]
- Generating Models
- Sample Data
- Investigation
- Assertions

[14] Kemerling - Pivotal Tracker - `http://bit.ly/1w5sXHk`

Going Further

- Formal Specifications - Thinking for Programmers[15]
- Molly - Peter Alvaro[16]
  - A system for automatically detecting errors in a program with a correctness spec, the program, and malevolent sentience as input

[15] http://bit.ly/1w5emM2
[16] http://bit.ly/1obnZLJ