Advanced Programming

QuickCheck for Erlang and Haskell

Ken Friis Larsen
`kflarsen@diku.dk`

Department of Computer Science
University of Copenhagen

October 26, 2017

Today's Program

- Testing complex data-structures
- Testing stateful programs
- QuickCheck in Haskell

QuickCheck recap

- Testing is a cost-effective way to help us assess the correctness of our code. However, writing test cases are boring (and sometimes hard).
- We need to come up with good input data — instead, *generate* random data (from a suitable distribution).
- Often we write many test for the same underlying idea — instead, write down that underlying idea (property) and *generate* test cases from that.
- QuickCheck motto: don't write a unit test-suite – *generate* it.

Testing Data Structure Libraries

- `dict`: purely functional key-value store
  - `new()`
  - `store(Key,Value,Dict)`
  - `fetch(Key, Dict)`
  - ...
- Even though Erlang exposes the internal representation, we can't really use it
  - Complex representation
  - Complex invariants
  - We'll just test the API

Keys Should Be Unique

- There should be no duplicate keys

```
no_duplicates(Lst) ->
    length(Lst) =:= length(lists:usort(Lst)).

prop_unique_keys() ->
    ?FORALL(D,dict(),
            no_duplicates(dict:fetch_keys(D))).
```

- We need a generator for `dicts`

Generating dicts

- Generate dicts using the API

```
dict_0() ->
    ?LAZY(
       oneof([dict:new(),
              ?LET({K,V,D},{key(), value(), dict_0()},
                   dict:store(K,V,D))])
      ).
```

- Generate dicts symbolically

```
dict_1() ->
    ?LAZY(
       oneof([{call,dict,new,[]},
              ?LET(D, dict_1(),
               {call,dict,store,[key(),value(),D]})])
      ).
```

Properties for Symbolic Values

```
prop_unique_keys() ->
    ?FORALL(D,dict_1(),
            no_duplicates(dict:fetch_keys(eval(D)))).
```

Improving our generator

- We can use `frequency` to generate more interesting `dicts`

```
dict_2() ->
  ?LAZY(
     frequency(
       [{1,{call,dict,new,[]}},
        {4,?LET(D, dict_2(),
           {call,dict,store,[key(),value(),D]})}]
     )
  ).
```

- We use `?LETSHRINK` to get better counterexamples

```
dict_3() ->
 ?LAZY(
   frequency([{1,{call,dict,new,[]}},
              {4,?LETSHRINK([D],[dict_3()],
                {call,dict,store,[key(),value(),D]})}])
   ).
```

Test your understanding: Generators

- Write a symbolic generator for `dict`s that will also generate calls to `dict:erase(K, D)`.
- `dict_4() ->`

```
dict_4() ->
 ?LAZY(
   frequency([{1,{call,dict,new,[]}},
              {4,?LETSHRINK([D],[dict_4()],
                  {call,dict,store,[key(),value(),D]})},
              {4,?LETSHRINK([D],[dict_4()],
                 ?LET(K, key_from(D),
                   {call,dict,erase,[K,D]}))}])
  ).

key_from(D) ->
    elements(dict:fetch_keys(eval(D))).
```

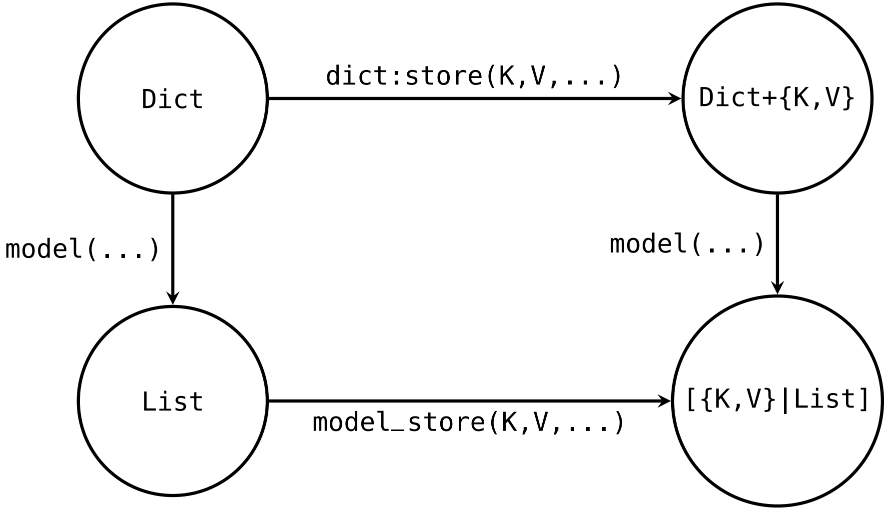
Testing Aginst Models

- A `dict` should behave like a list of key-value pairs
- Thus, we implement a model of `dicts`

```
model(Dict) ->
    dict:to_list(Dict).

model_store(K,V,L) ->
    [{K,V}|L].
```

Commuting Diagrams



Commuting Property

```
prop_store() ->
    ?FORALL({K,V,D},
            {key(),value(),dict()},
       begin
           Dict = eval(D),
           equals(model(dict:store(K,V,Dict)),
                  model_store(K,V,model(Dict)))
       end).
```

Testing Aginst The Right Model

- A `dict` should behave like a list of key-value pairs
- Thus, we implement a model of `dicts` as `proplists`
- But we must make sure that our models have a canonical form, that the lists should always be sorted.

```
model(Dict) ->
    lists:sort(dict:to_list(Dict)).

model_store(K,V,L) ->
    L1 = proplists:delete(K,L),
    lists:sort([{K,V}|L1]).
```

Test your understanding: Extending the model

- What do we need to do to support the `erase` function in the model based testing?
- Make a model version of `erase`:

```
model_erase(K,L) ->
    proplists:delete(K,L).
```

- Make a new property:

```
prop_erase() ->
    ?FORALL(D, dict(),
       ?LET(K, key_from(D),
       begin
           Dict = eval(D),
           equals(model(dict:erase(K,Dict)),
                  model_erase(K,model(Dict)))
       end)).
```

Process Registry

- Erlang provides a local name server to find node services
  - `register(Name,Pid)` associate `Pid` with `Name`
  - `unregister(Name)` remove any association for `Name`
  - `whereis(Name)` look up `Pid` associated with `Name`
- Another key-value store
  - Test against a model as before

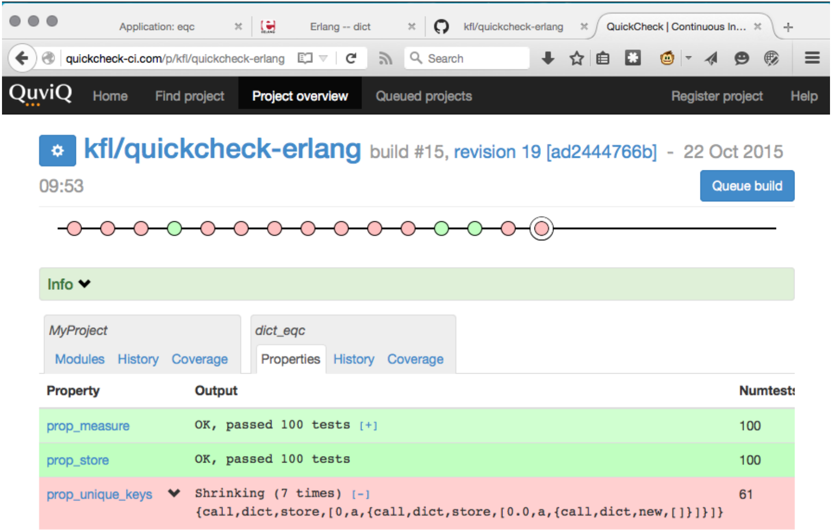
Stateful Interfaces

- The state is an implicit argument and result of every call
  - We cannot *observe* the state, and map it into a model state
  - We can *compute* the model state, using state transition functions
  - We detect test failures by observing the *results* of API calls

Testing Stateful Interfaces

- The commercial version of Erlang QuickCheck provides special support for checking stateful interfaces, this is done via callback modules.
- See the module `eqc_statem` (you can download the full version and read the documentation.)
- For use in **this course** you the library `apqc_statem` which should be API compatible with a subset of `eqc_statem`.

QuickCheck CI



Stateful Test Cases

- Test cases are sequences of *commands* taking us from one state to the next

```
prop_registration() ->
  ?FORALL(Cmds,commands(?MODULE),
     begin
         {H,S,Res} = run_commands(?MODULE,Cmds),
         cleanup(S),
         equals(Res, ok)
     end).
```

- The model (aka abstract state machine) of the system under test, is defined in a callback module.

“Statem” Behaviour

```
-type call() :: {call, module(), atom(), [expr()]}.
-type command() :: {'set', var(), call()}
                 | {'init', sym_state()}.
-type dyn_state() :: any().
-type sym_state() :: any().
-type var() :: {var, pos_integer()}.

-callback initial_state() -> sym_state().
-callback command(sym_state()) -> eqc_gen:gen(call()).
-callback precondition(sym_state() | dyn_state(), call())
                                                    -> boolean().
-callback postcondition(dyn_state(), call(), term())
                                                    -> boolean().
-callback next_state(sym_state() | dyn_state(),
                     var()       | any(),
                     call()) -> sym_state() | dyn_state().
```

Register Example: Modelling the State

```
-type proplist() :: [{atom(), term()}].

-type model_state() ::
        #{ pids := [pid()]     % list of spawned pids
         , regs := proplist()  % list of registered names
         }.
-spec initial_state() -> model_state().
initial_state() ->
    #{pids => [], regs => []}.
```

Generating Commands

- It's straightforward to generate commands:

```erlang
command(S) ->
    oneof(
      [{call,erlang,register, [name(),pid(S)]},
       {call,erlang,unregister,[name()]},
       {call,?MODULE,spawn,[]},
       {call,erlang,whereis,[name()]}]).
```

- But how do we generate a valid `pid` in a given state?

```erlang
spawn() ->
    spawn(fun() -> receive after 30000 -> ok end end).

pid(#{pids := Pids}) ->
    elements(Pids).
```

Better Generation of commands

```
command(#{pids := Pids} = S) ->
    oneof(
      [{call,erlang,register, [name(),pid(S)]}
       || Pids /= []]
      ++
      [{call,erlang,unregister,[name()]},
       {call,?MODULE,spawn,[]},
       {call,erlang,whereis,[name()]}]).
```

State Transitions

```
next_state(#{pids := Pids} = S, V,
           {call,?MODULE,spawn,[]}) ->
    S#{pids := Pids ++ [V]};

next_state(#{regs := Regs} = S, _V,
           {call,_,register,[Name,Pid]}) ->
    S#{regs := [{Name,Pid} | Regs]};

next_state(#{regs := Regs} = S, _V,
           {call, _, unregister, [Name]}) ->
    S#{regs := lists:keydelete(Name, 1, Regs)};

next_state(S,_V,_) ->
    S.
```

Callback summary

- `command` and `precondition`, used during test generation and shrinking
- `postcondition` used during test execution to check that the result of each command satisfies the properties that it should
- `initial_state` and `next_state`, used during both test generation and test execution to keep track of the state of the test case.

Meanwhile, back in the land of Haskell. . .

SkewHeap

- We have implemented a module for skew heaps, and we want to test it
- The interface

```haskell
module SkewHeap
  ( Tree(..)
  , empty
  , minElem
  , insert
  , deleteMin
  , toList
  , fromList
  , size
  )
where
```

Symbolic Expressions

```haskell
data Opr = Insert Integer
         | DeleteMin
         deriving Show

data SymbolicHeap = SymHeap [Opr]
                  deriving Show

eval (SymHeap ops) = foldl op SH.empty ops
  where op h (Insert n) = SH.insert n h
        op h DeleteMin  = SH.deleteMin h
```

```haskell
instance Arbitrary Opr where
  arbitrary = frequency [ (2, do n <- arbitrary;
                                 return (Insert n))
                        , (1, return DeleteMin)]

instance Arbitrary SymbolicHeap where
  arbitrary = fmap SymHeap arbitrary
  shrink (SymHeap oprs) = map SymHeap (shrink oprs)
```

Making a Model

```haskell
model :: SH.Tree Integer -> [Integer]
model h = List.sort (SH.toList h)

(f ‘models‘ g) h =
  f (model h) == model (g h)
```

Commuting Diagrams for Operations

```
prop_insert n symHeap =
  ((List.insert n) `models` SH.insert n) h
  where h = eval symHeap

prop_deleteMin symHeap =
  SH.size h>0 ==> (tail `models` SH.deleteMin) h
  where h = eval symHeap
```

Testing Algebraic Data Types

How can we generate random expressions for checking that Add is commutative:

```haskell
data Expr = Con Int
          | Add Expr Expr
     deriving (Eq, Show, Read, Ord)

value :: Expr -> Int
value (Con n) = n
value (Add x y) = value x + value y

prop_com_add x y = value (Add x y) == value (Add y x)
```

Generating Exprs

- Our first attempt

```
expr =  oneof [liftM Con arbitrary,
               liftM2 Add expr expr]

instance Arbitrary Expr where
  arbitrary = expr
```

  is correct,

- ... but may generate humongous expressions.
- Instead we should generate a `sized` expression

```
expr = sized exprN

exprN 0 = liftM Con arbitrary
exprN n = oneof [liftM Con arbitrary,
                 liftM2 Add subexpr subexpr]
  where subexpr = exprN (n `div` 2)
```

Test your understanding: Check that minus is commutative

- Add constructor and extend `eval`.
- Extend data generator:

```
expr = sized exprN
exprN 0 = liftM Con arbitrary
exprN n = oneof [liftM Con arbitrary,
                 liftM2 Add subexpr subexpr,
                 liftM2 Minus subexpr subexpr
                ]
  where subexpr = exprN (n `div` 2)
```

- Write a property

```
prop_com_minus x y =
  eval (Minus x y) == eval (Minus y x)
```

Shrinking in Haskell

- The `Arbitrary` type class also specify the function `shrink`

  ```
  shrink :: a -> [a]
  ```

  Which should produces a (possibly) empty list of all the possible immediate shrinks of the given value.

- For `Exprs`

  ```
  instance Arbitrary Expr where
    arbitrary = sized exprN
      where expr N 0 = ...

    shrink (Add e1 e2) = [e1, e2]
    shrink (Minus e1 e2) = [e1, e2]
    shrink _ = []
  ```

Generating functions and images

```haskell
import Test.QuickCheck
import Codec.Picture
import qualified Data.ByteString.Lazy as BL

instance Arbitrary PixelRGB8 where
  arbitrary = PixelRGB8 <$> arbitrary <*> arbitrary
                        <*> arbitrary

genImage :: Gen (Image PixelRGB8)
genImage = do
  f <- arbitrary      -- a generated function
  (x, y) <- arbitrary
          `suchThat` ( \(x,y) -> x > 0 && y > 0 )
  return $ generateImage f x y
```

https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html

Summary

- Install Quviq Erlang QuickCheck
- Use symbolic commands
- Test against models
- Be careful with your specification
- Stateful interfaces can (and should) be tested with QuickCheck

Course Evaluation

(This morning 24 out of 136 had answered)

Exam

- One week take-home project (3/11–10/11)
- Hand in via Digital Exam
- Check with OnlineTA before submission
- Max group size is **1** (one)
- (Please remember that the University have zero-tolerance policy regarding exam fraud)