Advanced Programming

Property-based Testing – Finalè

Ken Friis Larsen
`kflarsen@diku.dk`

Department of Computer Science
University of Copenhagen

October, 2021

Today's Program

- Model-based testing
- Testing stateful programs
- QuickCheck in Haskell

Part I

Model-Based Testing

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
- Generate dicts using the API

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

Generating Symbolic Call for `dicts`

- Generate symbolic calls to the `dict` API:

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

- We need to evaluate a symbolic call before we can use it with regular API calls:

```
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 `dicts` that will also generate calls to `dict:erase(K, D)`.
- Our first attempt could be:

```
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(),
                  {call,dict,erase,[K,D]}))}])
  ).
```

- However, is it interesting to try to erase *random* keys?

Parameterised Generators

- Let's try to only generate keys that are in the `dict`.
- That is, we make a parameterised generator

```
dict_5() ->
  ?LAZY(
   frequency([{1,{call,dict,new,[]}},
              {4,?LETSHRINK([D],[dict_5()],
                     {call,dict,store,[key(),value(),D]})},
              {4,?LETSHRINK([D],[dict_5()],
                  ?LET(K, key_from(D),
                     {call,dict,erase,[K,D]}))}])
    ).
key_from(D) ->
    elements(dict:fetch_keys(eval(D))).
```

- But what if `D` is empty? Then `elements` fails.
- One solution:

```
key_of(D) ->
    elements(dict:fetch_keys(eval(D)) ++ [snowflake]).
```

Let’s Sample It

```
1> eqc_gen:sample(dict_eqc:dict_5()).
{call,dict,erase, [snowflake,
     {call,dict,erase,[3,{call,dict,store,
                          [3,c,{call,dict,new,[]}]}]}]}

{call,dict,store, [-1,-1.6666666666666667,
   {call,dict,erase, [1,
      {call,dict,store, [1,14,
         {call,dict,store, [-1.1666666666666667,d,
            {call,dict,new,[]}]}]}]}]}

{call,dict,erase,[snowflake,{call,dict,new,[]}]}

...
```

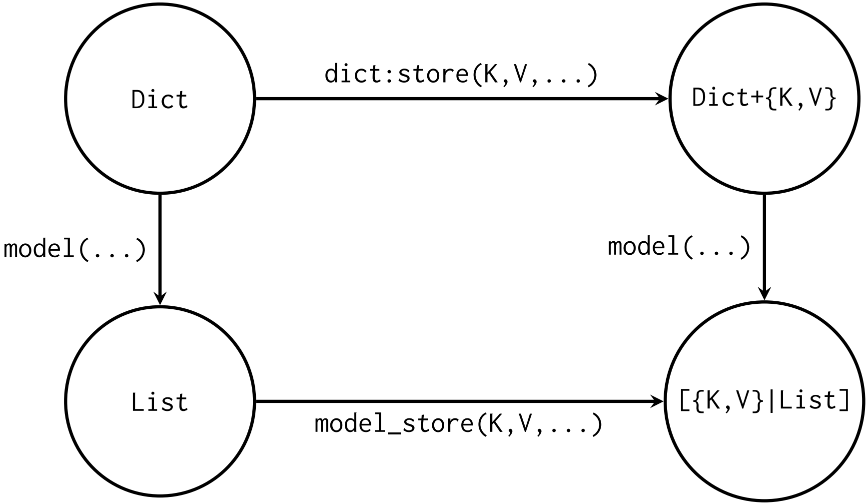
Testing Aginst Models

- A `dict` should behave like a list of key-value pairs
- Thus, we implement a model of `dict`s and a model of the `dict:store/3` function:

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

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

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).
```

That's Not Right

```
8> eqc:quickcheck(dict_eqc:prop_store()).
.Failed! After 2 tests.
{0,0, {call,dict,store, [a,0, {call,dict,erase, [snowflake,
   {call,dict,store, [0.0,0, {call,dict,store, [0,d,
      {call,dict,store, [0.0,c, {call,dict,erase, [snowflake,
         {call,dict,erase, [0.0, {call,dict,store, [0.0,0.0,
            {call,dict,erase, [snowflake,
              {call,dict,erase,
                [snowflake,{call,dict,new,[]}]}]}]}]}]}]}]}]}]}]}]}}
[{0,0},{0.0,0},{a,0}] /= [{0,0},{0.0,0},{0,d},{a,0}]
Shrinking ......x....(10 times)
{0,0,{call,dict,store,[0,a,{call,dict,new,[]}]}}
[{0,0}] /= [{0,0},{0,a}]
false
```

Testing Aginst The Right Model

- A `dict` should behave like a list of key-value pairs
- Thus, we implement a model of `dict`s 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, M) ->
    M1 = proplists:delete(K, M),
    lists:sort([ {K,V} | M1 ]).
```

Test your understanding: Extending the model

- What do we need to do to make a model-based property for testing `erase`?
- Make a model version of `erase`:

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

- Make a new property:

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

Part II

Testing Stateful Systems

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

Symbolic Commands

- Symbolic calls worked wonders. Let's generalise that!
- A *symbolic command* binds a variable to the result of a symbolic call. That is, a symbolic command has the form:

  ```
  {set, {var,N}, {call,Mod,Fun,Args}}
  ```

- So the Erlang code:

  ```
  Var1 = erlang:whereis(a),
  Var2 = erlang:register(b,Var1)
  ```

  is represented as a list of symbolic commands:

  ```
  [{set, {var,1}, {call,erlang,whereis,[a]}},
   {set, {var,2}, {call,erlang,register,[b, {var,1}]}}]
  ```

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 read the documentation online.)
- In **this course (only)** you can use the library `apqc_statem` which should be API compatible with a subset of `eqc_statem`.

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

```erlang
-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:

```
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?

```
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.
```

Pre- and Post-Conditions

- Let's start out with no pre- and post-conditions

```
precondition(_S, {call,_,_,_}) ->
    true.

postcondition(_S,{call,_,_,_},_R) ->
    true.
```

```
4> eqc:quickcheck(reg:prop_registration()).

...Failed! After 4 tests.
[{init,#{pids => [],regs => []}},
 {set,{var,1},{call,erlang,unregister,[c]}}]

Initial sym state: #{pids => [],regs => []}

V1 = erlang:unregister(c), % -> {'EXIT',{error,badarg}}
{exception,{'EXIT',{error,badarg}}} /= ok
Shrinking .x(1 times)
[{init,#{pids => [],regs => []}},
 {set,{var,1},{call,erlang,unregister,[a]}}]

Initial sym state: #{pids => [],regs => []}

V1 = erlang:unregister(a), % -> {'EXIT',{error,badarg}}
{exception,{'EXIT',{error,badarg}}} /= ok
```

Better Precondition

```
unregister_ok(#{regs := Regs}, Name) ->
    proplists:is_defined(Name, Regs).

register_ok(#{regs := Regs}, Name, _P) ->
    not (proplists:is_defined(Name, Regs))

precondition(S,{call,_, unregister, [Name]}) ->
   unregister_ok(S, Name);
precondition(S,{call,_, register, [Name, P]}) ->
   register_ok(S, Name, P);
precondition(_S,{call,_,_,_}) ->
    true.
```

But not good enough

```
6> eqc:quickcheck(reg:prop_registration()).
................Failed! After 17 tests.
[{init,#{pids => [],regs => []}},
 {set,{var,1},{call,reg,spawn,[]}},
 {set,{var,2},{call,erlang,register,[b,{var,1}]}},
 {set,{var,3},{call,erlang,register,[c,{var,1}]}},
 {set,{var,4},{call,erlang,whereis,[d]}},
 {set,{var,5},{call,erlang,unregister,[c]}}]

[...]
Shrinking ...x(3 times)
[...]

V1 = reg:spawn(), % -> <0.1604.0>
V2 = erlang:register(a, V1), % -> true
V3 = erlang:register(b, V1), % -> {'EXIT',{error,badarg}}
{exception,{'EXIT',{error,badarg}}} /= ok
false
```

The Right Precondition

```
unregister_ok(#{regs := Regs}, Name) ->
    proplists:is_defined(Name, Regs).

register_ok(#{regs := Regs}, Name, P) ->
    not (proplists:is_defined(Name, Regs))
    and not (lists:member(P, [ Val || {_K, Val } <- Regs ])).

precondition(S, {call,_, unregister, [Name]}) ->
   unregister_ok(S, Name);
precondition(S, {call,_,register, [Name, P]}) ->
   register_ok(S, Name, P);
precondition(_S, {call,_,_,_}) ->
   true.
```

Success

```
eqc:quickcheck(reg:prop_registration()).
........................................
OK, passed 100 tests

38.3% {reg,spawn,0}
37.0% {erlang,whereis,1}
16.1% {erlang,register,2}
8.6% {erlang,unregister,1}
```

What About Negative Testing?

- By having strong preconditions we only have positive testing.
- Alternatively, check in `postcondition` that we get errors when we expect them:

```
precondition(_S,{call,_,_,_}) -> true.

postcondition(S,{call,_,register,[Name, P]},Res) ->
     case Res of
          {'EXIT',_} -> not register_ok(S,Name, P);
          true       ->     register_ok(S,Name, P) end;
postcondition(S,{call,_,unregister,[Name]},Res) ->
     case Res of
          {'EXIT',_} -> not unregister_ok(S,Name);
          true       ->     unregister_ok(S,Name) end;
postcondition(_S,{call,_,_,_},_R) ->
    true.
```

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.

Part III

Meanwhile, back in the land of Haskell…

SkewHeap

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

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

Symbolic Expressions

```
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
```

Generating Symbolic Expressions

```
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

```
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:

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

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

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

Generating Exprs

- Our first attempt

  ```
  expr =  oneof [fmap Con arbitrary,
                 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 = fmap Con arbitrary
  exprN n = oneof [ fmap Con arbitrary
                  , Add <$> subexpr <*> subexpr ]
    where subexpr = exprN (n `div` 2)
  ```

Test your understanding: Check that minus is associative

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

```
expr = sized exprN
exprN 0 = fmap Con arbitrary
exprN n = oneof [ fmap Con arbitrary
                , Add <$> subexpr <*> subexpr
                , Minus <$> subexpr <*> subexpr ]
  where subexpr = exprN (n `div` 2)
```

- Write a property

```
prop_assoc_minus x y z =
  eval (Minus x (Minus y z)) == eval (Minus (Minus x y) z)
```

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 (Con n) = map Con $ shrink n
     shrink (Add e1 e2) =
       [e1, e2] ++
       [Add e1' e2' | (e1', e2') <- shrink (e1, e2)]
     shrink (Minus e1 e2) =
       [e1, e2] ++
       [Minus e1' e2' | (e1', e2') <- shrink (e1, e2)]
  ```

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

Part IV

Summary

QuickCheck Assignment

- *Let It Be* – QuickCheck for testing an evaluator and simplifier for arithmetic expression, in Haskell.
- *A QuickCheck Mystery* – Use QuickCheck to solve mysteries, that is find bugs in different versions of the same library.

Summary

- Install Quviq Erlang QuickCheck, and `apqc_statem`
- Use symbolic commands
- Test against models
- Be careful with your specification
- Stateful interfaces can (and should) be tested with QuickCheck
- Shrinking might seem superfluous, but it’s soooo helpful