# Advanced Programming

## QuickCheck for Erlang

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

October 24, 2017

# Today's Program

- Types in Erlang
- QuickCheck, with Erlang syntax
- Testing complex data-structures

# Flamingo Behaviour

- The Flamingo behaviour:

```
-type path() :: string().
-type request() :: {path(), [{string(), string()}]}.

-callback initialise(term()) -> {ok, term()}
                              | {error, any()}.
-callback action(request(), any(), State) ->
                        {new_state, string(), State}
                      | {no_change, string()}.
```

# Flamingo Callback Module

- The `greeter` module:

```
-module(greeter).
-export([initialize/1, action/3]).

-behaviour(flamingo).

initialize(_Arg) -> {ok, nothing}.

action({_Path, [{"name", Name} | _ ]}, Server, _) ->
    {no_changes,
     lists:concat(["Greetings ", Name, "\n",
                   "You have reached ", Server])}.
```

# Checking With `dialyzer`

- First build a *Persistent Lookup Table (PLT)* database for `dialyzer`:

  ```
  $ dialyzer --build_plt --apps erts kernel stdlib crypto
  ```

  (wait a few minutes)
- Check your types with the command:

  ```
  $ dialyzer --src flamingo.erl greeter.erl
  ```
- Fix your bugs

# Types are not just for callbacks

```
-type path() :: string().
-type route() :: #{ prefix := path()
                  , group  := integer()
                  }.


-spec find_by_prefix(path(), [route()])
                    -> none | {found, route()}.

find_by_prefix(_Path, []) -> none;
find_by_prefix(Path, [#{ prefix := Pre} = Route | Rest]) ->
    case lists:prefix(Pre, Path) of
        true -> {found, Route};
        false -> find_by_prefix(Path, Rest)
    end.
```

# 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.

# Warm-up: Student Activity

- Lets' write some tests for `lists:delete`
- Try to write a property relating `lists:delete` and `lists:member`

# Warm-up: Delete an element from a list

- When you delete an element from a list, it's not there anymore:

```
prop_delete_0() ->
  ?FORALL({X, Xs}, {int(), list(int())},
    not lists:member(X, lists:delete(X, Xs))).
```

- This succeeds when we check the property a hundred times
- . . . but not when we check it thousands of times

# The Problem

- There is a problem with our specification, `delete` only removes the first occurrence of the element
- How often do we even generate relevant test cases?

```
prop_member_probability() ->
  ?FORALL({X, Xs}, {int(), list(int())},
    collect(lists:member(X, Xs), true)).
```

# Collecting Statistics

Record the number of time `X` occurs in `Xs`. Running the property a large number of times reveals that the probability that a random value appears in a random list twice is around 0.5%.

```
occurs(X, Xs) ->
    lists:foldl(fun(Y, Sum)  ->
                        if X =:= Y -> Sum + 1;
                           X =/= Y -> Sum
                        end
                end, 0, Xs).

prop_list_classification() ->
    ?FORALL({X, Xs}, {int(), list(int())},
            collect(occurs(X, Xs), true)).
```

# Use Implication to Generate more interesting Test-cases

- Only look at cases where the value appears at least once in the list.

```
prop_delete_1() ->
  ?FORALL({X, Xs}, {int(), list(int())},
  ?IMPLIES(lists:member(X, Xs),
    not lists:member(X, lists:delete(X, Xs)))).
```

- Document that we expect the property to fail (within a hundred attempts)

```
prop_delete_2() ->
  fails(prop_delete_1()).
```

# Getting The Specification Right

- What it the right specification for `delete`?
- `delete` only remove the *first* occurrence of an element

```
prop_delete_3() ->
  ?FORALL({Pre,X,Post}, {list(int()), int(), list(int())},
  ?IMPLIES(not lists:member(X, Pre),
    equals(lists:delete(X, Pre ++ [X] ++ Post),
           Pre ++ Post))).
```

# Testing Data Structure Libraries

- `dict`: purely functional key-value store
  - `new()`
  - `store(Key,Value,Dict)`
  - `fetch(Key, Dict)`
  - `...`
- "No, stop! Don't expose your `dict`"
  - 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)))).
```

# How good is our generator

- Let's make a property for measuring the quality of our generator

```
prop_measure() ->
    ?FORALL(D,dict(),
     collect(length(dict:fetch_keys(eval(D))),true)).
```

- 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]})}]
     )
  ).
```

# I need a shrink now

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

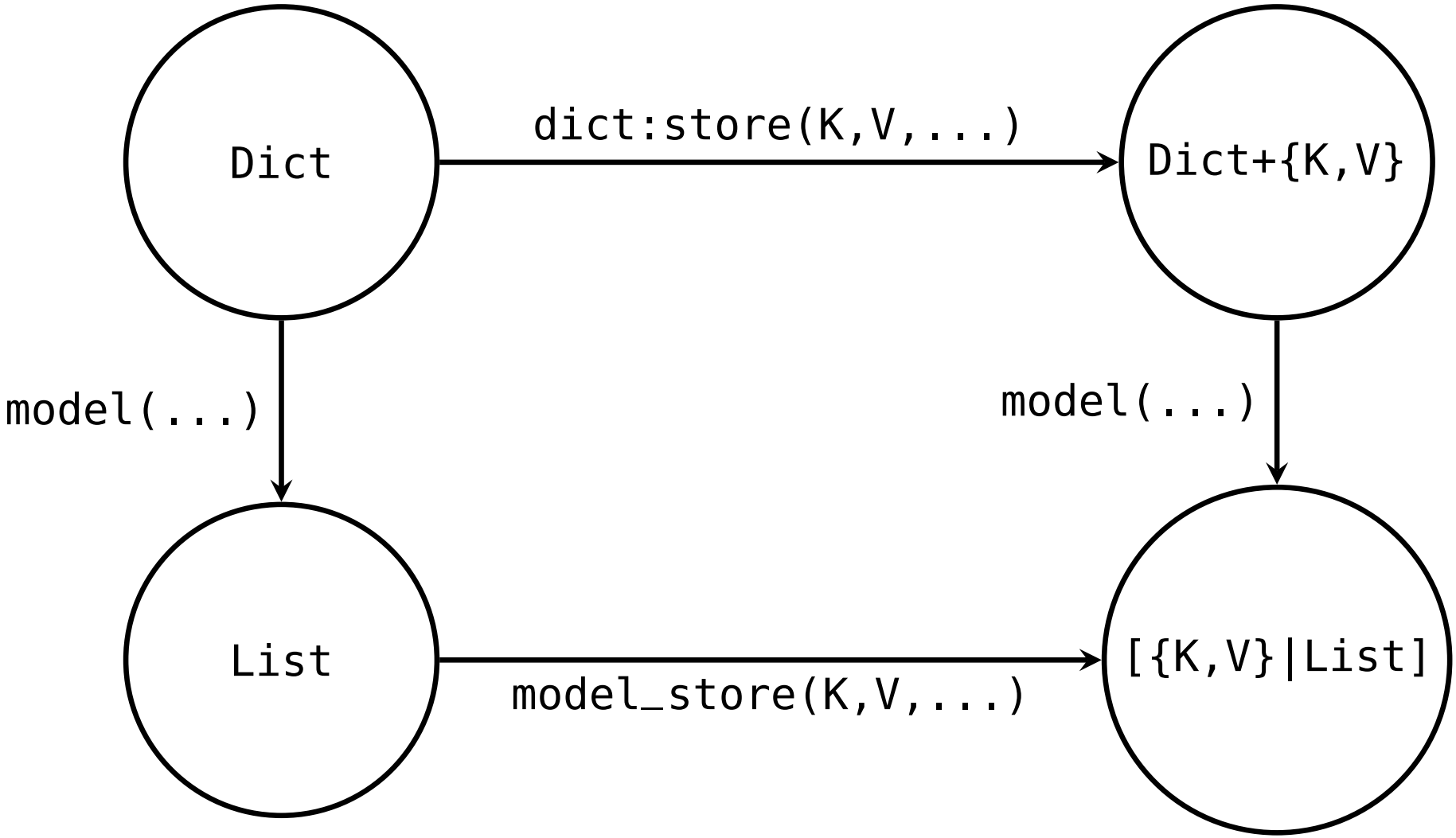
# Testing Aginst Models

- 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),
            model(dict:store(K,V,Dict))
             ==
            model_store(K,V,model(Dict)))
        end).
```

# Course Evaluation

(This morning 20 out of 136 had answered)

# Summary

- Install Quviq Erlang QuickCheck.
- Use symbolic commands
- Test against models
- Be careful with your specification
- We can test complex data structures by generating sequences of API calls
- Remember course evaluation

# 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)