Erlang/QuickCheck

Thomas Arts, IT University
John Hughes, Chalmers University
Gothenburg

A little set theory…

- Recall that $X \cup Y = Y \cup X$?

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?
- Property: $X \cup Y = Y \cup X$

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?
- Property: $\forall X. \forall Y. X \cup Y = Y \cup X$

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?
- Property: $\forall X{:}Set. \forall Y{:}Set. X \cup Y = Y \cup X$

A little set theory…

- Recall that X ∪ Y = Y ∪ X?
- Erlang has a sets library. Does this hold?
- Property: ∀ X:Set. ∀ Y:Set. X ∪ Y = Y ∪ X
- In Erlang/QuickCheck:

```
?FORALL(X,set(),
?FORALL(Y,set(),
sets:union(X,Y) == sets:union(Y,X)))
```

A little set theory…

- Recall that X ∪ Y = Y ∪ X?
- Erlang has a sets library. Does this hold?
- Property: ∀ X:Set. ∀ Y:Set. X ∪ Y = Y ∪ X
- In Erlang/QuickCheck:

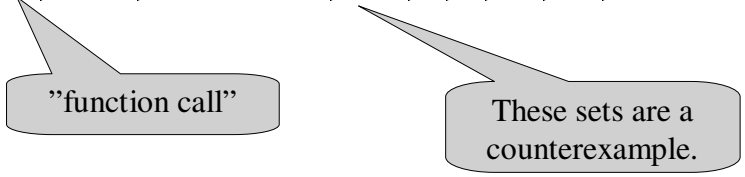
```
prop_union_commutes() ->
   ?FORALL(X,set(),
   ?FORALL(Y,set(),
   sets:union(X,Y) == sets:union(Y,X))).
```

Verifying the property

```
12> qc:quickcheck(
        setsspec:prop_union_commutes()).
```

Verifying the property

```
12> qc:quickcheck(
        setsspec:prop_union_commutes()).
.....................................
....
Falsifiable, after 45 successful tests:
{'@',sets,from_list,[[-6,7,11,10,2]]}
{'@',sets,from_list,[[7,7,1,-4,11,-7]]}
ok
```



Fixing the Property

- Sets are not represented uniquely by the `sets` library
- `union` builds two different representations of the same set

```
equal(S1,S2) ->
  lists:sort(sets:to_list(S1)) ==
  lists:sort(sets:to_list(S2)).

prop_union_commutes() ->
  ?FORALL(X,set(),
  ?FORALL(Y,set(),
  equal(sets:union(X,Y),sets:union(Y,X)))).
```

Checking the fixed property

```
15> qc:quickcheck(
        setsspec:prop_union_commutes()).
.....................................
.....................................
.................
OK, passed 100 tests
ok
```

What is QuickCheck?

- A *language* for stating properties of programs (implemented as a library of functions and macros).
- A *tool* for testing properties in randomly generated cases.

Properties

- Boolean expressions + `?FORALL` + `?IMPLIES`.

```
prop_positive_squares() ->
  ?FORALL(X,int(),X*X>=0).

prop_larger_squares() ->
  ?FORALL(X,int(),
  ?IMPLIES(X>1,
  X*X>X)).
```

A *precondition*

What are int() and set()?

- Types?

What are int() and set()?

- Types? NO!!!
- Test data generators.
  - Define a *set* of values for test data…
  - …plus a *probability distribution* over that set.
- Test data generators are defined by the programmer.

Defining generators

- We often want to define one generator in terms of another, *e.g.* squares of ints.
- But we cannot do this by writing

```
N = int(), N*N
```

Returns a test data generator, not an integer.

Result should be a generator, not an integer.

Defining generators

- We often want to define one generator in terms of another, *e.g.* squares of ints.
- But we cannot do this by writing

```
N = int(), N*N
```

- We define a *generator language* to handle generators as an ADT.

```
?LET(N,int(),return(N*N))
```

Bind a name to the *value generated.*

Convert a value to a *constant* generator.

How can we generate sets?

- An ADT can only be generated using the ADT operations.
- Choose randomly between all ways of creating a set.

A generator for sets

```
set() -> frequency([
  {6,?LET(L,list(int()),
     return({'@',sets,from_list,[L]}))},
  {6,?LET(S,set(),?LET(E,int(),
     return({'@',sets,add_element,[E,S]})))},
  {1,?LET(P,function(bool()),?LET(S,set(),
     return({'@',sets,filter,[P,S]})))},
  …]).
```

weights

?FORALL performs a call when it sees '@'

A problem with random generation

- How do we know we tested a reasonable range of cases, when we don't *see* them?

A problem with random generation

- How do we know we tested a reasonable range of cases, when we don't *see* them?
- **Simple approach**: collect statistics on test cases, so we see a *summary* of the test data.
- (A simple way to measure *test coverage*, which is a tangled topic in its own right).

An instrumented property

```
prop_union_commutes() ->
  ?FORALL(X,set(),
  ?FORALL(Y,set(),
  collect(sets:size(sets:union(X,Y)),
  equal(sets:union(X,Y),
        sets:union(Y,X))))).
```

Collect statistics on the *sizes* of the resulting sets.

Output: the distribution of set sizes

```
27> qc:quickcheck(
        setsspec:prop_union_commutes()).
........................................
........................................
................
OK, passed 100 tests
 16% 3    7% 7    3% 16   2% 9     1% 21
 11% 4    6% 12   3% 14   2% 0     1% 18
 9% 2     5% 13   3% 11   1% 20    ok
 8% 6     4% 8    3% 5    1% 10
 8% 1     3% 17   2% 24   1% 22
```

Testing concurrent programs

A simple *resource allocator*:

- `start()` – starts the server
- `claim()` – claims the resource
- `free()` – releases the resource

} in the client

These functions are called for their *effect,* not their result. How can we write QuickCheck properties for them?

Traces

- Concurrent programs generate traces of events.
- We can write properties of traces – they are lists!

Testing the resource allocator

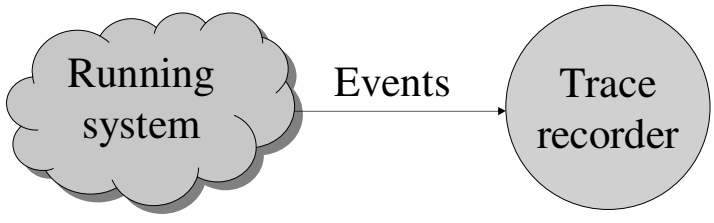
```
client() -> claim(), free(), client().
```

`clients(N)` – spawns `N` clients.

```
system(N) -> start(), clients(N).
```

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
… property of T …))
```

The trace recorder



- What should the recorded events be?
- How should we capture them?

Random traces: a problem

- What does this print?

```
test_spawn() ->
     spawn(io,format,["a"]),
     spawn(io,format,["b"]).
```

Random traces: a problem

- What does this print?

```
test_spawn() ->
     spawn(io,format,["a"]),
     spawn(io,format,["b"]).
```

- `ab` – every time!

Random traces: a problem

- What does this print?

```
test_spawn() ->
     spawn(io,format,["a"]),
     spawn(io,format,["b"]).
```

- `ab` – every time!
- But `ba` should also be a possible trace – the Erlang scheduler is too predictable!

Solution: simulate a random scheduler

- Insert calls of `event(Event)` in code under test.
  - Sends `Event` to trace recorder
  - Waits for a reply, sent in random order
- Allows the trace recorder to simulate a random scheduler.
- Answers question: which events should be recorded?

Simple example revisited

```
do(E) -> event(spawned), event(E).

?FORALL(T,
  ?TRACE(3,begin spawn(?MODULE,do,[a]),
                 spawn(?MODULE,do,[b])
           end),
  collect(rename_pids(nowaits(T)),true)))
```

Simple example revisited

```
OK, passed 100 tests
```

```
18% [{exit,{pid,1},normal},
  {event,{pid,2},spawned},
  {event,{pid,3},spawned},
  {event,{pid,2},a},
  {exit,{pid,2},normal},
  {event,{pid,3},b},
  {exit,{pid,3},normal},
  timeout]
```

```
18% [{exit,{pid,1},normal},
  {event,{pid,2},spawned},
  {event,{pid,3},spawned},
  {event,{pid,3},b},
  {exit,{pid,3},normal},
  {event,{pid,2},a},
  {exit,{pid,2},normal},
  timeout]
```

Simple example revisited

```
OK, passed 100 tests
```

```
18% [{exit,{pid,1},normal},
  {event,{pid,2},spawned},
  {event,{pid,3},spawned},
  {event,{pid,2},a},
  {exit,{pid,2},normal},
  {event,{pid,3},b},
  {exit,{pid,3},normal},
  timeout]
```

```
18% [{exit,{pid,1},normal},
  {event,{pid,2},spawned},
  {event,{pid,3},spawned},
  {event,{pid,3},b},
  {exit,{pid,3},normal},
  {event,{pid,2},a},
  {exit,{pid,2},normal},
  timeout]
```

Pids are renamed for collecting statistics

Trace recorder times out if no events happen for a while

A surprise!

```
Pid=spawn(fun()->
   event(spawned),
   event(ok) end),
event(spawn),
exit(Pid,kill),
event(kill)
```

```
1% [{event,{pid,1},spawn},
  {event,{pid,2},spawned},
  {event,{pid,2},ok},
  {event,{pid,1},kill},
  {exit,{pid,2},killed},
  {exit,{pid,2},noproc},
  {exit,{pid,1},normal},
  timeout]
```

No doubt there is a good reason…

Trace properties

- The resource allocator guarantees exclusion
- Instrumented code:

```
client() ->
  event(request),
  claim(),
  event(claimed),
  event(freeing),
  free(),
  client().
```

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event,_,claimed}))))))))))
```

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    next(?MATCHES({event,_,claimed}))))))))))
```

The trace T satisfies…

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    next(?MATCHES({event,_,claimed}))))))))))
```

…it's always true that…

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event,_,claimed}))))))))))
```

…if the current event is claimed…

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event,_,claimed}))))))))))
```

…then after this event…

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event,_,claimed}))))))))))
```

…until a freeing event happens…

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event,_,claimed}))))))))))
```

…there will be no further claimed event.

Trace property language

- Based on *linear temporal logic*
  - Logical operations:
    `tand, tor, tnot, ?TIMPLIES.`
  - Temporal operations:
    `always, eventually, next, until.`
  - Event matching operations:
    `?MATCHES, ?AFTER, ?NOW.`

A failing property

- The resource is always eventually granted.

```
prop_eventually_granted(N) ->
  ?FORALL(T,?TRACE(3,system(2)),
  satisfies(T,
  always(?AFTER({event,Pid,request},
      eventually(N,
          tor(?NOW({event,Pid2,claimed},
                        Pid==Pid2),
              ?MATCHES(more)))))).
```

A failing property

- The resource is always eventually granted.

Failing trace of 23 steps found after 80 successful tests.

```
prop_eventually_granted(N) ->
  ?FORALL(T,?TRACE(3,system(2)),
  satisfies(T,
  always(?AFTER({event,Pi
      eventually(N,
          tor(?NOW({event,Pid2,claimed},
                          Pid==Pid2),
                ?MATCHES(more))))))).
```

After at most N steps

End of the recorded trace

In progress

- Testing generic leader election behaviour
- Properties
  - Eventually a leader is elected, even in the presence of failures
  - There is always at most one elected leader

Experience

- There are as many bugs in properties as in programs!
  - QuickCheck checks for *consistency* between the two, helps improve understanding
- Random testing is effective at finding errors.
- Changes our perspective on testing
  - Not ”what cases should I test?”
  - But ”what properties ought to hold?”

QuickCheck is Fun!

Try it out!

www.cs.chalmers.se/~rjmh/ErlangQC

References

- Erlang/QuickCheck is based on a Haskell original by Claessen and Hughes.
  - *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, ICFP 2000.
  - *Testing Monadic Code with QuickCheck*, Haskell Workshop 2002.
  - *Specification Based Testing with QuickCheck*, in Fun of Programming, Palgrave, 2003.
  - *Testing and Tracing Functional Programs*, in Advanced Functional Programming Summer School, Springer-Verlag LNCS, 2002.

Questions?

Answers

(The remaining slides may be used to answer specific questions).

Random functions *are* pure functions!

```
1> F = qc:gen(qc:function(qc:nat()),10).
#Fun<qc.46.146913[illegible]>
2> F(1).
8
3> F(2).
9
4> F(3).
3
5> F(1).
8
```

Invokes a generator

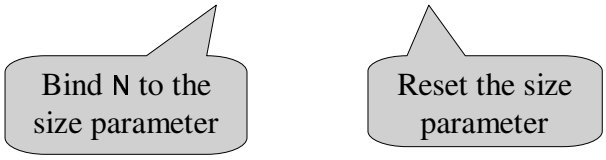
Random results

But consistent ones

Controlling sizes

- Test cases are regenerated w.r.t. a *size parameter*, which increases during testing.

```
prop_union_commutes() ->
     ?SIZED(N,resize(5*N,…))
```



- Set sizes now range up to 135 elements.