Testing Erlang Data Types with Quviq QuickCheck

Thomas Arts    Laura Castro
John Hughes

Erlang Workshop'08
Challenge

Erlang libraries supply a number of data types, but sometimes you want to design your own.

How can we ensure that we have fully tested an implementation of a home-made data type?
ARMISTICE is an information system for the insurance industry used by a large Spanish company. The system is written in Erlang.

To enable uniform way of marshalling a number of data types are (re)defined for the system and represented in a uniform way.

Example data structures: monetario, decimal, entero and logico
Data Types

*Logico* represents boolean, true as

\[ \{\text{ok}, \#\text{logico}\{\text{value} = \text{true}\}\} \]

*Entero* represents integers, division by zero as

\[ \{\text{error}, \text{division\_by\_zero}\} \]

Computations on *server side* always result in a value returned to the *client*, sometimes representing an error.
Amounts of money are based on the *decimal* data type. Some digits before and some after the "dot".
Testing

Idea: use random *decimal* values to check the operations (implement by using QuickCheck)

Generator for decimals:

```plaintext
decimal() ->
  ?LET(Tuple, {int(),nat()}, new(Tuple)).
```

Able to generate all possible decimals.
Testing

With generator for random *decimals* we can formulate a property to generate test cases:

```
prop_sum_comm() ->
  ?FORALL([D1,D2], [decimal(),decimal()],
            sum(D1,D2) == sum(D2,D1)).
```

Run QuickCheck and thousands of randomly generated tests will pass.
Testing

Which other properties do we add?
When do we have sufficiently many properties?
Testing

Which other properties do we add?
When do we have sufficiently many properties?

Use a Model

\[ \text{sum}(D1, D2) = [D1] + [D2] \]
\[ \text{subs}(D1, D2) = [D1] - [D2] \]
\[ \text{mult}(D1, D2) = [D1] \times [D2] \]
\[ \text{lt}(D1, D2) = [D1] < [D2] \]

Erlang functions

Model operations
Model Data Type

Use Erlang/C floating point implementation as model (based upon IEEE 754-1985 standard)

model(Decimal) ->
    decimal:get_value(Decimal).

Similarly *logico* modeled by booleans, *entero* by integers, etc.
Testing model equivalence

For each operator one property, e.g.:

prop_sum() ->
  ?FORALL({D1,D2},{decimal(),decimal()},
    model(sum(D1,D2)) ==
    model(D1) + model(D2)).

prop_lt() ->
  ?FORALL({D1,D2},{decimal(),decimal()},
    logico_model(lt(D1,D2)) ==
    model(D1) < model(D2)).
Testing model equivalence

We run QuickCheck....

> eqc:quickcheck(decimal_eqc:prop_sum()).
   ....Failed! After 5 tests.
   {{decimal,10000000000000000000000000000000}},
   {{decimal,11000000000000000000000000000000}}
false

Error presented in internal representation of the data structure
Hard to understand how value was obtained
Symbolic data

Use symbolic data structures instead of real data structures in test generation:

easier to analyze errors

decimal() ->
  ?LET(Tuple, {int(), nat()},
  new(Tuple)).
Symbolic data

Use symbolic data structures instead of real data structures in test generation:

easier to analyze errors

decimal() ->
  ?LET(Tuple, {int(), nat()},
       {call, decimal, new, [Tuple]}).
Symbolic data

Translate symbolic value to real value in property

prop_sum() ->
  ?FORALL(\{D1,D2\},\{decimal(),decimal()\},
      model(sum(D1,D2)) ==
      model(D1) + model(D2)).
Symbolic data

Translate symbolic value to real value in property

prop_sum() ->
  ?FORALL({SD1,SD2},{decimal(),decimal()},
  begin
    D1 = eval(SD1),
    D2 = eval(SD2),
    model(sum(D1,D2)) ==
      model(D1) + model(D2)
  end).
Testing model equivalence

We run QuickCheck....

> eqc:quickcheck(decimal_eqc:prop_sum()).

.........Failed! After 9 tests.

{{call,decimal,new,[{2,1}]},
 {call,decimal,new,[{2,2}]}}

Shrinking..(2 times)

{{call,decimal,new,[{0,1}]},
 {call,decimal,new,[{0,2}]}}

false

Thus: 0.1 + 0.2 =/= 0.3 ??
Testing model equivalence

Indeed!
Unavoidable rounding error according to IEEE 754-1985. Our model is incorrect.

> (0.1+0.2) == 0.3.
false
> (0.1+0.2) - 0.3.
5.55112e-17
Improve the model

We know that ARMISTICE decimals have 16 digits precision.

\[-\text{define}(\text{ABS}\_\text{ERROR}, 1.0\times 10^{-16}).\]
\[-\text{define}(\text{REL}\_\text{ERROR}, 1.0\times 10^{-10}).\]

\[
equiv(F_1, F_2) \rightarrow \\
\hspace{1em} \text{if } (\text{abs}(F_1 - F_2) < ?\text{ABS}\_\text{ERROR}) \rightarrow \text{true}; \\
\hspace{1em} (\text{abs}(F_1) > \text{abs}(F_2)) \rightarrow \\
\hspace{2em} \text{abs} \left( \frac{(F_1 - F_2)}{F_1} \right) < ?\text{REL}\_\text{ERROR}; \\
\hspace{1em} (\text{abs}(F_1) < \text{abs}(F_2)) \rightarrow \\
\hspace{2em} \text{abs} \left( \frac{(F_1 - F_2)}{F_2} \right) < ?\text{REL}\_\text{ERROR} \\
\text{end.}
\]

Dawson 2008
We know that ARMISTICE decimals have 16 digits precision.

\[
\text{prop\_sum()} \rightarrow \\
\quad \text{?FORALL}\{\{\text{SD1, SD2}\},\{\text{decimal(), decimal()}\}\}, \\
\quad \text{begin} \\
\quad \quad \text{D1} = \text{eval(SD1)}, \\
\quad \quad \text{D2} = \text{eval(SD2)}, \\
\quad \quad \text{equiv(model(sum(D1,D2)),} \\
\quad \quad \quad \text{model(D1) + model(D2))} \\
\quad \text{end).}
\]
Improve the model

We know that ARMISTICE decimals have 16 digits precision.

```
prop_sum() ->
    ?FORALL({SD1,SD2},{decimal(),decimal()},
        begin
            D1 = eval(SD1),
            D2 = eval(SD2),
            equiv(model(sum(D1,D2)),
            model(D1) + model(D2))
        end).
```

Property prop_sum() passes thousands of test cases.
Recursive generators

But... we are missing things!
- No 100% code coverage of new/1
- No operations combination

```
{call,decimal,sum,
 [ {call,decimal,sum,
   [ {call,decimal,mult,
     [ {call,decimal,new,[{11,"4003351"}]},
     {call,decimal,new,["-930764"}]},
     {call,decimal,new,["-2.35986"]}],
     {call,decimal,new,[-2.35986]}}],
   {call,decimal,new,[1.64783]}}}
```
Recursive generators

\texttt{decimal()} ->
\texttt{\textbackslash ?SIZED\textbar\textbackslash Size, decimal\textbar\textbackslash Size).}

\texttt{decimal(0) ->}
\{\texttt{call, decimal, new,}
\[
\texttt{[oneof([int(),}
\texttt{real(),}
\texttt{separator\textbar\textbackslash decimal\_string\textbar\textbackslash (),digits\textbar\textbackslash (),}
\texttt{[oneof([int\textbar\textbackslash (), decimal\_string\textbar\textbackslash ()],}
\texttt{oneof([nat\textbar\textbackslash (), digits\textbar\textbackslash ()]])
\[
\texttt{]}]}\}
\texttt{]}\}
\texttt{;}
\texttt{decimal\textbar\textbackslash Size) ->}
\texttt{Smaller = decimal\textbar\textbackslash Size div 2,}
\texttt{oneof([}
\texttt{decimal\textbar\textbackslash 0),}
\texttt{\{call, decimal, \texttt{sum}, [Smaller, Smaller\}],}
\texttt{\{call, decimal, \texttt{mult}, [Smaller, Smaller\}]
\texttt{])}.}
Testing model equivalence

Testing model equivalence

6> eqc:quickcheck(decimal_eqc:prop_mult()).

...............Failed! After 16 tests.

Shrinking.........................................(31 times)

{call,decimal_eqc,sum,
  [{call,decimal_eqc,sum,
    [{call,decimal_eqc,new, ["+0"]},
     {call,decimal_eqc,new, [0.00000e+0]}]},
    {call,decimal_eqc,mult,
     [{call,decimal_eqc,new, [1]},
      {call,decimal_eqc,new, [10.1400]}]}],
   {call,decimal_eqc,sum,
    [{call,decimal_eqc,mult,
     [{call,decimal_eqc,new, ["00.4"]},
      {call,decimal_eqc,new, [{"-0,000", "40"}]}]},
     {call,decimal_eqc,mult,
      [{call,decimal_eqc,new, ["40"]},
       {call,decimal_eqc,new, ["-000,000.078"]}]}}]}

false

Isn't that just zero
We have not told QuickCheck what smaller values are. We need to do that.

\[
\text{signed}(G) \rightarrow \\
\quad \text{?LETSHRINK}([S], [G], \\
\quad \quad \text{oneof}([S, "++", "--", "-++"]).
\]

\[
\text{decimal}(Size) \rightarrow \\
\quad \text{Smaller} = \text{decimal}(\text{Size} \div 2), \\
\quad \text{oneof}([ \\
\quad \quad \text{decimal}(0), \\
\quad \quad \text{?LETSHRINK}([D1, D2], [\text{Smaller}, \text{Smaller}], \\
\quad \quad \quad \{\text{call}, \text{decimal}, \text{sum}, [D1, D2]\}), \\
\quad \quad \text{?LETSHRINK}([D1, D2], [\text{Smaller}, \text{Smaller}], \\
\quad \quad \quad \{\text{call}, \text{decimal}, \text{mult}, [D1, D2]\}) \\
\quad ]).
\]
Shrinking

Now an error in the implementation can be understood:

Shrinking.........................(51 times)
{{call,decimal_eqc,new,[10.1400]},
 {call,decimal_eqc,sum,
  [{call,decimal_eqc,new,"0.4"}],
  {call,decimal_eqc,mult,
   [{call,decimal_eqc,new,"47"}],
   {call,decimal_eqc,new,"-0.078"}}}}}}

Real  -331.172
Model  -33.1172
false

factor 10 difference -> incorrect carrier propagation
Testing model equivalence

Fix the error, add subs and divs to generator and test \textbf{same property} again:

\begin{verbatim}
> eqc:quickcheck(decimal_eqc:prop_sum()).
............Failed!
After 13 tests.
Shrinking....(4 times)
Reason:
{'EXIT',{not_ok,{error,decimal_error}},
  [{common_lib,ok,1},
   {decimal_eqc,'-prop_subs/0-fun-0-',1},
   {eqc,'-forall/2-fun-4-',2},
   ...
]}}
{{call,decimal,divs,
   [{call,decimal,new,[{0,[]}]}],
   {call,decimal,new,["0"]}],
   {call,decimal,new,[0]}}
false
\end{verbatim}
Negative testing

We do want to test that division by zero results in an error... *in prop_divs, not in prop_sum*

```prolog
prop_divs() ->
    ?FORALL([{SD1, SD2}, {decimal(),decimal()}], begin
        D1 = eval(SD1),
        D2 = eval(SD2),
        case catch (model(D1)/model(D2)) of
            {'EXIT',_} ->
                is_error(divs(D1, D2));
            Value ->
                equiv(model(divs(D1, D2)), Value)
        end
    end).
```
Well-defined values

decimal() ->
  ?SIZED(Size, well_defined(decimal(Size))).

well_defined(G) ->
  ?SUCHTHAT(E, G, defined(E)).

defined(E) ->
  case catch {ok, eval(E)} of
    {ok, _}     -> true;
    {'EXIT', _} -> false
  end.
Check base case

The well_defined trick can potentially hide errors, since if generation crashes, we will never use it in a test. For the operators, this is no problem, we have one property for each.

prop_new() ->
  ?FORALL(SD,decimal(0),
  is_float(model(eval(SD)))).

no well_defined in base case generation
Check base case

The well_defined trick can potentially hide errors, since if generation crashes, we will never use it in a test. For the operators, this is no problem, we have one property for each.

\[
\text{prop}\_\text{new}() \rightarrow \\
?\text{FORALL}(SD, \text{decimal}(\theta),
\text{is}\_\text{float}(\text{model}(\text{eval}(SD))))).
\]

Finally! All properties pass the tests!!
Conclusion

We introduced a method to test Erlang data structures
Conclusion

We introduced a method to test Erlang data structures

1. Define a model
2. Generate well-defined values, work symbolic, include all productive operations
3. Write one property for each operation, consider expected failing cases
4. Fine-tune your own shrinking preferences
Conclusion

We introduced a method to test Erlang data structures

1. Define a model
2. Generate well-defined values, work symbolic, include all productive operations
3. Write one property for each operation, consider expected failing cases
4. Fine-tune your own shrinking preferences

When following this method, one has a guarantee that the data structure is fully tested.
Thanks!
Recursive generators

Assume we test a set as follows:

\[
\text{set()} \rightarrow \{\text{call, sets, from_list, [list(int())]}\}.
\]

\[
\text{prop_union()} \rightarrow \\
\quad \texttt{?FORALL(\{S1, S2\}, \{\text{set()}, \text{set()}\}, \\
\quad \quad \text{equiv(model(sets:union(S1, S2)), \\
\quad \quad \quad \text{model}(S1) ++ \text{model}(S2))}).}
\]

\[
\text{prop_delete()} \rightarrow \\
\quad \texttt{?FORALL(\{S, E\}, \{\text{set()}, \text{int()}\}, \\
\quad \quad \text{equiv(model(sets:delete(E, S)), \\
\quad \quad \quad \text{model}(S) -- [E]))}.
\]
Recursive generators

Assume we test a set as follows:

```erlang
set() -> {call,sets,from_list,[list(int())]}.

prop_union() ->
    ?FORALL({S1,S2},{set(),set()},
        equiv(model(sets:union(S1,S2)),
             model(S1) ++ model(S2))).

prop_delete() ->
    ?FORALL({S,E},{set(),int()},
        equiv(model(sets:delete(E,S)),
             model(S) -- [E])).
```

equiv(S1,S2) ->
    S1 - S2 == []
    andalso S2 -- S1 == []
Recursive generators

Although, most likely, all code for union and delete is covered, an important error may remain.

from_list(L) -> lists:sort(L).

unions(S1,S2) -> S1++S2.
   %% instead of correct lists:sort(S1++S2).

delete(E,[]) -> [];
delete(E,[E|S]) -> S;
delete(E,[I|S]) when I < E -> [I|delete(E,S)];
delete(E,S) -> S.
Recursive generators

Although, most likely, all code for union and delete is covered, an important error may remain.

delete(∅,
    union(from_list([1], from_list([∅]))).

A solution is to generate values by combining operations.