Testing Erlang Data Types with Quviq QuickCheck

Thomas Arts
IT University of Gothenburg (Sweden) / Quviq AB
thomas.arts@ituniv.se

Laura M. Castro
MADS Group - University of A Coruña (Spain)
lcastro@udc.es

John Hughes
Chalmers (Sweden) / Quviq AB
john.hughes@chalmers.se

Abstract

When creating software, data types are the basic bricks. Most of the time a programmer will use data types defined in library modules, therefore being tested by many users over many years. But sometimes, the appropriate data type is unavailable in the libraries and has to be constructed from scratch. In this way, new basic bricks are created, and potentially used in many products in the future. It pays off to test such data types thoroughly.

This paper presents a structured methodology to follow when testing data types using Quviq QuickCheck, a tool for random testing against specifications. The validation process will be explained carefully, from the convenience of defining a model for the datatype to be tested, to a strategy for better shrinking of failing test cases, and including the benefits of working with symbolic representations.

The leading example in this paper is a data type implemented for a risk management information system, a commercial product developed in Erlang, that has been used on a daily basis for several years.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms verification

Keywords erlang, quickcheck, datatypes

1. Introduction

Software testing is an as necessary as delicate matter. In particular, designing good test sets is a non-trivial task. Unconscious assumptions about the functionality of the subject under test may leave important scenarios or possibilities out of the testing range/scope. That is why automatic test generation tools that provide random input can be helpful products.

As a successful example, Quviq QuickCheck has proven itself as useful tool for testing Erlang programs (Thomas Arts et al. 2006; Quviq). The first QuickCheck tool was invented by Claessen and Hughes (Koen Claessen and John Hughes 2000). The version of Quviq is implemented in Erlang and adapted to better fit industrial needs.

Erlang'08, September 27, 2008, Victoria, BC, Canada.

In this paper we present a method for validating user-defined Erlang data types using Quviq QuickCheck[1]. As a case study, we use the risk management information system ARMISTICE (ARMISTICE). We selected one data type from this risk management information system as our leading example for this paper: the *decimal* data type, a data type for fractional numbers without writing a numerator and denominator. We found a surprising error in the implementation of this data type, of which symptoms had occurred before, but whose scarce bug reports were not well understood. Errors in a data type may occur in a very unexpected way at the user level. The data type was fixed and thoroughly tested with the described method, increasing the confidence that the present implementation is correct.

2. Motivation

ARMISTICE (Víctor M. Gulías et al. 2006, 2005) is an information system whose business logic has been developed using Erlang/OTP. Using a functional language such as Erlang was a key factor for success not only in implementing a software application to deal with such a complex business domain as insurance management (David Cabrero et al. 2003), but also in reaching an abstraction level at the definition of the system which makes it applicable to different business fields. This innovative risk management system (RMIS) is meant to be a tool for both the expert and the daily non-expert users. An advanced profile will use ARMISTICE to specify a set of resources and their relevant properties of interest, as well as the insurance policies contracted to protect those resources from the consequences of potentially harmful events, whichever these might be for each particular case. On the other hand, the system is of assistance also to the kind of user that, without any expert knowledge regarding coverages and warranties, has to deal with incident reports, accident claims, and file trackings. In this case, ARMISTICE helps by retrieving and isolating just the relevant information for each scenario, according to the provided contextual data, and thus, giving valuable support to actions and decisions.

This software system has been in production for a few years now, after being tested by regular users during the last stages of development. Such testing process is common in software development cycles, but it is hardly ever complete and exhaustive. The fact that an application has been running daily without major problems is just a weak empirical proof of correctness.

In order to provide a greater degree of confidence, and taking advantage of the application's core being implemented in Erlang, we decided to use QuickCheck to automatically generate random tests. As a good starting point, we chose ARMISTICE data types. Data types are the smallest logic element that can be tested in most software applications, and the components on which all other business objects are built upon.

[1] In this paper, QuickCheck refers to Quviq QuickCheck version 1.13.

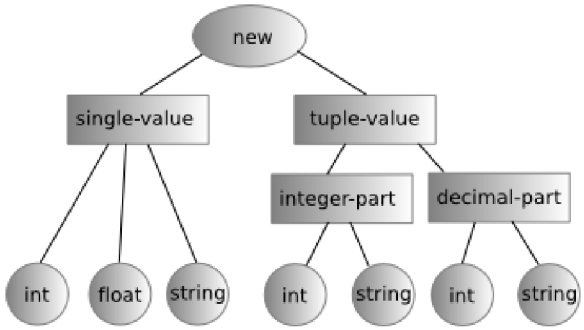

Figure 1. Decimal data type creation options

3. Testing Data Types

There are a number of data types implemented in ARMISTICE and some of them, such as *logico* for booleans and *entero* for integers, are very similar to the basic data types in Erlang. Other data types are built upon these basic types, for example a data type *currency* for representing amounts in different currencies.

In order to be able to have a uniform way of marshalling and unmarshalling values within the system, all ARMISTICE data types have the same structure: a value is represented by a record with the name of the value and wrapped in a tuple with *ok* as the first parameter. In case a data type operation results in an error, the value of the data type is represented by a tuple with first argument *error* and second argument an atom describing the cause of the error. Thus, instead of representing booleans by atoms *true* and *false*, they are represented by $\{ok,\#logico\{value = true\}\}$ and similar for *false*. A division by zero error with two *entero* values will not result in a crash, but in a return value $\{error,division\_by\_zero\}$. This way, even failing operations on the server side are detectable at the client side.

Communication between the ARMISTICE client, written in Java, and the ARMISTICE server, written in Erlang, uses an XML-RPC based protocol. The server first takes care of unmarshalling the messages to obtain Erlang terms, then invoking the corresponding business core service, and finally marshalling the results before replying to the client. For that reason, all data types have constructors to create a value from a string and similarly they all implement a function *to_string* to convert a value to a string. Furthermore, such operations are the basis of all communications with the client, so they are performed very frequently. This means that such conversions need to be fast.

All data types have been tested with the method[2] presented, motivated, and evaluated in this paper.

3.1 Decimal Data Type

We present the method for testing data types with QuickCheck by a leading example in the form of ARMISTICE's decimal data type. A *decimal* is used to represent values with some digits before and some after the decimal separator. It need not have the same range as floats, since it is used to represent sums of money. This data type is defined in a module called `decimal.erl` which exports a creator named `new` in four flavours. As displayed in Fig. 1, input to the function is either a single value or a two-element tuple (first component for the integer part, second for the decimal part). Values are either an integer, a float, or a string representation of one of the two. When providing a single string value, it can contain commas (thousands separator) and/or dot (decimal separator). The decimal separator cannot be used if the two-element tuple notation is used.

[2] The method has been developed by the company and has been fine tuned during this case study.

Other data type constructors provided include mathematical operators: sum, substraction, product, division, negation, absolute value, maximum and minimum; and relational operators, such as 'greater than', and 'less than or equal'.

Now, for testing the decimal data type, one needs a generator for this data type, creating random instances of it, and a property that represents what one likes the data type to fulfil. A relatively naïve approach to generate a *decimal* would be to define the following QuickCheck generator in which the function `new`[3] is applied to an arbitrary integer and an arbitrary positive integer.

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

Note that a decimal can be constructed in many ways, as explained by the different inputs `new` can take, but the temptation followed here is to keep the code for the generator small, since all possible decimals seem to be producible in this way. We will show later that giving in to this temptation results in a missed opportunity to catch an error.

One of the properties that one may like to check is that the sum operator is actually commutative,

```
prop_sum_comm() ->
   ?FORALL({D1, D2}, {decimal(), decimal()},
           sum(D1, D2) == sum(D2, D1)).
```

QuickCheck is used to check this property with successful result, meaning that the specified property holds for tens of thousands of randomly generated test cases. So, `sum` seems indeed commutative. We now are faced with the questions: *which other properties do we add?* and *when do we have sufficiently many properties to cover testing of this data type?*

3.2 Model for data type

We know from the field of mathematics and formal methods (Floyd 1967; Hoare 1972) that creating a model of our data type could help in deciding whether we have created enough properties for the data type. We would like to show that each operation on *decimals* can be simulated by our model operations. Thus, we want an injection $[\![\circ]\!]$ from the decimal data type into our model, such that $\forall D_1, D_2 \in$ decimal

$$\begin{aligned} [\![sum(D_1, D_2)]\!] &\equiv [\![D_1]\!] + [\![D_1]\!] \\ [\![subs(D_1, D_2)]\!] &\equiv [\![D_1]\!] - [\![D_1]\!] \\ [\![mult(D_1, D_2)]\!] &\equiv [\![D_1]\!] * [\![D_1]\!] \\ [\![divs(D_1, D_2)]\!] &\equiv [\![D_1]\!] \,/\, [\![D_1]\!] \\ [\![lt(D_1, D_2)]\!]_l &\equiv [\![D_1]\!] < [\![D_1]\!] \\ &\vdots \end{aligned} \tag{1}$$

In general, we may need to implement a model with all these operations. In this case, though, as our model we can use the standard Erlang implementation of floating point numbers (in itself built upon the C implementation that implements the IEEE 754-1985 standard (IEEE 1985)). We are lucky here, since this choice perfectly fits our *decimals* regardless some rounding issues we will discuss later on (see page 3), but in many situations one can implement a model that is simpler than the data type itself, for example by not caring about efficiency and leaving out optimisations.

We choose a simple injection, viz, mapping *decimals* to Erlang floating point numbers. In fact, this injection function was already present in the code under test:

[3] To enhance reading we simplified the operations in this paper. For instance, the function `new` is a local function that calls `decimal:new` and removes the `ok` tag from its result, and similarly for `sum`, `mult`, etc.

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

Note that in the last equation shown above we use a different model, viz $[\![\circ]\!]_l$ for interpreting the result of the function `lt(D1,D2)`, since that result is a *logico*, not a *decimal*. The model for *logico* simply maps to Erlang booleans and the injection that we use is also already present in the corresponding module.

```
logico_model(Logico) ->
   logico:get_value(Logico).
```

QuickCheck properties to check whether our implementation is equivalent to the Erlang floating point implementation look like:

```
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))).
```

If we now create one such property for each operation defined in the data type, then by checking each of them for a large number of random inputs, we would gain confidence that we tested the data type operations sufficiently.

We start by checking the first property with QuickCheck, which immediately results in a failure.

```
> eqc:quickcheck(decimal_eqc:prop_sum()).
....Failed! After 5 tests.
{[{decimal,1000000000000000}],
 [{decimal,11000000000000000}]}
false
```

After five tests, QuickCheck finds a counterexample against the equivalence between adding two `decimal` values and adding the corresponding two floats. However, the counterexample values reported back by QuickCheck are in their internal representation, which is hard to understand by anyone who is not familiar with the decimal data type implementation. This would be even worse for more complex data types. For a trained QuickCheck user, the values are at least surprising, since one expects rather small integer values and not values with 15 or more zeros. The fact that we actually failed in this test with values 1 and 1.1 is only directly obvious to the developer of the decimal data type.

We do not want to rely on the internal representation of a data type, for one reason because it may be hard for others than the developer of the module to understand, for another reason, because implementations of data types may change and we want tests to depend on them as little as we want implementations to depend on them. Moreover, the internal representation is only the final result of a computation constructing the data structure. We like to know which steps were performed in this construction, since they may reveal information about an observed failure. Therefore, we work with *symbolic* values instead of *real* values. This means that QuickCheck generates a symbolic representation of a *decimal*, which will be evaluated when needed (i.e., during testing). So, our generator is rewritten to:

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

Thus generating a symbolic call to `decimal:new(Tuple)`, i.e., creating a tuple with tag `call`, module, function name, and arguments, instead of actually performing the call.

Of course, we change properties accordingly and introduce in their definition the evaluation of such symbolic values, a standard QuickCheck function.

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

With these modifications, a similar failure is reported back by QuickCheck as:

```
> 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
```

Now it is much easier to see that the problem can be reproduced by using values 0.1 and 0.2.

Conform to the IEEE 754-1985 standard, Erlang float values present an unavoidable rounding error , demonstrated by typing the values in the shell:

```
> (0.1+0.2) == 0.3.
false
> (0.1+0.2) - 0.3.
5.55112e-17
```

In other words, since floats are represented as a list of bits, there is not always an exact representation of each *decimal*. Since our computations are performed on the *decimals* and the conversion to floats is only necessary to compare the results, we are satisfied with an approximate equality. Therefore, we define the equivalence relation `==` with respect to two maximum tolerance levels: an absolute error value (`ABS_ERROR`), which measures how different two floats are; and a relative error value (`REL_ERROR`), which takes into account not only the values themselves, but also their magnitudes (Dawson 2008). Note that we divide by the maximum of the absolute values of the two floats, ensuring that the maximum never is zero (unless they both are zero, in which case the absolute error value is used).

```
-define(ABS_ERROR, 1.0e-16).
-define(REL_ERROR, 1.0e-10).

equiv(F1,F2) ->
   if (abs(F1-F2) < ?ABS_ERROR) -> true;
      (abs(F1) > abs(F2)) ->
         abs( (F1-F2)/F1 ) < ?REL_ERROR;
      (abs(F1) < abs(F2)) ->
         abs( (F1-F2)/F2 ) < ?REL_ERROR
   end.
```

To set the reference error values, we use the knowledge that the decimal data type in ARMISTICE has 16 digits precision (hence, the value of `ABS_ERROR`) and agree to a 99.9999999999% accuracy (hence, the value of `REL_ERROR`). Just a minor change in the

QuickCheck specification is necessary to introduce the new equivalence function.

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

Finally, the property passes thousands of randomly generated test cases.

3.3 Generator to cover data structure

Now we need to introduce one such property for each operation on our abstract data type. Even though it might then seem that we completely tested the decimal data type, we recognise that this is not true. In the first place, and a simple code coverage can demonstrate this, we applied the function `new` to only one of its many types of input (cf. Fig. 1). In the second place, we may have missed to test part of the data structure. Operations on the data structure may actually invalidate an invariant. For example, imagine a data type *set* in which elements are stored in a sorted list, but a set obtained from the union of two sets may invalidate that invariant. Hence, not testing to delete an element from a set obtained form the union of two other sets is a missed opportunity to find an error. Note that code coverage will most likely **not** reveal that we missed testing to delete an element from a set created by the union of two sets, because we have one property testing all code involved in a deletion and one property testing all code involved in computing the union. Together, they cover all code involved in both, which is not the same as covering combinations of creating unions and deleting elements from the union.

Similarly, in our case study, we want to test, for example, multiplication of two *decimals* where each *decimal* itself is obtained by operations that may invalidate an invariant, e.g.

```
   mult(new("12,837.12"),
        sum(new(12), new({13,4}))).
```

To do so, we create a **recursive generator** to obtain arbitrary nesting of *decimals* as arguments of constructors for *decimals*. The depth of the recursion is determined by QuickCheck such that small values are tried first, slowly growing as long as no errors are detected. QuickCheck gives access to the parameter that controls the structural size of the generated test case via the `?SIZED` macro.

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

decimal(0) ->
   {call, decimal, new,
    [oneof([int(),
            real(),
            separator(decimal_string(),digits()),
            {oneof([int(), decimal_string()]),
             oneof([nat(), digits()])}
           ])
    ]};
decimal(Size) ->
   Smaller = decimal(Size div 2),
   oneof([
      decimal(0),
      {call, decimal, sum, [Smaller, Smaller]},
      {call, decimal, mult, [Smaller, Smaller]}
   ]).
```

Note that so far we only consider the `sum` and `mult` operators to create data structures. We will soon see how to add the other operators. Note also that we use `Size div 2` for smaller decimals. This is based on the fact that the smaller decimals have (at most) two subtrees and we want to keep the number of nodes in the tree roughly determined by the size parameter.

Additional code is necessary to specify how the strings in the input may look like. The generator for an arbitrary number of digits is again recursively defined.

```
digits() ->
   ?SIZED(Size, digits(Size)).

digits(0) ->
   [digit()];
digits(Size) ->
   Smaller = digits(Size-1),
   oneof([digits(0), [digit()|Smaller]]).

digit() ->
   choose($0, $9).
```

We can use these digits to generate string representations of decimals, either as a long list of digits, or as such a list grouped by 3 digits at a time and commas inbetween.

```
decimal_string() ->
   signed(oneof([digits(), groups(3)])).

signed(G) ->
   ?LET(S, G, oneof([S, "+"++S, "-"++S]))

separator(G1, G2) ->
   oneof([G1,
          ?LET({S1, S2}, {G1, G2}, S1++"."++S2)]).

%% groups of N digits
groups(N) ->
   ?SIZED(Size, groups(N, Size div N)).

groups(G, 0) ->
   digits(N-1);
groups(G, Size) ->
   Smaller = groups(N, Size-1),
   oneof([
      groups(N, 0),
      ?LET([S1, S2],
           [Smaller, vector(N, digit())],
           S1++","++S2)
   ]).
```

With this recursive generator, symbolic calls can be generated that cover, in principle, the whole data structure. For example, in a QuickCheck sample, the following value was generated:

```
{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,[1.64783]}]}
```

Of course, we need to add the other operators as well, but first we use QuickCheck to check our previously defined property for the sum operator, which successfully passes thousands of tests.

Now we add a similar property for multiplication, which fails after only a few generated tests.

```
6> eqc:quickcheck(decimal_eqc:prop_mult()).
..............Failed! After 16 tests.
{{call,decimal_eqc,sum,
   [{call,decimal_eqc,sum,
      [{call,decimal_eqc,new,["+1"]},
       {call,decimal_eqc,new,[2.36314e+4]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,[-5]},
       {call,decimal_eqc,new,[-9.61993e+5]}]}]},
 {call,decimal_eqc,sum,
   [{call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["74.4"]},
       {call,decimal_eqc,new,[{"-6,179","40"}]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["47"]},
       {call,decimal_eqc,new,["-467,725.079"]}]}]}}
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
```

As we can see from this example, the failing test case contains a fairly large expression. The shrinking procedure reduces the test case quite a lot, but there are still a number of terms in there that a human tester would reduce further. For example the sign could be removed from `{call,decimal,new,["+0"]}`, but even better one could try to remove the whole term. Another reduction could be used for the string `"-000,000.078"`, where six zeros could be reduced to one, at least if the value is important and not the actual structure of the string.

3.4 Improved shrinking of recursive data types

The reason why the previous terms are not shrunk up to the extent we would like they were, lays in the definition of our generators. When we nest `LET` macros, QuickCheck first shrinks on the outermost level and when that is no longer possible, the generator defined inside the `LET` is shrunk[4].

QuickCheck offers a macro `SHRINK` with which one can define one's own shrinking rules. The macro has two arguments, the first being a generator, the second a list of generators. The generators in that list are added as shrinking alternatives to the generator in the first argument. These shrinking alternatives are applied before the built-in shrinking. For example, `SHRINK` could be used to ensure that the `signed` generator always tries to shrink to a representation without a plus or minus symbol.

```
signed(G) ->
   ?LET(S, G,
      ?SHRINK(oneof([S, "+"++S, "-"++S]), [S])).
```

Thus, whenever a test case fails, QuickCheck will first re-test by removing the sign. The original `signed` generator would not do so, since it would only shrink within the alternative chosen by the `oneof` generator.

Note that the order in which we mix `LET` and `SHRINK` above is important. Should we have written

```
signed(G) ->
   ?SHRINK(
      ?LET(S, G,
         oneof([S, "+"++S, "-"++S])), [G])).
```

then shrinking would choose a sequence of digits without a sign, but the sequence would have nothing in common with the original sequence. That is not what we want in this case.

In fact, we often want to know the generated value in order to decide which shrinking steps to add, thus, often having first a `LET` and then a `SHRINK` macro. Therefore, Quviq has added a `LETSHRINK` macro to QuickCheck which combines the two earlier mentioned macros. As arguments we provide a list of bindings and a list of generators, resulting in adding those bindings as shrinking alternatives. In that way, the above `signed` generator can be simplified to:

```
signed(G) ->
   ?LETSHRINK([S], [G],
              oneof([S, "+"++S, "-"++S])).
```

where `S` is automatically added as a shrinking alternative.

The `?LETSHRINK` macro can also be used to reduce the number of groups present in the failing test case `"-467,725.079"`, for example, which shrinks to `"-000,000.078"` instead of also removing the group of three zeros. We define that each smaller generator in the recursively defined generator is automatically added to the shrinking alternatives[5].

```
groups(G, Size) ->
   Smaller = groups(N, Size-1),
   oneof([
      groups(N, 0),
      ?LETSHRINK([S1, S2],
                 [Smaller, vector(N, digit())],
                 S1++","++S2)
   ]).
```

With this shrinking rule added the string `"-000,000.078"` will shrink to `"-000.078"`. Of course, we would still like to shrink that further, which means that we need to look at the generator for digits.

```
digits(Size) ->
   Smaller = digits(Size-1),
   oneof([digits(0),
          ?LETSHRINK([Digits], [Smaller],
                     [digit()|Digits])
   ]).
```

Which will now enable to shrink to `"-0.078"` as we can see from re-checking the property on the same example with all above shrinking alternatives added.

[4] The `choose` generator built-in shrinking strategy always defaults to the first element in the argument list, while shrinking of `int`, `real` or `nat` tends towards zero.

[5] Strictly speaking, a group should start with a number of digits less than or equal to three. However, we add the possibility to have a group with exactly three digits. We add this as our second shrinking alternative, whereas the first is a flexible number of digits.

```
Shrinking.....................(35 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,["0.4"]},
       {call,decimal_eqc,new,[{"-0","40"}]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["40"]},
       {call,decimal_eqc,new,["-0.078"]}]}]}}
false
```

With these rules for shrinking added, we increase the simplicity of our failing test case. However, the term structure is still unchanged, although we know that adding zero to a number would result in that same number. We can shrink the structure by once more using the `LETSHRINK` macro, now in the definition of the recursive generator for decimals.

```
decimal(Size) ->
   Smaller = decimal(Size div 2),
   oneof([
      decimal(0),
      ?LETSHRINK([D1, D2], [Smaller, Smaller],
                 {call, decimal, sum, [D1, D2]}),
      ?LETSHRINK([D1, D2], [Smaller, Smaller],
                 {call, decimal, mult, [D1, D2]})
   ]).
```

In addition to simplifying the actual term, we also want to be able to see the difference between the value created by the implementation and the value computed in our model. We can do so by adding a `WHENFAIL` macro to the property, which only evaluates its first argument if the second argument is false.

```
prop_mult() ->
   ?FORALL({SD1, SD2}, {decimal(), decimal()},
      begin
         D1 = eval(SD1),
         D2 = eval(SD2),
         Model = model(D1) * model(D2),
         Real = model(mult(D1, D2)),
         ?WHENFAIL(
            io:format("Real ~p\nModel ~p\n",
                      [Real, Model]),
            equiv(Real, Model))
      end).
```

Checking this re-defined property on the same example, shows a difference in outcome by a factor 10. Running QuickCheck a few additional times always shows the same factor 10 difference.

```
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
```

The difference was rather quickly identified as an error in the `decimal` module implementation: the carrier was incorrectly propagated. The problem arose when values were rounded to ignore the least significant digits, which are not to be stored, as previously said. In such cases, a rounding operation was considered for the last decimal digit to be stored, but no carrier was taken into account to be propagated to the left. Instead of that, if the last decimal was to be modified (rounded) and this digit turned out to be a 9, then the 9 was erroneously replaced by a 10. For instance, when rounding a large number like $123.45678\mathbf{9}987654$ on six significant digits, we should obtain $123.4567\mathbf{90}$. But, instead, the erroneous code was replacing it by $123.4567\mathbf{10}$. Since the internal representation of `decimal`s is a sequence of digits with a fixed number of decimals (six in this example), this longer sequence is then interpreted as: $1234.567\mathbf{10}$.

Strangely enough, this rounding error had been in the code for several years without being found a problem. However, after detecting it, the developer could actually relate it to some unsolved, obscure error reports from the customer.

After correcting the code, the properties for addition and multiplication operators passed thousands of generated test cases.

3.5 Well defined generators

We now add the additional operations substraction and division to the generator and create properties for them similar to those for addition and multiplication. This takes hardly any effort after having the framework in place.

```
decimal(Size) ->
   Smaller = decimal(Size-1),
   oneof([
      decimal(0),
      ?LETSHRINK(
         [SD1, SD2], [Smaller, Smaller],
         {call, decimal, sum, [SD1, SD2]}),
      ?LETSHRINK(
         [SD1, SD2], [Smaller, Smaller],
         {call, decimal, mult, [SD1, SD2]}),
      ?LETSHRINK(
         [SD1, SD2], [Smaller, Smaller],
         {call, decimal, subs, [SD1, SD2]}),
      ?LETSHRINK(
         [SD1, SD2], [Smaller, Smaller],
         {call, decimal, divs, [SD1, SD2]})
   ]).
```

We check the properties with QuickCheck and now the property for the addition fails again! It does because it crashes in the evaluation of a generated value. In other words, we generated a symbolic value that does not correspond to a real value.

```
> 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
```

Indeed, shrinking helps us in determining the problem: we divide by zero when we create our decimal. Evaluating this symbolic value returns a different wrapper than `ok`, viz, `not_ok` and hence the function taking the wrapper away creates a *badmatch* exception.

Actually, we do want to test that division by zero gives us an expected error value. That is, precisely, done in the property for division, since there we test for each *decimal* value that a division in the model results in the same value as a division of *decimal* values. Division by zero in the model should equally well generate an exception.

```
prop_divs() ->
   ?FORALL({SD1, SD2}, {decimal(), decimal()},
      begin
         D1 = eval(SD1),
         D2 = eval(SD2),
         case model(D2) == 0.0 of
            true ->
               {'EXIT', _} =
                  (catch model(D1)/model(D2)),
               is_error(divs(D1, D2));
            false ->
               equiv(model(divs(D1, D2)),
                     model(D1) / model(D2))
         end
      end).
```

With such a property, we already check that division by zero is an exception case. What we want is to avoid generating these exception cases, instead we only want to generate well defined symbolic values, meaning that such symbolic value does not raise an exception when evaluated. We use a simple, generally applicable concept for doing so. We define a generator `defined` which evaluates the symbolic value and catches a potential exception; the symbolic value is only defined if no exception occurs. We count on the fact that the majority of the symbolic values will not raise an exception when we evaluate them and we introduce a generator `well_defined` to keep generating values until we find a defined one.

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

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

We use these generators in our QuickCheck specification for *decimals* by replacing the generator for decimal by

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

One may consider this to be cheating. After all, we could have an error in our code that makes an operation crash, even though we do not want that to happen. If we filter the generators and never produce such faulty value, how well do we test? Here it is important to remember that we check **each** operation in a property. Thus, if there is any value for which the addition fails, then that value is checked in the property for addition. The only case we never check is whether the `new` operation crashes on a specific input. Therefore, we add one additional property, checking that generating base values always succeeds. In addition we also check that applying the model function to an obtained *decimal* does not raise an exception.

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

Finally we succeed in passing hundred thousand tests for each property in our specification. Therewith we have thoroughly tested the *decimal* data type.

A last property remained untested, which is the actual motivation for introducing the *decimal* data type in the software. The data type is used to ease marshalling from and to strings to enable communication via a web-service-like interface. We want to verify that, for each and every `decimal` data structure we generate in any possible way, converting it to a string and then performing the reverse operation (call `decimal:new` with that same string) is idempotent.

```
prop_decimal_string() ->
   ?FORALL(SD, decimal(),
      begin
         D = eval(SD),
         decimal:new(decimal:to_string(D)) == D
      end).
```

With this last property also successfully checked, we conclude the testing of the *decimal* implementation.

4. Conclusions

The contribution of this paper is the introduction of a methodology to test data types, consisting in four steps:

1. Define a model for the data type: the goal is checking whether the data type implementation is equivalent to the model, thus holding equivalent properties.
2. Create as many model-equivalence checking properties as operations in the data type. Work with symbolic values instead of real values, to keep independent of internal representation of the data type.
3. Write data type generators. Make sure they cover all the data structure, i.e., define recursive generators that include all operations producing values of the data type as a result. Mind that generated values are well-defined, placing exception cases testing just at relevant properties.
4. If needed, define your own shrinking preferences to help reaching the least significant counterexample.

With this methodology, QuickCheck can be used in a more structured way, giving more confidence in the result of all QuickCheck tests. The methodology has been applied to Erlang data types, but the approach should as well be applicable to data types defined in another language, such as C and Java, provided we can interface Erlang with such language. We plan on investigating that as part of our continued research.

Even though one may expect data types to be rather simple pieces of software, we have shown that failures can be detected in software considered very stable and used for several years. We have applied the developed methodology to other data type implementations as well, for example, the *entero* and *logico* data types in the same risk management information system, and the *queue* data type in the Erlang standard library. This showed that the method is generally applicable and basically a recipe to follow.

Using either QuickCheck or another test case generation tool is not a trivial step, though. A too naïve approach can convice

ourselves we have tested enough when we are actually missing a lot or even not really proving anything relevant. In this paper we have explained, step by step, an exhaustive procedure that can be easily followed. By describing the process thoroughly we have tried to justify each stage of the process and illustrate possible problems (wrong approaches, obscurity of dealing with real values and their internals, failure to test all possibilities, unsatisfactory shrinking) and how to overcome them (model definition, symbolic values, recursive generators, self-defined shrinking rules).

As said before, we plan to extend this methodology to non-Erlang data types in the near future, taking advantage of already existing intercommunication alternatives between Erlang and other technologies such as C or Java. We are also working on a broader methodology that will consider how to check not just a relatively simple issue as data types, but a whole application business logic. We aim to do so focusing on layered client/server applications, and independently from the user interface and persistent storage.

Acknowledgments

We thank Víctor M. Gulías from the University of A Coruña for his support and for creating the possibility for this collaboration. This research was partly sponsored by two grants: FARMHANDS (MEyC TIN2005-08986 and XUGA PGIDIT06PXIC105164PN) and EU FP7 Collaborative project *ProTest*, grant number 215868.

References

ARMISTICE. Armistice. http://www.madsgroup.org/armistice/, 2002.

David Cabrero, Carlos Abalde, Carlos Varela, and Laura M. Castro. Armistice: An experience developing management software with erlang. In *Principles, Logics, and Implementations of High-Level Programming Languages*, Uppsala, Sweden, August 2003.

Bruce Dawson. Comparing floating point numbers. http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm, 2008.

R.W. Floyd. Assigning meaning to programs. In *Proc. of Symposia in Appl. Math.* American Mathematical Society, 1967.

C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

IEEE. Standard for binary floating-point arithmetic. http://grouper.ieee.org/groups/754/, 1985.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

Quviq. Quviq. http://www.quviq.com, 2008.

Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, New York, NY, USA, 2006. ACM Press.

Víctor M. Gulías, Carlos Abalde, Laura M. Castro, and Carlos Varela. A new risk management approach deployed over a client/server distributed functional architecture. In *18th International Conference on Systems Engineering*, pages 370–375, University of Nevada, Las Vegas (USA), August 2005. IEEE Computer Society. http://www.icseng.info.

Víctor M. Gulías, Carlos Abalde, Laura M. Castro, and Carlos Varela. Formalisation of a functional risk management system. In *8th International Conference on Enterprise Information Systems*, pages 516–519, Paphos (Cyprus), May 2006. INSTICC Press. http://www.iceis.org.