CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Testing Approach for Haskell SQLite Bindings

Master's Thesis in Computer Science

JIMMY SVENSSON
VICTOR EVERTSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

MASTER THESIS 2018:NN

Testing Approach for Haskell SQLite Bindings

Master's thesis in Computer Science

JIMMY SVENSSON
VICTOR EVERTSSON

UNIVERSITY OF
GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2018

Testing Approach for Haskell SQLite Bindings
Jimmy Svensson
Victor Evertsson

Chalmers Supervisor: Michał Palka, Department of Computer Science and Engineering
Examiner: Alejandro Russo, Department of Computer Science and Engineering

Master's Thesis 2018:NN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2018

Testing Approach for Haskell SQLite Bindings
Jimmy Svensson
Victor Evertsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

To make an SQLite database as accessible as possible, it is important that it can be used from several programming languages such as Haskell. Haskell is a purely functional programming language that supports higher-order functions and has a rich type system. In order to use an existing database in Haskell, a binding could be created between the database and the programming language. The binding needs to handle many different type conversions in order to convert the types between Haskell and the database properly.

This thesis presents one testing approach on how to find potential bugs in SQLite bindings to the programming language Haskell. The approach uses a predefined sequence of database queries that is constructed in such a way that the correct answer easily can be decided. The approach makes use of two custom made test cases that are created in Haskell. The properties are used together with the testing tool QuickCheck to perform automatic random tests.

After the tests are performed, the found bugs are presented. The method we have developed gives an easy way to find bugs in bindings between Haskell programs and SQLite databases.

Keywords: Binding, Database, Haskell, Properties, Test and QuickCheck.

Acknowledgements

We would like to thank our supervisor Michał Palka for good guidance, good feedback on the report and that he always has been available during the project.

We also want to thank our examiner Alejandro Russo for good feedback and ensuring the quality of the report.

Contribution

The authors of this report have contributed equally to the work and the writing.

Jimmy Svensson & Victor Evertsson, Gothenburg, 2018

Contents

List of Figures

List of tables

Definitions

FFI	FFI (Foreign Function Interface) is a programming mechanism that makes it possible to use services from another programming language.
Differential testing	A testing technique used in random testing. The technique uses two or more systems that perform the same task and compare them with each other. Randomly generated test cases are used as inputs to the systems and the outputs are compared for differences, infinite loops or crashes. [1]
Tuple	A data type in Haskell that contains values separated with a comma inside a parenthesis.

1 Introduction

Today, there are a large number of libraries that is implemented in one programming language and needs to be accessed in another. There can be various reasons why a specific programming language can be beneficial compared to others. The programmer may lack experience in other programming languages or simply prefer one programming language over another. Therefore, the implemented libraries should preferably be available in as many programming languages as possible. The obvious solution would be to recreate the libraries in every programming language. However, this would be very time-consuming and unnecessary since the programming code for executing the library already exists. One way to reuse a library in another programming language is by using a binding.

Assume that we prefer to use a library in one programming language, for instance Haskell but the library is implemented in another language, for instance C. The library can then be used in Haskell if we use a binding between the two languages (shown in Figure 1.1). A binding is a connection between two programming languages that runs the code for both languages as one process. However, code written in two different languages can communicate without a binding, for instance, they can agree to exchange information of a certain format that is supported in both languages. Assume that we use a binding and we would like to use the sinus function from the C library in Haskell. To use the function, we start by sending a *Double* value that we would like to use in the sinus function, let us say the value 90.0, to the binding. The binding, which must handle any type conversions between the languages, will convert the type of the *Double* to C*Double*, which the binding uses to represent C's *double* type. The binding then sends the value with the converted type to the library's function. The function executes the sinus calculation that gives the result 1.0 (sinus 90.0 = 1.0). The result is sent back to the binding that converts the value back to Haskell's type. Finally, it sends the value back to Haskell.

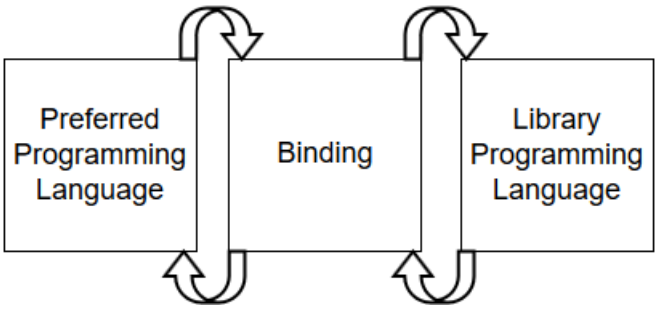

Figure 1.1: The paths to and from the binding

Databases are often important when building user applications. Many clients for databases such as SQLite and MySQL are implemented in C. Without bindings, it will force developers to use C as their programming language. C is a low-level programming language that can make that the programmer focuses more on the details instead of the algorithms for their application. However, it is important that the binding's type conversions are correct and therefore testing is necessary. This thesis aims to test the correctness of database bindings.

A common problem that can appear in a database binding is incorrect type conversions. An example of a bug that was found in the SQLite binding: *sqlite* [2] is shown in Figure 1.2, where the SQLite code is executed by Haskell by sending a request to insert the Haskell *String* "ø" (a letter of the Latin alphabet) in a database table. Then, it sends another request to receive the value from the table but the output of the request is the Haskell *String* "Ã¸".

In the example, Haskell uses the type *String* and the database uses the C programming language type *char* * (a *char* pointer to the first char in the *String*, stored in memory) using the UTF-8 encoding. The binding converts the Haskell *String* to the C *char* * and the other way around. The code of the binding was examined and it showed that the database returned a correct value in the UTF-8 format but a bug occurred when the value was received from the binding and it means that the value of the type *char* * using the UTF-8 encoding was incorrectly converted to a value of the Haskell type *String*.

```
insert into table (column1) values ('ø');
select * from table;

Output
"Ã¸"
```

Figure 1.2: SQL-code for the database requests and its output

The types in different programming languages often differ and it makes it nontrivial to write the code for all the type conversions in a binding. Literature confirms that it is a difficult task to write a binding [3].

The chosen programming language in this thesis is Haskell. Most Haskell bindings are written for C since Haskell's FFI (a tool that can be used in a binding to make it easier for Haskell to communicate with other languages) has support for calling C functions from Haskell. This means that there are many bindings to databases (since many databases are implemented in C). However, bindings for other programming languages can be used but they have less support by default for calling functions in their programming language from Haskell.

There are many testing techniques that are possible to use when testing bindings. Two of the alternatives are Unit testing [4] and QuickCheck's random testing [5]. The difference between these test techniques is that Unit testing uses static tests where the programmer specifies each input while QuickCheck runs randomly generated inputs for every test. A large number of values should be tested when a database binding is used, in order to verify that the binding converts every value correctly. Therefore, we preferred to use QuickCheck's random testing as the testing technique since it can generate random values of Haskell types. Most databases take *Strings* as input and they can be generated with QuickCheck. The results from database bindings are Haskell values and they can be verified for correctness with QuickCheck.

This thesis tests the seven most downloaded SQLite bindings from Hackage [6] (last checked in September 2016). The testing is performed with two properties (shown in section 5.2) and the testing tool QuickCheck. We found four bugs in three different bindings and these are presented in chapter 6.

1.1 Problem Description

Many well-known databases such as SQLite and MySQL are carefully tested in order to guarantee correctness [7], [8]. However, bindings from databases to other programming languages have not been targeted in the same way, even though they must be correct in order to receive the correct values from a database. A binding is simpler than a database since it only handles type conversions while the database handles the core functionalities. However, the binding has to handle a number of type conversions that is nontrivial in many cases.

There are several challenges when writing a binding between a library in one programming language and another programming language. To make a binding work, the programmer needs to understand the type system, calling conventions, memory allocation, data structures and the linking strategy of the language [3]. Errors in each of these areas can lead to incorrect outputs from bindings.

An example of different type systems is shown in Table 1, where Haskell types, Haskell representation of the C types and C types are compared.

Haskell types	Haskell's representation of C types	C types
Int	CInt	int
Int	CUInt	unsigned int
String	CString	char ∗

Table 1: Example of differences in type systems

The C types *int* and *unsigned int* are converted to the Haskell type *Int* since Haskell's native types only use signed *Integers*. To represent a C *int* and *unsigned int*, the Haskell types *CInt* and *CUInt* are used. The C type *char* * points to the first *char* of a char-sequence in the memory while Haskell stores the whole char-sequence with the *String* type. The C *char* * is represented in Haskell with the type *CString*.

There are some problems that can appear when type conversions are performed to the Haskell type *Int*. An *Int* in Haskell is guaranteed to be able to store at least a 30 bits signed *Integer* but it can store a 64 bits signed *Integer* when a 64 bit GHC (Glasgow Haskell Compiler) is used. The data types *CInt* and *CUInt* consist of 32 bit types, which means that they can be larger than an *Int* in Haskell. This is unsafe when using standard Haskell functions for number conversions like *fromIntegral* that will return completely wrong values if the value is larger than the target type. It is also possible to use Haskell's *Integer* type that has no defined max size limit. However, if conversion to Haskell's *Int* should be made then it is important to check the size of *CInt* or *CUInt* to make sure that it will fit in the Haskell *Int*. The types *String* and *CString* have no defined max limit.

A particular challenge between Haskell and for instance C is that the individual types are very different and therefore it is difficult to make the type conversions correct in the binding. To verify the correctness of the type conversions, it is important to test many values of the supported types since some values might be incorrectly converted. Our aim is to test database bindings with random testing, especially to verify the correctness of the type conversions but also to verify other problems that result in incorrect outputs, for example memory allocation.

The data structures in the languages can also differ, as shown in Table 1, *Strings* are handled by pointers to memory locations by the user in C while Haskell handles the memory management without involving the user. The binding needs to ensure that the data structure is converted to an equivalent data structure in the other programming language.

An example of a memory problem that can appear is when the type from Haskell should be represented in C since, for some types, Haskell and C have a different amount of memory allocated. The binding needs to assure that the memory allocation is handled correctly and that it does not allocate insufficient memory (memory out of bounds).

1.2 Contribution

- We develop a testing approach for testing database bindings and we implement it in Haskell using QuickCheck. The approach uses a predefined sequence of queries (*create*, *insert* and *select*) that have random names and values. The sequences are constructed in such a way that the expected result of running them is easily determined. Our purpose for this approach is to give the user the possibility to run multiple tests on one binding in isolation.

- We have found four bugs in the tested bindings. The bugs are in the bindings *HDBC-sqlite3*, *simplest-sqlite* and *sqlite*. The bugs involve three different types, *ByteString*, *Double* and *String* and all bugs have been reported to the developers of the bindings. One bug has already been fixed. The bug affected unusual symbols (such as "ø"), inserted as c*hars* in an SQLite database table since they were retrieved as different *chars* in Haskell. The problem was that the *char* encoding in the database was not correctly converted to the *char* encoding that Haskell uses.

1.3 Limitations

There are a few limitations of the testing approach. One limitation is that the approach only uses the queries, *create*, *insert*, *select* and *drop*/*delete*. These queries are executed in a predefined sequence. If there are type conversion errors that occur with random sequences of queries, then these errors will not be found. Similarly, errors that appear with other kinds of queries will not be found. Another limitation is that the testing approach does not cover all functions of the bindings since they have similar functions or that the functions are not useful in our properties.

2 Haskell FFI Bindings

Haskell code can call functions from foreign libraries written in other programming languages by using bindings. To create a binding for Haskell, every function in a library needs to be treated separately and be called in the right way by Haskell. For this task, an FFI (Foreign Function Interface) can be used. FFI is a programming mechanism that makes it possible to use services from another programming language. The FFI has default enabled the possibility to call any programming code that uses the C calling convention from Haskell.

To use the FFI, a compiler pragma needs to be included at the top of a Haskell file (shown in Figure 2.1).

```
{ −# LANGUAGE ForeignFunctionInterface #−}
```

Figure 2.1: Compiler pragma for FFI

The types for the input and output of C functions must have a C type representation in Haskell to be able to distinguish C types and Haskell types. The C type representation must be converted to native Haskell types to complete a binding. The supporting definitions for calling C functions from Haskell are contained in the Foreign package. The package contains C functions such as *CString*, *CInt*, *malloc* (memory allocation on the heap) and *alloca* (allocates memory within the current function's stack frame). The names of the C types representation in Haskell are indicated with a C in front of the type, for instance *CDouble*. The next step is to import the C function(s).

When the import is made, the programmer must specify the name of the C function with quotes and its C type represented in Haskell. For instance, Figure 2.2 demonstrates an import of the *sin* function from C. The *sin* function calculates sine of a given angle.

```
foreign import ccall "sin" c_sin :: CDouble → CDouble
```

Figure 2.2: An import of the *sin* function from C

This defines a function with the name *c_sin* that uses the C calling convention, indicated by the *ccall* keyword in the import. The function takes a *CDouble* and returns a *CDouble*. Whenever the *c_sin* function is called with a *CDouble*, then Haskell will at runtime pass the C*Double* to the *sin* function in C, execute its calculations and finally return a C*Double* to Haskell. The function (*c_sin*) can have any name the programmer chooses. However, the normal naming convention when using FFI is to add "c_" in front of the name, of the C function when native Haskell types are not used. This is a good practice since it can avoid using the unconverted version by mistake.

The last step is to convert the *c_sin* function's input and output to native Haskell values. This step is where errors easily can occur if the programming code is not written correctly. The Haskell compiler cannot type-check the arguments to the C function and will therefore accept incorrect types. This can cause errors like C compiler warnings, crashes or even silent errors that might appear later in the execution of a program. A binding function that takes a Haskell *Double* value as input, convert it to a *CDouble*, use it as input to the *c_sin* function (that outputs a *CDouble*) and converts the output back to a Haskell *Double* value, is shown in Figure 2.3.

```
sinc :: Double → Double
sinc doub = realToFrac (c_sin (realToFrac doub))
```

Figure 2.3: The binding function of the c_sin function

A *Double* is a double precision floating-point type and the function *realToFrac* can be used for conversion between floating-point values (the function *fromIntegral* is used for *Integers*). In the example, the function *sinc* uses the function *realToFrac* to convert its input (*doub*) from *Double* to *CDouble*. The function *c_sin* is then executed with the *CDouble* input and calculates the sinus value and the result of the function will be *CDouble* that *realToFrac* converts back to *Double*. The *sin* function in C can now be used as any other Haskell function by using *sinc*.

2.1 FFI Tool

To make it easier to create bindings, the tool Hsc2hs can be used. Hsc2hs makes it easier to bind Haskell to C code since some parts are automatically done. The tool reads an hsc file containing the code for the binding, C headers, C types and processing rules and outputs a Haskell file containing processed information from the C headers. The information that is added can for instance be field offsets in a C *struct*. The hsc file is used to connect Haskell to C in our approach but it supports more programming languages [3].

2.2 Database Bindings

Databases can use bindings to connect to one or more programming languages. In order to use the database correctly from the other programming language, it is necessary that the bindings make all type conversions properly. Incorrect type conversions can result in incorrect database behaviour and therefore, it is important to test the bindings.

SQLite [9] is a serverless database that is very popular and it is easy to configure. For instance, SQLite is often used to store data on mobile devices. The difference between SQLite and most other databases is that SQLite is normally used locally to read and write directly to files on the disk. The file management makes it easier to use in testing since setting up and removing the contents of the database can be done by creating and deleting a file instead of managing a database. It is also faster than many other databases when it is used with small amount of data.

A simplified function that executes queries using a Haskell database binding is shown in Figure 2.4. Assume there is a C function *runCommand* that executes queries and is implemented in C. There is also a Haskell file that is able to use the function *runCommand* through the binding. The connection between the Haskell file and the C file is made with an hsc file (the connection is shown in Figure 2.5). The file contains two functions. The first is the *c_runCommand* function and it is an imported version of *runCommand* in the C file with corresponding C types as inputs and outputs. The second is the *runComHaskell* function that the Haskell file uses.

The function is created in the same way as in the *sin* example but with *String* instead of *Doubles*. To execute the query, a Haskell *String* with a query is sent to the *runComHaskell* function (shown in Figure 2.4). The query must be converted to a *CString* in order to be used by the C function and then call the *c_runCommand* with *CString* to execute the query. The type conversion is made with *withCString* and it also uses the *c_runCommand* function to execute the query (which is a *CString*). When the query is executed it will return the result as *IO CString*. The *CString* is extracted from the *IO* and stored to the variable *ans*. The *peekCString* function takes the *CString ans* and converts it to *IO String*. The *runComHaskell* function uses only native Haskell types and it is therefore safe to use in a Haskell file.

The conversions of *Strings* performed by the functions *peekCString* and *withCString* are more complicated than with *Doubles* since the functions must use the *char* * to retrieve each *char* in the char-sequence and they have to be converted and allocated in memory, compared to *Doubles* that are passed and returned directly.

```
foreign import ccall "runCommand"
   c_runCommand :: CString → IO CString

runComHaskell :: String → IO String
runComHaskell query = do
   ans ← withCString query c_runCommand
   peekCString ans
```

Figure 2.4: The functions in the HSC file

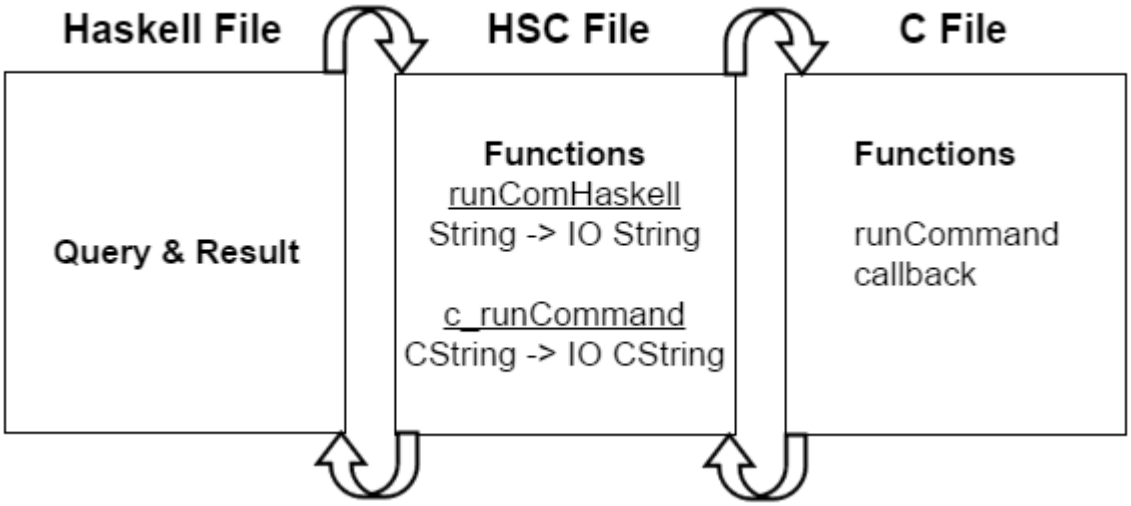

Figure 2.5: The connections between the files

3 Testing Background

The testing techniques that will be explained in this chapter are black box testing, white box testing and some different test design techniques. QuickCheck, which is a black box testing tool used in this thesis, will be also explained.

3.1 Black box testing

Black box testing [10] is a technique that takes some input to a function and tests the outputs from it without knowing anything about the internal code of the function. For example, consider a black box function that takes a degree in Celsius as input and returns the corresponding Fahrenheit degree. How the black box function makes the calculations to convert the Celsius to Fahrenheit is unknown to the user. Figure 3.1 demonstrates the function, where 10° Celsius is used as input and the function returns 50° Fahrenheit. To the user it may seem like the function multiply the input by 5, but it multiplies the input by 1.8 and adds 32. Testing functions without looking at the code is a good approach to find unexpected bugs, especially for complex functions.

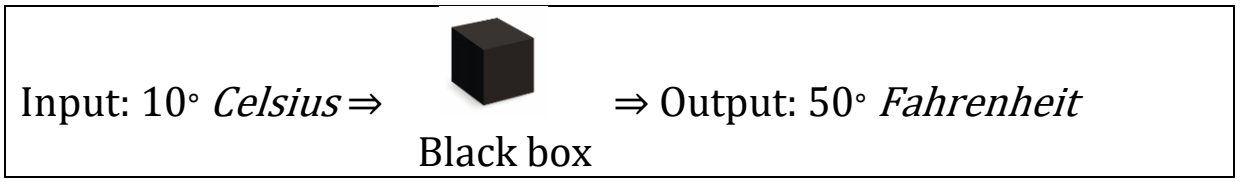

Figure 3.1: A black box example

Black box testing can be performed with different design techniques. One of them is boundary value analysis [10]. In boundary value analysis, the tests are focusing on the minimum and maximum inputs and outputs. This is useful since programmers often make mistakes in corner cases when using comparison operators (i.e. greater than, less than, greater or equal and less or equal). An example of boundary value analysis for inputs is shown in Figure 3.2, where a black box function takes the number of months you would like to pay for a service as input. For this example, six tests should be performed on the black box function that only accepts values from 1 to 12. The first two tests take the input values 1 and 12 that test values exactly as the input boundaries. The third and fourth tests take the input values 0 and 11 that test values just below the boundaries while the last two tests use values just above the boundaries.

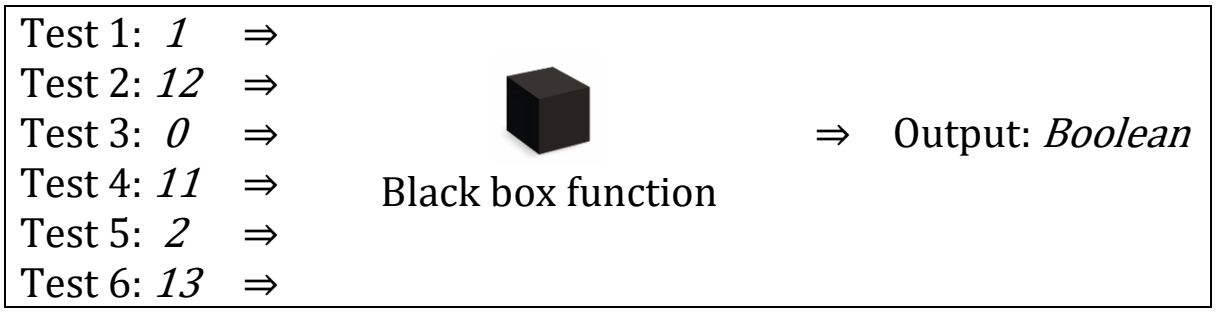

Figure 3.2: A boundary value analysis example

Another black box design technique is decision table testing [10] where the goal is to test specific predicates that should hold or not hold for the program. The technique uses a sequence of predicates that test cases fulfill or not. A test case defines a number of *Boolean* values for the predicate and that will give a certain result. The test cases should also have a unique combination of the possible values (True/False) in order to test new functionality.

An example of a decision table is shown in Table 2, with three test cases, two predicates and one result. If both predicate have the value True, then the result is accomplished (Player A stays in game) and has the value True. It is also accomplished if predicate 1 (Player A lands on player B's property) is False, independently on the second predicate. If the first predicate is True while the second predicate is False, then the result is not accomplished and has the value False (it means that player A is not in the game anymore).

	Test case 1	**Test case 2**	**Test case 3**
Predicate 1: Player A lands on player B's property	True	False	True
Predicate 2: Player A has enough money to pay rent	True	--	False
Result: Player A stays in game	True	True	False

Table 2: An example of a decision table

Decision tables could be useful in testing since it provides a systematic way of finding bugs. In our testing it could be used by defining a number of predicates such as *insert UTF-8 Strings*. The expected result would be that the binding can handle UTF-8 inserts.

Random testing is a black box design technique that generates random inputs to a program that should be tested. The output of the program must be possible to verify against a specification to be able to decide if the output is correct. If the service payment function (the example in Figure 3.2) would be tested with random testing, it would generate a random number as input to the function and test if the number is within the accepted value range and compare it with the output from the service payment function.

Random testing covers many unexpected inputs that can cause bugs, which is very efficient when testing bindings since they usually have many inputs and it is not possible to test every input within a reasonable time.

There are similarities between the three design techniques. All three techniques test input values on a condition. However, a difference is that a decision table uses all of the possible input values to get the action's result while boundary value analysis and random testing does not have to test all possible values since they analyses corner cases and uses random inputs. To test bindings with a decision table, all possible errors have to be defined as test cases and boundary value analysis has to define what the boundaries are in each binding, which is not trivial. However, random testing does not require to define every possible error that can occur in bindings, only that the input generations is appropriate and it ensures that a large amount of inputs are tested.

3.2 White box testing

White box testing [11] is a technique where the tester uses the knowledge of the internal code from a program to maximize the code coverage. Code coverage is used to measure the amount of code that is tested by a test suite. The test suite consists of test cases where the inputs often are chosen by the tester to cover as many reachable paths as possible in the program. This is important since untested code has a higher probability of containing bugs. To measure the code coverage, one or several coverage criteria may be used. The three most used coverage criteria are statement coverage, branch coverage and path coverage. Statement coverage means that every line of the programming code is executed at least once. An example demonstrating statement coverage is shown in Figure 3.3, where the value 20 is given as input and therefore the if condition is fulfilled so it performs every line of the code.

```
INPUT a = 20

if (a > 10)
  Print "Statement coverage is fulfilled"
```

Figure 3.3: Pseudocode example of statement coverage

The branch coverage criterion means that if there are different paths in the program, for example an if statements, it checks that each if statement has been tested with both the *Boolean* value True and False, at least once. The example in Figure 3.3 has 50% branch coverage since it does not test the if statement with the value False.

The path coverage criterion means that every possible code execution path of the given program is tested. For example, in Figure 3.4, it means that every possible combination of clauses of the three if-else statements (for example one if statement True and the others False) will be executed. The total number of tests performed in the example will then be 8.

```
INPUT Boolean a, b, c

if a then Print "The first if statement is true"
else      Print "The first if statement is false"

if b then Print "The second if statement is true"
else      Print "The second if statement is false"

if c then Print "The third if statement is true"
else      Print "The third if statement is false"
```

Figure 3.4: Pseudocode example of branch coverage

Sometimes it is impossible to get 100% path coverage, for instance, two if-else statements with the same condition will not be able to take the path True, False or False, True (shown in Figure 3.5). The maximum path coverage is therefore bounded by a certain percentage, for instance, the path coverage in the two if-else statements are bounded to 50%. There are no defined stopping rules for path coverage (i.e. when the maximum path coverage is reached) of a program but it is possible to avoid writing if-else statements that uses the same condition, which makes it possible to reach 100%.

```
INPUT Boolean a

if a then Print "The first if statement is true"
else      Print "The first if statement is false"

if a then Print "The second if statement is true"
else      Print "The second if statement is false"
```

Figure 3.5: Pseudocode example of maximum branch coverage

For a given program, it is often theoretically possible to reach 100% statement- and branch coverage but in practice it is difficult since many programs have code that handles error conditions or code that never should be executed.

3.3 QuickCheck

QuickCheck [12] is a testing tool that generates and runs automatic random tests based on a predicate defined in a property. The property specifies the data types that are used in the testing of a function, what the expected output of the function should be and possible preconditions that the input must fulfill when it performs testing. When QuickCheck is used on a property, it will run a predefined number of tests and if a test fails during the run, it will show a counterexample (the input to the property that made the test fail). Otherwise it will print that all tests have passed.

Unlike unit testing, that uses static tests, QuickCheck generates random inputs for each test. This means that QuickCheck will be able to test new inputs in each test while unit testing tests the same values each time.

3.3.1 Property-based Testing

Property-based testing uses properties to define how the program under test should behave. For instance, a simple property can be created to, test the reverse function in Haskell (shown in Figure 3.6).

```
prop_reverse :: String → Bool
prop_reverse string = reverse (reverse string) == string
```

Figure 3.6: A property that tests the reverse function

The property tests that applying the reverse function twice yields the same *String* as not applying the reverse function at all. The property itself is a function and it is tested with QuickCheck that inputs randomly generated *Strings* to the property and verifies it. The result of the tests is shown in Figure 3.7.

```
* Main > quickCheck prop_reverse
+++ OK, passed 100 tests
```

Figure 3.7: A QuickCheck run of the prop_reverse

This indicates that the property holds for at least 100 generated values. It may or may not be necessary to run QuickCheck with more tests to ensure that the property holds for more values.

Another property is defined (shown in Figure 3.8) to demonstrate how QuickCheck can give a counterexample for a property that does not hold.

```
prop_digit :: String → Property
prop_digit string = and [isNumber char | char ← string] ==>
                    and $ map isDigit string
```

Figure 3.8: A property that tests the function *isDigit* on the numbers (represented as *Strings*)

The property is a function that gets a *String* as argument. The *String* is used in a precondition (the second line in Figure 3.8) that test each *Char* of the *String* with the function *isNumber*. If the function returns false for any *Char*, then the precondition will evaluate to false and the *String* will be discarded. If this happens, a new *String* will be generated. This means that the property only tests *Strings* that represent numbers. It checks that every *Char* (which represents a number) in the *String* is an ASCII digit in the range 0-9 (with the function *isDigit*). The property is run with QuickCheck and the result in shown in Figure 3.9.

```
* Main > quickCheck prop_digit
*** Failed! Falsifiable (after 44 tests and 2 shrinks): "\189"
```

Figure 3.9: A QuickCheck run of the *prop_digit* that failed

QuickCheck was able to run 44 tests (and 2 shrinks and it mean that the size of the counterexample has been decreased twice) until it found a counterexample that was the *String* "\189". QuickCheck uses the *Show* class to print its result but since *show* has a limited charset, some UTF-8 symbols are shown with a backslash followed by the UTF-8 number that represents the symbol. By running *putStr* on the *String*, the correct symbol is displayed and in this case, the output gives: "1/2", which indeed is a number. However, the function *isDigit* does not accept such a symbol and it is the reason why the property failed.

3.3.2 Generators

QuickCheck has good support for generating random values of basic Haskell types. It uses the Arbitrary type class shown in Figure 3.10 to define the test data generators for the types.

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a → [a]
```

Figure 3.10: The Arbitrary type class

The Arbitrary type class contains the *arbitrary* function of the type *Gen a*, which is a generator for values of type *a*. The Arbitrary type class also contains the *shrink* function, which given an argument of type *a* will create a list of smaller values of the same type. An Arbitrary instance for the data type *a* is used by QuickCheck to generate random values of this data type. In the instance, it is possible to restrict the values of type *a* to, for example, only contain numbers within a certain range. When QuickCheck tests a property that takes a value of type *a* as input then the Arbitrary instance for this type will be used to generate the values used for testing. For example, the property *prop_reverse* used the Arbitrary instance for *String* to generate *Strings*.

QuickCheck has predefined Arbitrary instances for all the basic Haskell types such as *Integer*, *Float*, *Double*, *String*, *pairs* and lists of any values that can be generated [13]. The types can easily be used in a property.

It is possible to test a generator with the function *sample* to generate some values of an arbitrary generator. An example of this is shown in Figure 3.11 where an arbitrary generator of the type *String* is used. Each line in the figure is a new *String* created by the arbitrary generator.

```
* Main > sample (arbitrary :: Gen String)
""
"W"
"\vuY\SUB"
"R > "
"\EM > h\211"
"X\STX^%"
"HBH\155\DC4d = ro\161t\SOH"
"L}E\182~\160|f\ENQ\SUB3j\ENQ"
"_\DC4V\184\213\SYN\DLEJ\182\172\198 <\NULK"
"\152EX\210\b"
"\"\223\149Rr"
```

Figure 3.11: A sample of the *String* arbitrary generator

The *shrink* function in the Arbitrary instance is used to make a counterexample as small as possible. This is useful in random testing since the counterexample could be very large, which makes debugging more difficult. However, the same error can occur with smaller inputs. The *shrink* function takes the input that made the property fail and shrinks the input to a list of smaller versions of the inputs. QuickCheck uses the *Strings* in the shrunk list and runs the property again with each of them to check if the property still fails. If it does not, then it takes the next *String* in the list otherwise it shrinks the *String* again until it found the smallest *String* it can find in the current list that makes the property fail. The *shrink* function is implemented for the basic Haskell types. However, a custom *shrink* function may have to be implemented when a custom made Arbitrary instance is used in a QuickCheck property (an example of a custom made *shrink* function is shown later in Figure 3.16).

QuickCheck also has plenty of functions for creating custom generators. Some of them are shown here:

- suchThat :: Gen a → (a → Bool) → Gen a
  Given a generator and a predicate function, it generates a value that satisfies the predicate. If the predicate is difficult to fulfill, it could be beneficial to specify a generator that satisfies the predicate.

- listOf1 :: Gen a → Gen [a]
  Given a generator, it generates a non-empty list of a random length.

- vectorOf :: Int → Gen a → Gen [a]
  Given a length and a generator, it generates a list of the specified length of the given generator.

- sized :: (Int → Gen a) → Gen a
  QuickCheck uses a size parameter for the generators to generate small tests at first which then increases with every new test. The generators may use the size parameter differently, for instance, the list generator uses it as the upper bound for the length of the list that is generated. The function *sized* can be used to access the size parameter by specifying a function that given an *Int*, creates the desired generator which size depends on the *Int*. For instance, sized $ \n → choose (0,n) will generate numbers from 0 to the size parameter.

The functions *listOf1* and *vectorOf* are useful when a generator that creates values, should change the values to a random or specific length. *SuchThat* is able to restrict the values that a generator creates. However, caution should be taken when using the function since it will try to generate a value and if it does not fulfill the predicate, it will retry to generate a different value. This can cause problems with the distribution of the generated values and will also take a lot of time to generate values if the predicate is difficult to satisfy. Consider the examples in Figure 3.12 and Figure 3.13, where two different generators for *Strings* that only contains numbers, are specified.

```
∗ Main > sample $ suchThat arbitrary (\x → (and $ map isNumber x) && length x > 0)
"\189"
"\189"
"\8"
"\178"
"2"
"4"
"5"
"0"
"2"
"2"
"\179"
```

Figure 3.12: The First example of a generator that creates a *String* with only numbers

```
∗ Main > sample $ listOf 1 $ suchThat arbitrary isNumber
"4"
"5"
"14"
"9227\190\190"
"233439"
"524660"
"57235321364"
"9"
"4567\178\&374\178\&2470"
"502\190\&6646495669"
"\188\185\&1\188\&84014\188\&7"
```

Figure 3.13: The second example of a generator that creates a *String* with only numbers

In the first example (Figure 3.12), each *String* is randomly generated and tested against the predicate. The predicate in the example is a function, which tests that every *Char* is a number in the *String* and also that the *String* is non-empty. The *sample* function in this example uses a generator that generates *Strings* of very short lengths. In the example, the C*har* '\189' represents the Unicode *Char* '½' in Haskell, which *isNumber* accepts as a number. The reason is that a generated *String* with more *Chars* has a lower probability to consist of only numbers.

In the second example (Figure 3.13), this is solved by specifying a *Char* generator (*arbitrary* in this case) that uses the *suchThat* function and the *isNumber* predicate to remove non-numeric *Chars*. The function *listOf1* is then used to generate a non-empty list of *Chars* (which is the same as a *String*).

In this example, only one C*har* will be generated at a time and tested with the predicate instead of generating a *String* and check all *Chars* of the *String* with the predicate. The generated *String* will have a uniform distribution of each *Strings* length since the generator does not depend on the probability to find a random *String* where each C*har* of the S*tring* is a number, but rather on finding a random C*har* that is a number. The generated *Chars* are then used to create a *String* with random length. The result of the *sample* is more similar to the values generated by the *String* generator from Figure 3.11.

The function *vectorOf* can be used to generate *Strings* of a given length (which represent the numbers). An example of the function is shown in Figure 3.14, where the length is specified as 3.

```
∗ Main > sample $ vectorOf 3 $ suchThat arbitrary isNumber
"243"
"611"
"773"
"627"
"797"
"082"
"86\178"
"561"
"864"
"28\189"
"\742"
```

Figure 3.14: An example with *vectorOf* that generates fixed length of *Strings* with only numbers

The function *sized* is used to access the QuickCheck's size parameter. Let us assume that we would like to only generate *Strings* that have a length that depends on the *sized* parameter. The example in Figure 3.15 shows how the *sized* function can be used and an output of running it. The generated *Strings* (that represent numbers) are small at first but increases after each test.

```
∗ Main > sample $ sized $ \n →
                              sequence [suchThat arbitrary isNumber | _ ← [1..k]]

""
"41"
"9172"
"50405\185"
"2\178\&944696"
"8\178\&9374\185\&930"
"511913\185\&22502"
"8\178\178\&59276260425"
"48\188\&6\189\&5144930886\185"
"\188\&13654637\189\&5034\189\178\&75"
"56944120\188\&120940\189\&3636"
```

Figure 3.15: An example of *sized* that generates *Strings* within a specific length range

Consider the example shown in Figure 3.16, that creates an Arbitrary instance for the *String* generator of numbers.

```
newtype StringOfNumbers = StringOfNumbers String deriving (Show, Eq)

instance Arbitrary StringOfNumbers where
arbitrary = do
  l ← listOf 1 $ suchThat arbitrary isNumber
  return (StringOfNumbers l)
shrink (StringOfNumbers l) = [StringOfNumbers x | x ← shrink l, all isNumber x]
```

Figure 3.16: A new Arbitrary instance of the type *StringOfNumbers*

A new *String* type with the name *StringOfNumbers* is defined to create a new arbitrary instance. The instance generates a non-empty list of *Chars* where each *Char* is tested with *isNumber* to guarantee that it is a number. A custom *shrink* function is also defined, which given the *String l* applies Haskell's standard *shrink* function on *l*. Each *Char* of each *String* in the shrunk list is then tested with *isNumber* and if all *Chars* are accepted, the *String* is added to a list of all accepted *Strings*, which is returned.

QuickCheck uses the *Strings* in the shrunk list and runs the property again with each of them to check if the property still fails. If it does not, then it takes the next *String* in the list otherwise it shrinks the *String* again until it found the smallest *String* it can find that makes the property fail.

3.3.3 Applying Generator in Property

The Arbitrary instance from Figure 3.16 can be used in a property to only generate inputs of the *StringOfNumbers* type. For instance, it is possible to change the property *prop_digit* from Figure 3.8 to use the type *StringOfNumbers* as shown in Figure 3.17. QuickCheck will then use the Arbitrary instance to generate values of the type *StringOfNumbers* instead of generating *Strings* with *Char* that need to be filtered out using a precondition.

```
prop_digit :: StringOfNumbers → Bool
prop_digit(StringOfNumbers n) = and $ map isDigit n
```

Figure 3.17: A property that tests the function *isDigit* on the Number generator

The use of the Arbitrary instance for *StringOfNumbers* also means that the found counterexample will be shrunk to the minimal possible *String* that only consists of numbers. A run of the updated property is shown in Figure 3.18, that finds the counterexample with fewer tests than the previous property (6 tests now and 44 earlier). It does not discard any tests and it shrinks the counterexample 3 times.

```
* Main > quickCheck prop_digit
*** Failed! Falsifiable (after 6 tests and 3 shrinks): StringOfNumbers "\178"
```

Figure 3.18: A QuickCheck run of the updated *prop_digit* that failed

4 SQLite

This chapter explains SQLite's types and a few of its interface functions. The types and the functions are necessary in order to understand the testing of the SQLite bindings and the found bugs.

4.1 Data types in SQLite

Data types in SQLite are dynamically determined while in most other databases they are statically determined. The type of the stored value depends on the value that is inserted in the database rather than the specified column type. For example, a floating point value 1.2 can be stored in an *Integer* column. However, it uses storage classes (the actual type) to store the values and type affinity (preferred column types) to prefer to store the value within a specific storage class, which can affect how the value is stored. Consider an example where a table with two columns has the type affinities *Real* and *Integer*. The *Float* 1.0 is inserted in both columns but the stored values will differ. The stored value will be 1.0 in the *Real* column but 1 in the *Integer* column.

There are five storage classes *Null*, *Integer*, *Real*, *Text*, *Blob* [14]. A *Null* value is simply used when no value specified, *Integer* values are represented using 1, 2, 3, 4, 6 or 8 bytes as a signed *Integer*, a *Real* value is stored as an 8-byte IEEE floating point number, a *Text* value is stored as UTF-8, UTF-16BE or UTF-16LE *Text String*, while a *Blob* value is stored as binary data.

SQLite supports five type affinities: *Text*, *Numeric*, *Integer*, *Real*, *Blob* [14]. *Text* can use *Null*, *Text* or *Blob* types. *Numeric* can use all five classes. *Integer* and *Numeric* works in the same way but have differences when casting/converting values of some affinity type to the *Integer* or *Numeric* affinities types. *Real* works like *Numeric* but forces *Integer* values to be represented as floating points. *Blob* does not prefer any storage class.

Consider an example where a table *t* has the columns names: *te* of affinity *Text*, *n* of affinity *Numeric*, *r* of affinity *Real*, *b* of affinity *Blob*. An insertion is made with the value '100.0' (*String* representation) in all columns. The storage classes for the columns *te* and *r* are unchanged, but *n* have the storage class of *Integer* and *b* have the storage class of *Text* (*Blob* does not prefer any storage class). This means that the value '100.0' is stored in *te* and *b* as a *String* without changing the value, in *n* the value is stored as the *Integer* 100 since 100.0 is equal to the *Integer* 100 and in r the value is stored as the *Real* 100. However, if the inserted value would have been 100.2 (as a floating-point number) all columns except *Text* would have to use the *Real* storage class to be able to store the value 100.2.

4.2 Binding Values to Prepared Statements

Database values can be stored in an SQLite database in two ways: either by representing them directly in the SQL query *String*, or by binding them to a prepared statement. This section explains the C and Haskell functions that are used for binding values to statements and retrieving the values from the database and is relevant for some bugs that were found.

The first of the two Haskell functions are *sqlite_bind_blob*. The function is used to bind a single value to a column of a query that is of the type *ByteString* (*Blob* in the database). The C type declaration of the function and an example of how most SQLite bindings use it in Haskell is shown in Figure 4.1. The example starts by using the function *prepare* to create a prepared statement for inserting a *ByteString* in the database table (*abc*). The value is bound to the first column in the database table since ?1 is in the position that corresponds to A. The number 1 means that the value will be inserted at the first occurrence of the variable ?1. The second line calls the *bind* function with the statement, the column index and the *ByteString*. The third line uses the function *step* to execute the query.

The *bind* function is defined in the bindings and therefore it looks different depending on which binding that is chosen, but the bind function roughly looks like in Figure 4.1. The *bind* function executes the Haskell function *sqlite_bind_blob* that performs the conversion and then calls the C function *sqlite3_bind_blob*. The function *sqlite3_bind_blob* takes 5 parameters as input, the statement, the index column, a pointer (*ptr*) to the *Blob* value, the length of the *Blob* pointer (*lenPtr*, in number of *Bytes*) and a destructor that is used to dispose the *Blob* pointer after it has finished.

Type declaration

```
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*) (void*));
```

Example

```
stmt ← prepare db "insert into abc (A) values (?1)"
bind stmt 1 ((ByteString.pack "ò") :: ByteString)
step stmt

bind = sqlite_bind_blob
sqlite_bind_blob (Stmt sm) intCol bs =
                        sqlite3_bind_blob sm (fromIntegral intCol) ptr lenPtr nullPtr
```

Figure 4.1: The type declaration of *sqlite3_bind_blob* and an example of how to use it

In order to use the fourth parameter in the *sqlite3_bind_blob* function, the third parameter cannot be a *Null* pointer since then the fourth parameter will be ignored. Note also that if the value of the fourth parameter is negative then the behaviour of the function is undefined.

The second function that is important to recognize in order to understand the bugs is *sqlite3_bind_text*. The structure of the function is similar to *sqlite3_bind_blob*, but it is used for the database type *Text* instead. The type declaration of the function is shown in Figure 4.2. The function is used in the same way as *sqlite3_bind_blob* but it takes a *String* value as the third input value.

```
int sqlite3_bind_text(sqlite3_stmt*, int, const char*, int, void(*)(void*));
```

Figure 4.2: The type declaration of *sqlite3_bind_text*

One difference is that if the fourth parameter to *sqlite3_bind_text* is negative then the length of the *String* will be the number of *Bytes* up to the first zero terminator. If the parameter is positive, then it must be the *Byte* offset where the *Null*-terminator would occur (assuming it is *Null*-terminated).

4.3 Result Values from a Query

The different database values can be retrieved with different functions and for each type there are one or more approaches to retrieve the value. Three of the functions, which are important in order to understand three of the found bugs, will be explained in this section.

The first of the three functions is *sqlite3_column_blob*. The function is used to retrieve a single column of the current result row of a query where the column is of the type *ByteString* (*Blob* in the database). The type declaration of the function and an example of how to use it is shown in Figure 4.3. The example starts by creating a statement (*stmt*) that reads everything from the database table (*abc*). The second line uses the function *step* that executes the statement. The statement is used on the third line to call the *column* function together with the value 0 and it indicates that the first column of the result will be retrieved within the *column* function (to get the second column of the table use 1 instead of 0 and so on). The *column* function uses the function *sqlite_column_blob* that performs the conversion and then call the C function *sqlite_column_blob*. The C function will retrieve the first column of the database table where the column is of the type *Blob*. The function *sqlite3_column_blob* is only used when the column type is *Blob*, for other types other column functions are used.

Type declaration

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
```

Example

```
stmt ← prepare db "select * from abc;"
step
column stmt (0 :: Int)

column = sqlite_column_blob
sqlite_column_blob (Stmt sm) intCol = sqlite3_column_blob sm (fromIntegral intCol)
```

Figure 4.3: The type declaration of *sqlite3_column_blob* and an example of how to use it

In order to retrieve a valid result, the statement must point to a valid row and the column index must be valid (not index out of range). If the *Blob* is empty (zero-length) then the function (*sqlite3_column_blob*) will return a *Null* pointer.

The function *sqlite3_column_bytes* can be used to retrieve the length of the stored *Blob* value in bytes (or a C *String*). The returned count of the function does not include the zero terminator at the end of the C *String*. The type declaration of the function *sqlite3_column_bytes* and an example of how to use is shown in Figure 4.4.

Type declaration

```
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
```

Example

```
stmt ← prepare db "select * from abc;"
step
column stmt (0 :: Int)

column = sqlite_column_blob
sqlite_column_blob (Stmt sm) intCol=do sqlite3_column_blob sm (fromIntegral intCol)
                                       sqlite3_column_bytes sm (fromIntegral intCol)
```

Figure 4.4: The type declaration of *sqlite3_column_bytes* and an example of how to use it

When type conversions occur, the function *sqlite3_column_text* may return invalid values. If the stored value is, for instance a *Blob*, then it might be necessary to add a zero-terminator at the end. The safest way to invoke *sqlite3_column_text or sqlite3_column_blob* is by using the function *sqlite3_column_bytes* after the function to ensure that the whole content of the *Blob* is retrieved from the database.

The third and last function that will be explained is *sqlite3_column_double*. The function is similar to the other two functions and can be used as their examples but with *sqlite3_column_double* instead of *sqlite3_column_text* or *sqlite3_column_bytes.* However, the function will return the C *Double* without any conversions. If the function *sqlite3_column_text* is used to retrieve the C *Double*, it will create a textual representation of the number as a C *String* that holds only up to 15 significant digits.

5 Testing SQLite Bindings

Our approach of testing SQLite bindings uses a predefined sequence of queries. The queries have some elements that are chosen at random, for example, table names, values and column names. The sequences of queries are constructed in such a way that the expected result of running them is easily determined.

An example sequence of queries that was generated is shown in Figure 5.1. The first SQL query (line 1) creates a random table name, *tab1* containing random column names (*col1* – c*ol4*), that have the types *Text*, *Integer*, *Real* and *Blob*. The second line inserts one random value into each of the columns, where the type of each value is the same as the column type. For instance, a random *Double* value, *dou* is inserted into c*ol3*. The last query reads the contents of the table. The result from the last query can be compared with the inserted random values, which are known at runtime, to verify that they are unchanged.

```
create table tab1 (col1 Text, col2 Integer, col3 Real, col4 Blob);
insert into tab1 (col1, col2, col3, col4) values (str, int, dou, byteStr);
select * from tab1;
```

Figure 5.1: An example of predefined SQL queries and order of them

5.1 The Tested Bindings

In our thesis, we have tested seven SQLite Haskell bindings. The selected bindings were the most downloaded ones from Hackage [6] the last 30 days (as of September 2016). The names and the number of download the last 30 days of the packages are the following:

- persistent-sqlite, 148 downloads [15]
- sqlite-simple, 54 downloads [16]
- direct-sqlite, 45 downloads [17]
- groundhog-sqlite, 31 downloads [18]
- HDBC-sqlite3, 30 downloads [19]
- sqlite, 29 downloads [2]
- simplest-sqlite, 26 downloads [20]

The bindings use different functions to connect, insert and receive values from the database. The approach of how to run a test case for each binding is therefore quite different. The following sections describe and give examples of how each binding communicates with the database. The bindings use prepared statements in order to insert the values in the database table.

5.1.1 The *persistent-sqlite* binding

The *persistent-sqlite* binding's approach for creating a table and executing queries (an example is shown in Figure 5.2) is similar to *groundhog-sqlite* but different from most of the other tested bindings. In the example, it uses the *share* function to define a database table with the name *Machine* and a column with the name *cost* which is of the type *Double*. The *Machine* data type must be defined before runtime in order to run the binding.

In the main function, the example (shown in Figure 5.2) starts by using *runSqlite* that creates a single connection to an existing database file or creates a new file with the given name. The function *runMigration* together with *migrateAll* creates the database table. The function *insert* creates a new record in the database and returns its *ID*. The *ID* can be used in the function *get* to retrieve the inserted value. The ID is unique for each table which means that *persistent-sqlite* will give a compile error if an ID from another table is used. Note that the *get* function can be used to retrieve one value but in order to retrieve more values, the function *selectList* can be used instead.

The return type of the function *runSqlite* will be *Maybe Machine* running in the IO monad. *Machine* is a Haskell record and the field *machineCost* (table name and column name together) will hold the inserted *Double*.

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persistLowerCase|
Machine
   cost Double
|]

main =
  runSqlite "test.db" $ do
      runMigration migrateAll
      insID ← insert $ Machine 12.34
      get insID  :: ReaderT SqlBackend (NoLoggingT (ResourceT IO)) (Maybe Machine)
```

Figure 5.2: An example of the *persistent-sqlite* binding

5.1.2 The *groundhog-sqlite* binding

The *groundhog-sqlite* binding uses a data type defined by the user to specify the table name and the types of columns. The data type must be defined statically in order to run the binding. An example of this is shown in Figure 5.3, where the name of the data type, *Machine*, is the database table name and *cost* is the column name with the type *Double* in the database table. The *entity* contains the name of the table. To use more tables, another *entity* needs to be added.

In the example, the function *withSqliteConn* is used to open or create a database file. The function *runDbConn* is performed to run actions such as insert and get within the connection. To create the table *Machine*, the function *runMigration* together with *migrate* (*undefined :: Machine*) is used. Then it uses similar functions as *persistent-sqlite* and it evaluates *Maybe Machine* in the IO monad to retrieve the *Double* from the database table.

```
data Machine = Machine { cost :: Double }

mkPersist defaultCodegenConfig [groundhog|
- entity: Machine
|]

main =
  withSqliteConn "test.db" $ runDbConn $ do
    runMigration $ migrate (undefined :: Machine)
    ins ← insert $ Machine 12.34
    get ins :: DbPersist Sqlite (NoLoggingT IO) (Maybe Machine)
```

Figure 5.3: An example of the *groundhog-sqlite* binding

5.1.3 The *sqlite-simple* binding

The *sqlite-simple* binding uses a data type (*TestField* in this case) defined by the user to hold the values of the rows (shown in Figure 5.4). The function *open* is executed to open or create a database file. The function *execute_* is used to create a database table and to insert a value in the table, the function *execute* (without underscore) is used. The inserted value can then be retrieved with the function *query_*. The return type of the *query_* function needs to be constrained in order to get the correct row type. The instance needs to be defined to read a row creating a value of *TestField*.

The *[TestField]* type represents a list of rows where *TestField* is one row. An instance *FromRow* needs to be defined to map the values from the *TestField* type to *SQLData*. The value can then be extracted from the *SQLData* type by pattern matching on the value constructor *SQLiteFloat*.

```
data TestField = TestField Double

instance FromRow TestField where
  fromRow = TestField <$> field

main  = do
  db ← open "test.db"

  execute_ db "create table Machine (cost Double)"
  execute db "insert into Machine (cost) values (?)" (Only (12.34 :: Double))

 query_ db "select * from Machine" :: IO [TestField]
 close db
```

Figure 5.4: An example of the *sqlite-simple* binding

5.1.4 The *direct-sqlite* binding

The *direct-sqlite* binding uses the function *open* to create or open a database file (shown in Figure 5.5). The function takes a filename represented as the *Text* datatype and therefore the *String* needs to be converted to *Text* using the function *pack* from the package *Data.Text*. The function *exec* executes a given query that in this case creates a database table. To insert a value in a statement, a statement needs to be created using the function *prepare*, the value needs to be inserted using the function *bindSQLData* and then the statement needs to be executed using the function *step*. To retrieve the values from the database, a select statement needs to be prepared and then executed using the function *step*. In order to get the value, the function *columns* needs to be used and it has the return type *[SQLData]*.

In order to retrieve the value from the column function, the IO action needs to be executed and its result will be in the IO monad. The return type of the column function has the type *[SQLData]*, which represents all column values of a row. In the example, *[SQLData]* will have the *Double* 12.34 packed with the value constructor of *SQLFloat* that can be extracted by pattern matching. To get a new row, the function *step* must be executed again.

```
main = do
  db ← open (pack("test.db"))

  exec db (pack "create table Machine (cost Real)")
  stmt ← prepare db (pack "insert into Machine (cost) values (?1)")
  bindSQLData stmt 1 (SQLFloat 12.34)
  step stmt

  statement ← prepare db (pack "select * from Machine")
  step statement
  columns statement :: IO [SQLData]
  close db
```

Figure 5.5: An example of the *direct-sqlite* binding

5.1.5 The *HDBC-sqlite3* binding

HDBC [21] provides bindings to for instance MySQL, Oracle, PostgreSQL, SQLite, ODBC-compliant databases, where the *HDBC-sqlite3* is the binding to SQLite. To connect to a database, the function *connectSqlite3* (shown in Figure 5.6) is used with the filename of the database as the argument. The function *runRaw* can be used to execute a query. In order to insert a value, a prepared statement needs to be created and then executed. This is performed using the functions *prepare* and *execute*. To retrieve the value from the database, the functions *prepare* and *execute* needs to be performed on the select query. The result is used by the function *fetchAllRowsAL'* to retrieve the values represented using the type *[[(String, SqlValue)]]*. The function *commit* is used to commit all the changes to the database file.

The type *[[(String, SqlValue)]]* represents list of database rows where *[(String, SqlValue)]* is one row. The *String* is the column name paired with the value in that column. The *SqlValue* can be converted to Haskell values with the function *fromSql* in the HDBC package.

```
main = do
  db ← connectSqlite3 "test.db"

  runRaw db "create table Machine (cost Real)"
  insert ← prepare db "insert into Machine (cost) values (?)"
  execute insert [(toSql (12.34 :: Double))]

  select ← prepare db "select * from Machine"
  execute select [ ]
  fetchAllRowsAL' select :: IO [[(String, SqlValue)]]

  commit db
  disconnect db
```

Figure 5.6: An example of the *HDBC-sqlite3* binding

5.1.6 The *sqlite* binding

The *sqlite* binding opens a connection to a database by using the function *openConnection* (shown in Figure 5.7) that takes the name of the database as the argument. The function *execStatement* uses the connection to the database (the first argument) and evaluates the query, which is the second argument to the function. The return value of the *IO* action is *Either String [[Row Value]]*. In order to insert a value in the database table, a prepared statement needs to be created and then executed together with a query, which can be performed with the function *execParamStatement_*.

The *select* query returns *Either String [[Row Value]]*. The *Either* type makes it possible for a function to either return one type or another. In this case, the select query will return either the *Left* constructor, which holds a *String* that displays why the query failed or the *Right* constructor, which contains the result of the query using the type *[[Row Value]]*. The *[[Row Value]]* represents list of rows that is selected (one list is unnecessary) and *Row Value* represents a single row and it is of the type *[(ColumnName, Value)]*, where *columnName* is a *String* and *Value* is a type holding all types that sqlite can return. The *Double* value from Figure 5.7 can easily be extracted by pattern matching on *Double*.

```
main = do
  db ← openConnection "test.db"

  execStatement db "create table Machine (cost Real)"  :: IO (Either String [[Row Value]])
  execParamStatement_ db "insert into Machine (cost) values (?1)" [("?1", Double 12.34)]
  execStatement db "select * from Machine"  :: IO (Either String [[Row Value]])

  closeConnection db
```

Figure 5.7: An example of the *sqlite* binding

5.1.7 The *simplest-sqlite* binding

This section describes an example of how to use the binding but it also explains four important functions that will be necessary to understand for two of the bugs.

The *simplest-sqlite* binding uses the function *withSQLite* (shown in Figure 5.8) to open a connection to a database, which is used to execute SQL statements. All the statements use the function *withPrepared* that takes a function as input to be able to define the action that should be performed with the given statement. The input function needs to use the function *step* to execute the query. In order to insert a value in the table, the value needs to be bound to a prepared statement using the bind function and the *step* function needs to be used to execute the query. To retrieve the answer from the *select* query, the function *column* is used together with a column index (zero represents the first column).

The answer from *withSQLite* will have the return type (*Double*, *String)* running in the IO monad. The *Double* is the inserted value 12.34 and the *String* displays potential errors that occurred by executing the query.

```
main = do
  withSQLite "test.db" (\db → do
      withPrepared db ("create table Machine (cost Real)") (\stmt → step stmt)
      withPrepared db ("insert into Machine (cost) values (?1)") (\stmt → do
                                  bind stmt "?1" (12.34 :: Double)
                                  step stmt)
      withPrepared db ("select * from Machine") (\stmt → do
                                  step stmt
                                  column stmt 0) :: IO (Double,String))
```

Figure 5.8: An example of the *simplest-sqlite* binding

The binding uses two functions internally (*packCString* and *useAsCString*) that were involved in causing two of the bugs that we found. Two other relevant functions, *packCStringLen* and *useAsCStringLen*, will also be explained.

The first function that the binding uses is *packCString* from the package *Data.ByteString*. Its type signature and an example of how to use it are shown in Figure 5.9. The example first creates a *CString* using the function *newCString*. The *CString* is packed to a *ByteString* using the function *packCString* and then the *ByteString* is unpacked and printed. As shown in the *Output*, it prints the *String abc* and it means that the original *String* has been cut at *\NUL* (a *Null*-terminator). In the documentation for the *packCString*, it is stated that the *CString* must be *Null*-terminated, which causes it to be cut prematurely.

Type signature

```
packCString :: CString → IO ByteString
```

Example

```
cstr ← newCString "abc\NULdef"
bytStr ← packCString cstr
print $ unpack bytStr
```

Output

“abc”

Figure 5.9: The type signature of the function *packCString* and an example of it

A function that is similar to *packCString* is *packCStringLen*. The difference is that the function takes a *CStringLen* that contains a pointer to the first *Char* and the length of the *Char* sequence as input. The *String* does not need to be *Null*-terminated since it will be managed on the Haskell heap. An example of using *packCStringLen* is shown in Figure 5.10, where the whole string containing *\NUL* is stored in a *ByteString*, and subsequently printed.

Type signature

```
packCStringLen :: CStringLen → IO ByteString
```

Example

```
cstr ← newCStringLen "abc\NULdef"
bytStr ← packCStringLen cstr
print $ unpack bytStr
```

Output

“abc\NULdef”

Figure 5.10: The type signature of the function *packCStringLen* and an example of it

The second function that the binding uses is *useAsCString* from the package *Data.ByteString*. Its type signature and an example of how it can be used are shown in Figure 5.11. The function takes a *ByteString* and a function that requires a *Null*-terminated *CString*. The *ByteString* is converted to a *CString* and the function is applied on the *CString*, where the resulting type depends on the return type of the function. In the example, *useAsCString* will cut the *ByteString* at *\NUL* if the length of the *Bytestring* is not calculated.

Type signature

```
useAsCString :: ByteString → (CString → IO a) → IO a
```

Example

```
let byteStr = pack "abc\NULdef"
useAsCString byteStr (\cs → packCString cs)
```

Figure 5.11: The type signature of the function *useAsCString* and an example of it

The function *usAsCStringLen* (shown in Figure 5.12) is similar to *usAsCstring* but it provides a *CStringLen* instead of a null-terminated *CString*.

Type signature

```
useAsCStringLen :: ByteString → (CStringLen → IO a) → IO a
```

Example

```
let byteStr = pack "abc\NULdef"
useAsCStringLen byteStr (\csl → packCStringLen csl)
```

Figure 5.12: The type signature of the function *useAsCStringLen* and an example of it

5.2 Implemented Properties

Two different properties have been created (shown later). The properties test the bindings by combining most of the functions that they use to interact with the database. The difference between the properties is that the first one, *prop_bind_string*, inserts the values as *String* while the other property, *prop_bind_value*, binds the values and inserts them as their types. The reason for creating *prop_bind_string* was to test that the binding is able to convert the *Strings* from Haskell containing the values to *CStrings* and execute the *CString* correctly in the database. The reason for creating the *prop_bind_value* property is to test that the Haskell types are correctly converted in the binding, for instance that a *Double* is converted to a *CDouble* and then inserted correctly in the database.

Both of the properties use the structure shown in Figure 5.13 as pseudocode. First, a connection to the database is created, and then a table with four columns of the types *Text*, *Integer*, *Real* and *Blob* is created. After the table is created, the values are inserted in the respective columns. Then the properties *select* everything from the table and store the result in a variable (*answer*). Finally, they drop the table and close the connection. The database connection is now closed, but since the result from the *select* query is stored, it is possible to test if the read values are the same as the inserted ones.

```
open "test.db";
create table tab1 (col1 Text, col2 Integer, col3 Real, col4 Blob);
insert into tab1 (col1, col2, col3, col4) values (str, int, dou, byteStr);
answer ← select * from tab1;
drop table tab1;
close "test.db";
let [String retStr, Int retInt, Double retDou, ByteString retBytStr] = answer
assert (str == retStr && int == retInt && dou == retDou && byteStr == retByteStr)
```

Figure 5.13: Pseudocode for the properties behaviour

In order to make the properties as general as possible, the Haskell record data type *SQLiteBind* (shown in Figure 5.14) was created. A value of the *SQLiteBind* data type consists of functions that a specific binding uses to perform different actions involving the database. For example, the functions *openConn* and *closeConn* are used by both properties to open and close the connection to the database. The function *openConn* takes a *String* as input, which contains the database name and returns a database handle running in the IO monad. The function *closeConn* takes the database handle as input and returns an empty IO action.

The functions *createQ*, *insertQ*, *selectQ*, *deleteQ*, *getVal* and *static* are used in the property *prop_bind_string*. The function *getVal* is used to retrieve the results from a select query as a list since the return type of the query often differs between the bindings. The function *static* indicates if the binding needs to use the same table name during testing or not (the *persistent-sqlite* and *groundhog-sqlite* bindings need this, shown in section 5.1.1 and 5.1.2).

The functions *prepCreateQ*, *prepInsQ*, *prepSelQ*, *prepDelQ*, *prepGetVal* and *prepParam* are used in the property *prop_bind_value*. The functions execute the same queries as before but with prepared statements instead of directly as *Strings*. The function *prepGetVal* has the same functionality as *getVal*, but is used with queries involving prepared statements. The function *prepParam* is used to create prepared statements by defining a list of *Strings* that represent each value's position when it is inserted in the prepared statement. For instance, the function *prepParam* can have the value *["?1", "?2", "?3", "?4"]*.

```
data Vals = I Int | D Double | S String | B ByteString | N Int
 deriving Show

data SQLiteBind conn res prepRes =
SQLiteBind :: { openConn    :: String → IO conn
              , closeConn   :: conn  → IO ( )
              , createQ     :: conn  → String → IO ( )
              , insertQ     :: conn  → String → IO ( )
              , selectQ     :: conn  → String → IO res
              , deleteQ     :: conn  → String → IO ( )
              , getVal      :: res   → [Vals]
              , static      :: Bool

              , prepCreateQ :: conn     → String → IO ( )
              , prepInsQ    :: conn     → String → [Vals] → IO ( )
              , prepSelQ    :: conn     → String → IO prepRes
              , prepDelQ    :: conn     → String → IO ( )
              , prepGetVal  :: prepRes  → [Vals]
              , prepParam   :: [String]


              }
```

Figure 5.14: The data type of the SQLiteBind

An example value of the *SQLBind* record providing the operations of the *sqlite* binding is shown in Figure 5.15. Similar values needed to be provided for all tested bindings. In this example, the binding uses the function *openConnection* to open a database connection. In order to execute queries, the binding uses the function *execStatement* in most cases. The only exception is the function *execParamStatement* that is used to insert prepared statements.

```
let sqliteBindSqlite = SQLiteBind {
        openConn = openConnection,
        closeConn = closeConnection,

        createQ = \conn query → (execStatement conn query
                                    :: IO (Either String [[Row Value]])) >> return ( ),
        insertQ = createQ sqliteBindSqlite,
        selectQ = \conn query → (execStatement conn query
                                    :: IO (Either String [[Row Value]])),
        deleteQ = createQ sqliteBindSqlite,
        getVal = \val → if isRight val then
                          map (\(x,y) → (getValueSqlite y)) $ concat $ concat
                                    $ fromRight' (val :: (Either String [[Row Value]]))
                        else [S "", I 0 ,D 0.0],
        static = False,
        prepCreateQ = createQ sqliteBindSqlite,
        prepInsQ = \conn query [S str,I int,D dou,B bytStr] →
                          (execParamStatement conn query [(":1",Text str),
                          (":2",Int (fromIntegral int)),(":3",Double dou ),
                          (":4",Blob bytStr)]
                        :: IO (Either String [[Row Value]])) >> return ( ),
        prepSelQ = selectQ sqliteBindSqlite,
        prepDelQ = deleteQ sqliteBindSqlite,
        prepGetVal = getVal sqliteBindSqlite,
        prepParam = [":1",":2",":3",":4"]}
```

Figure 5.15: Defining the *sqlite* binding's functions with the *SQLiteBind* type

We chose to use random table and column names to test whether the bindings can handle UTF-8 *Chars*. The table and the column names cannot be empty or contain spaces or non-alphabetic Unicode characters. The reason why non-alphabetic Unicode characters cannot be used is that SQLite does not support ASCII symbols like "+" and "–" in a table name or column name without using brackets. The names cannot either contain SQL keywords, such as *select*, without using brackets.

To create a valid table and column name, the generators *letterGen* and *WordGen* (shown in Figure 5.16) were created and also a *forbidden* list that contains all SQL keywords. The *letterGen* only generates Unicode letters by testing a generated *Char* using the function *isLetter*. The *letterGen* generator is then used in *WordGen* in order to create a non-empty *String* from the letters. The *String* cannot be a SQL keyword since it is compared with the *forbidden* list. The generator *WordGen* is used in an arbitrary instance for the type *Word*, as shown in Figure 5.17 (not related to the Haskell type *Word*), which is used to generate table names. To generate column names, we also need to make sure that their names in a single table are different. Therefore, the function *allDifferent* was created and it checks that every *String* in the list is unique. Two more arbitrary instances for the types *ThreeWords* and *FourWords* (shown in Figure 5.17) were created that generate three or four *Strings* using the generator *WordGen* making sure that they are unique.

```
letterGen :: Gen Char
letterGen = suchThat arbitrary isLetter

WordGen :: Gen String
WordGen = suchThat (listOf1 letterGen) (\x → map toLower x `notElem` forbidden)
```

Figure 5.16: Generators for *Strings*

```
newtype Word = Word String
 deriving (Show,Eq)

instance Arbitrary Word where
 arbitrary = do
   l ← WordGen
   return (Word l)

newtype FourWords = FourWords [String]
 deriving (Show,Eq)

instance Arbitrary FourWords where
 arbitrary = do
   l ← suchThat (vectorOf 4 WordGen) allDifferent
   return (FourWords l)
```

Figure 5.17: Arbitrary instance of the table and column names

Generators for each of the types *String*, *Int64*, *Double* and *ByteString* were also created, mainly to disable shrinking since it took additional calculation time and did not contribute to finding the cause of the problem. The *Int64* type was chosen since most SQLite bindings use *Int64* to make sure that 64-bit Integers are representable. QuickCheck does not support direct generation of *ByteString*, but it can generate a list of *Word8* values that then can be converted to a *ByteString* using the function pack (from the package *Data.ByteString*).

In order to insert a *Bytestring* as a *String* in SQLite without converting the *Bytestring* directly to a *String* (and possibly lose data due to *Bytestring* to *String* conversion), we chose to insert the *Bytestring* in hexadecimal format. The hexadecimal format and the corresponding *ByteString* was generated using the Arbitrary instance *ByteStringS* (shown in Figure 5.18). The generator starts by generating a *ByteString* and then calls the function *bsToHex* that creates a *String* that contains the hexadecimal representation of the *ByteString*. The function converts the *ByteString* to a list of *Word8*, which is a list of 8-bit *Ints* and converts each *Word8* to *Int* and runs the function *intToHex* on each *Int*. The instance then returns a pair that contains the hexadecimal representation of *Bytestring* as a *String* and the corresponding *Bytestring*.

For instance, if the function *intToHex* takes *Int* 26 as input, then it is used by the function *DivMod* which divides 26 with the value 16 and returns the value 1 and the remainder 10. The function *intToDigit* convert the values to '1' and 'a', which is combined to "1a" and returned by the *intToHex* function.

```
newtype ByteStringS = ByteStringS (String, B.ByteString)
deriving (Show,Eq)
instance Arbitrary ByteStringS where
 arbitrary =  do
   bytStr ← arbitrary :: Gen B.ByteString
   let str = bsToHex bytStr
   return (ByteStringS (str,bytStr))

bsToHex :: ByteString → String
bsToHex bs = concatMap (intToHex . fromIntegral) (B.unpack bs)
  where intToHex :: Int → String
    intToHex n = let (qu, re) = n `divMod` 16
      in  [intToDigit qu, intToDigit re]
```

Figure 5.18: *ByteString* generator for the property *prop_bind_string*

The first tested property was *prop_bind_string* (shown in Figure 5.19) that takes seven parameters. The first parameter (*bind*) is the *SQLiteBind* value of the binding that will be tested. The second (*table*) is a random table name of the type *Word*. The third is a list of four random *Strings* that are used as column names. The last four parameters (*str*, *num*, *dou* and a tuple of *hexStr* and *BytStr*) represent a *String*, an *Int*, a *Double* and a pair containing a *String* and a *ByteString* where the *String* is the hexadecimal representation of the *ByteString*. These values (except *BytStr*, which is represented by *hexStr*) are the values that will be inserted in the table.

The property starts execution by connecting to a database, creating the table and inserting the values. The values are then retrieved before the table is removed and the connection is closed. The retrieved values are then compared with the inserted values.

To create the table, the property uses the function *createTable'* (its implementation is not shown) that takes three parameters, a *Bool*, a *String* and a list of *Strings*, and returns a SQL *String*. The *Boolean* argument determines if a static table name needs to be used, which is required by the *persistent-sqlite* and *groundhog-sqlite* bindings. This means that if the Boolean argument is False, then the table name needs to be defined outside of the property. Otherwise, the randomly generated table name is used as the second parameter of the *createTable'* function. The last parameter is a list of column names. The function simply creates the SQL *String* that is used to create the table with the specified name, the column names and the types of the columns. When the SQL *String* is created, the property uses the function from the *SQLiteBind* value, *createQ* to execute the query. For instance, in the case of the *sqlite* binding, *createQ* refers to the function *execStatement* that executes the query.

The *insert* and *select* queries are performed in a similar way as creating the table. They use the functions *insertTable* and *selectTable* that both create the *Strings* containing the respective queries. The result from the *select* query is extracted and stored to a variable (*ans*). The table is then deleted (the function *dropTable* creates the SQL *String*) from the database and the connection is closed.

The result from the *select* query is then used with the function *getVal* from *SQLiteBind* in order to retrieve the values that the database has returned. The values are compared against the inserted values in the assertion (in the last line). The *monitor* function in the property will print the inserted and expected values if a counterexample is found.

```
prop_bind_string bind (Word table) (FourWords [colStr, colNum, colDou, colBytStr])
(Word str) (IntS num) (DoubleS dou) (ByteStringS (hexStr, bytStr)) = monadicIO $ do

  ans ← run $ do
        conn ← (openConn bind) "test.db"
        (createQ bind) conn $ createTable' (static bind) table
                                           [colStr, colNum, colDou, colBytStr]
        (insertQ bind) conn $ insertTable (static bind) table
                                  [colStr,colNum,colDou, colBytStr]
                                     [str, show num, show dou, "x'"++ hexStr ++ "'"]
        ans ← (selectQ bind) conn $ selectTable "*" table ""
        (deleteQ bind) conn $ dropTable table
        (closeConn bind) conn
        return ans
  let [S a, I b, D c, B d] = (getVal bind) ans
  monitor (
       whenFail'
            (putStrLn $ "--------- Database:  str: " ++ a ++ " expected: " ++ str
                        ++ " num: " ++ show b ++ " expected: " ++ show num
                        ++ " dou: " ++ show c  ++ " expected: " ++ show dou
                        ++ " bytStr: " ++ T.unpack d ++ " expected: "
                        ++ T.unpack bytStr))
  assert( a == str && b == num && c  == dou &&  d == bytStr)
```

Figure 5.19: The property *prop_bind_string*

The second property, *prop_bind_value* (shown in Figure 5.20), is similar to *prop_bind_string* but it uses a prepared statement and binds the values to the statements before their execution. Another difference is how the *ByteString* values are inserted. In *prop_bind_string*, they are inserted as *String* values using the hexadecimal representation of the *ByteString*, while in *prop_bind_value* they are inserted directly as *ByteString* values.

The property *prop_bind_value* uses *prepCreateQ* instead of *createQ* and so on, since some bindings use different functions to create and insert values with prepared statements. In the *insert* function, it also uses the function *prepParam* (from the *SQLiteBind* value). The function defines in which position the values should be inserted in the prepared statement. The function *insertTablePrep* creates the *String* query using the S*trings* from *prepParam* and it is used in the prepared statement. For instance, a *String* query from *insertTablePrep* could be "*Insert into abc (A,B,C,D) values (?1,?2,?3,?4)*", where the provided values are bounded to the four question mark variables, which corresponds to the columns A, B, C and D (Binding Values to Prepared Statements was described in section 4.2).

```
prop_bind_value bind (Word table) (FourWords [colStr, colNum, colDou, colBytStr])
(Word str) (IntS num) (DoubleS dou) bytStr = monadicIO $ do

  ans ← run $ do
    conn ← (openConn bind) "test.db"
    (prepCreateQ bind) conn $ createTable' (static bind) table
                                           [colStr, colNum, colDou, colBytStr]
    (prepInsQ bind) conn (insertTablePrep table [colStr, colNum, colDou, colBytStr]
                          (prepParam bind)) [S str, I num, D dou, B bytStr]
    ans ← (prepSelQ bind) conn $ selectTable "*" table ""
    (prepDelQ bind) conn $ dropTable table
    (closeConn bind) conn
    return ans
  let [S a, I b,D c, B d] = (prepGetVal bind) ans
  monitor (
        whenFail'
            (putStrLn $ "--------- Database: " ++ " str: " ++ a ++ " expected: " ++ str
                        ++ " num: " ++ show b ++ " expected: " ++ show num
                        ++ " dou: " ++ show c  ++ " expected: " ++ show dou
                        ++ " bytStr: " ++ T.unpack d ++ " expected:  "
                        ++ T.unpack bytStr))
  assert( a == str && fromIntegral b ==  num  && c == dou &&  d == bytStr)
```

Figure 5.20: The property *prop_bind_value*

5.3 Testing the properties

This section explains how the tests of the SQLite bindings are performed using the testing tool QuickCheck and the two properties. In order to test a binding with a property, an *SQLiteBind* value referencing the binding's functions needs to be chosen. We tested both properties with every binding. An example of how the tests are performed is shown in Figure 5.21, where the function *quickCheckWithResult* is used. The function takes two parameters, the first is the test arguments and the second is the property that should be tested. The first parameter can specify, for example, the maximum number of passed tests (it stops earlier if an error occurs) with the constructor *maxSuccess*, on top of the record *stdArgs* that contains the default test arguments. In the example, QuickCheck will run 500 000 tests with both properties on the *sqlite* binding (since *sqliteBindSqlite* is used).

```
Runs the property prop_bind_string
quickCheckWithResult stdArgs {maxSuccess = 500000} (prop_bind_string sqliteBindSqlite)

Runs the property prop_bind_value
quickCheckWithResult stdArgs {maxSuccess = 500000} (prop_bind_value sqliteBindSqlite)
```

Figure 5.21: How to perform the testing on the bindings

In order to decide when the testing is complete, many parameters can be considered. For instance, code coverage, the number of tests, the number of bugs that should be found and the amount of testing time. We cannot ensure that all bindings have bugs so that is not an option for us. The code coverage of the bindings does not increase by performing more tests since a single test will likely cover all the functions that are tested in the bindings. The tests after the first one will be testing the different values for each type but not increasing the code coverage. Therefore, we consider the number of tests and testing time as the best options for our approach. All the bugs in the performed testing are found in less than 1 million tests and it could be considered as a lower bound for the testing. It is difficult to know when the random testing has tested enough, but we consider our testing to be complete when the tests run for at least a day.

6 Bugs

This chapter explains four bugs that we have found. Examples are given to understand what the bugs are, why they appear and to show the solutions to the bugs. The bugs involve three different data types: *String*, *Double* and *ByteString*. The bugs were found with the two properties *prop_bind_string* and *prop_bind_value*, (described in section 5.2) and in three different bindings: *sqlite*, *HDBC-sqlite3* and *simplest-sqlite*.

6.1 Conversion Bug Involving the UTF-8 Encoding

The first bug was found when we performed random testing of the property *prop_bind_string* but the bug can also occur with the property *prop_bind_value*. Figure 6.1 shows the output of QuickCheck testing the property when the bug is triggered, reporting a discrepancy in a *String* returned from the database.

```
--------- Database:  str: Ã¸ expected: ø num: 1 expected: 1
dou: 0.40735100738214747 expected: 0.40735100738214747
bytStr: \US expected: \US

*** Failed! Assertion failed (after 1 tests):
Word "h"
FourWords ["MR", "mO", "PG", "J"]
Word "\248"
IntS 1
DoubleS (0.40735100738214747)
ByteString ("1f", "\US")
```

Figure 6.1: The output from the test that failed

Figure 6.1 shows a counterexample that was found with the property *prop_bind_string*. The output lists the returned values from the database (after the select query) and the expected values that were inserted from Haskell. There are four returned values: *str* (*String* Ã¸), *num* (*Int* 1), *dou* (*Double* 0.40735100738214747) and *bytStr* (*ByteString* \US). The lines after *Assertion failed* are QuickCheck's default output when a counterexample has been found and it shows all the inputs to the property which made it fail. The input *Word* is the table name, *FourWords* is the column names, the rest are the input values (*str*, *num*, *dou* and *bytStr*). The reason that the property failed is that the received *String* is "Ã¸" while the expected one is "ø" (also represented as \248).

After a small number of tests with QuickCheck, it was shown that the bug only occurred with non-ASCII *Chars*. A recreated example of the programming code that demonstrates the problem is shown in Figure 6.2. The code simply inserts a *String* containing the non-ASCII character 'ø' into the database and then executes a select query on the table it was inserted into.

The first two lines printed by the function contain the *Strings* "ø" and "Ã¸" (note the comma after Ã) which represent the inserted *String* and the received *String* from the database. These two lines should both print "ø". Furthermore, the last two lines prints the *Strings* "/248" and "195/184" (it should be "/248") which represent the Unicode code points that the *Strings* correspond to.

The bug was caused by the *sqlite* binding package not converting the UTF-8 representation ("\195\184") of the *String* "ø" to Unicode code points ("\248") which Haskell's *Char* type uses.

To fix the bug, the binding had to convert the returned *String* after executing a select query from the database. The *String* is converted to Unicode code points used by the Haskell *Char* type and could then be returned from the binding.

The bug was found in the *sqlite* binding package 0.5.2.2 on Hackage and it has been reported. However, the package's GitHub repository had a recent commit that had already fixed the bug. Version 0.5.3 of the package was released on Hackage after our report and included a fix for the bug.

```
main = do
  db ← openConnection "test.db"
  execStatement_ db "create table abc (A text);" :: IO (Maybe String)
  execStatement_ db "insert into abc (A) values ('ø');" :: IO (Maybe String)
  table ← execStatement db "select * from abc;" :: IO (Either String [[Row Value]])

  let value = snd $ head $ concat $ concat $ fromRight table
  putStrLn "ø"
  putStrLn $ getValue value
  putStrLn $ show "ø"
  putStrLn $ show $ getValue value
 where fromRight (Right b) = b
       getValue (Text a) = a
```

Output:

```
"ø"
"Ã¸"
"\248"
"\195\184"
```

Figure 6.2: An example when the bug occurred and its output

6.2 Incorrect Double Value

Many bindings seemed to exhibit an incorrect behaviour when some *Double* values were inserted as *Strings* when running the property *prop_bind_string*. An example of such behaviour is shown in Figure 6.3, where the *Double* 62.027393 is inserted as a *String* and the retrieved value turns out to be 62.027393000000004, which means that the returned value is slightly larger than the inserted one. This test was performed on all seven tested bindings and only one (*HDBC-sqlite3*) retrieved the correct value 62.027393.

```
create table tab1 (col Real);
insert into tab1 (col) values (62.027393);
select * from tab1;

Output:
62.027393000000004
```

Figure 6.3: An example that were tested with *Double* values

However, all tested bindings returned the inserted value if the value was inserted as a *Double* using prepared statements. An example of how a *Double* value is bound to a prepared statement using the binding HDBC-sqlite3 is shown in Figure 6.4.

```
insert ← prepare db "insert into Machine (Cost) values (?)"
execute insert [(toSql (12.34 :: Double))] >> return [ ]
```

Figure 6.4: An example of how to bind *Double* values in *HDBC-sqlite3*

Sometimes when *Doubles* are converted to *Strings* in for instance GHCi, their values can be truncated. An example of this is shown in Figure 6.5, where both values output the same result. This could be a problem both when the value is inserted in the database since it can insert a truncated value but also in testing since two different values will be printed in the same way, which might make it difficult to say why a test case is failing.

```
show 1.1234567890123456
show 1.12345678901234567

Output
"1.1234567890123456"
"1.123456789012346"
```

Figure 6.5: An example of how *Double* can be truncated in GHCi

To avoid this problem, we used differential testing to compare the results returned by two of the tested bindings, *HDBC-sqlite3* and *sqlite*. The differential testing is performed by creating a table with six different combinations of *insert* and *select* statements (shown in Table 3). These statements are executed with different bindings (note that *sqlite* is used with and without *bind* when the value is inserted).

The left-most column in Table 3 indicates which binding that was used to *select* the value and the first line is how the value was inserted in the table. For instance, the left bottom value (62.027393000000004) uses *HDBC-sqlite3* to insert the value in the table but *sqlite* to retrieve it. If we assume that the slightly larger value is the correct value, then *HDBC-sqlite3* gives an incorrect value when it performs both the *insert* and *select* but it is also incorrect when *sqlite without bind* inserts the value and *HDBC-sqlite3 select* the value. This means that a bug can occur when the value is retrieved in the *HDBC-sqlite3*.

Insert / Select	HDBC-sqlite3	sqlite with bind	sqlite without bind
HDBC-sqlite3	62.027393	62.027393	62.027393
sqlite	62.027393000000004	62.027393	62.027393000000004

Table 3: Differences between the *HDBC-sqlite3* binding and the *sqlite* binding

In order to increase the probability that all bindings except *HDBC-sqlite3* had an incorrect behaviour, the pseudocode in Figure 6.6 was executed in C (it will eliminate any potential type conversion errors that can appear when executing SQLite code from Haskell). The third to the fifth line inserts the value as a prepared statement where the value is bound to the question mark. The sixth and seventh lines insert the value directly as a *String*. The last two lines select all values and the output of it is 62.027393 in both cases. The select statement is performed to find differences in the results.

```
1  prepare create table tab1 (col1 Real);
2  step
3  prepare insert into tab1 (col1) values (?);
4  bind value 62.027393
5  step
6  prepare insert into tab1 (col1) values (62.027393);
7  step
8  prepare select * from tab1;
9  step
```

Figure 6.6: Pseudocode for testing the SQLite values

If the stored values are compared with for instance a cross join (shown in Figure 6.7), then the result table should have four rows if the values are equal and two rows if they are not equal.

```
prepare select * from tab1 as a cross join tab1 as b where a.col1 = b.col1;
step
```

Figure 6.7: Cross join example

The result from the query in Figure 6.7 is two rows and it means that they differ. Therefore, the value 62.027393000000004 was bound in the prepared statement instead of 62.027393 (line 4 in Figure 6.6). The cross join test was run again and returned four rows which mean that the inserted values are equal. This means that if the value 62.027393 is inserted as *String*, then the value is changed to 62.027393000000004. This may be a bug in the SQLite database and the behaviour has been reported to SQLite.

To understand why *HDBC-sqlite3* did not return the value 62.027393000000004 that the SQLite database stored, we have to know if the bug occurred when the value was inserted or retrieved. After inspecting the function that retrieves data in the *HDBC-sqlite3* binding, it became apparent that the binding uses the function *sqlite3_column_text*, which truncates the *Double* after the first 15 significant digits. This means that the *Double* 62.027393000000004 becomes 62.0273930000000 (the two last digits are removed), which is the same as 62.027393. Therefore, the value seems to be correct since the function cuts it, but it is incorrect because the stored value is larger.

The bug appears since the SQLite function *sqlite3_column_text* is used to fetch values of all types except *Null*, and therefore truncates *Doubles* that contain more than 15 significant digits. To fix the bug, the SQLite function *sqlite3_column_double* was used instead, which does not truncate the stored *Double*.

6.3 Null-termination of ByteString

The third bug involved null-termination of *ByteStrings* that were stored in the database. The bug was found with the property *prop_bind_value* and it only appeared in the *simplest-sqlite* binding. An example input that made the property fail is shown in Figure 6.8. It also shows the *ByteString* that was retrieved by the property.

Input:
"Y\191\210\187\DC1\"\NULWeD"

Output:
"Y\191\210\187\DC1\""

Figure 6.8: An example when the null-termination of *ByteString* can appear

The bug appears when the *simplest-sqlite* binding uses the function *packCString* (its type signature is shown in Figure 6.9) to retrieve a value from the database. The function truncates the *ByteString* at *\NUL* and therefore the result in the example becomes shorter. In order to fix the bug, the function *packCStringLen* can be used, which takes a *CStringLen* as input instead of a *CString*. The data type *CStringLen* contains a *CString* and an *Int* that specifies the length of the *CString* in bytes. This extra information means that it is possible for the function *packCStringLen* to retrieve a *ByteString* that contains *Null* characters correctly, since the *Null* character is not used for termination. In addition, we need to use the C function *sqlite3 _column_bytes* that retrieves the number of bytes used to store a particular value in the database. An example of actions performed by the original and new code is shown in Figure 6.9.

Type signatures
```
packCString :: CString → IO ByteString
packCStringLen :: CStringLen → IO ByteString
type CStringLen = (Ptr CChar, Int)
```

Original code
```
cStr ← sqlite3_column_blob stmt intCol
packCString (castPtr cStr)
```

New code
```
cStr   ← sqlite3_column_blob stmt intCol
bytes ← sqlite3_column_bytes stmt intCol
packCStringLen (castPtr cStr, fromIntegral bytes)
```

Figure 6.9: The solution to the null-termination bug

Another problem appeared when the null-termination bug was fixed. The problem was that a *Null*-termination was added to the *ByteString*. An example of an input that demonstrates the problem is shown in Figure 6.10. Note that even though *\NUL* appears in the *ByteString*, it is also added to the end of it.

Input:
"\184<<\ETB\GS\234\132\&36?\DLE\232z\NUL3"

Output:
"\184<<\ETB\GS\234\132\&36?\DLE\232z\NUL3\NUL"

Figure 6.10: An example of the problem where *\NUL* is added at the end of the *ByteString*

The problem appeared since the *bind* function uses the function *useAsCString*, which expects a *Null*-terminated *CString* (the type signature is shown in Figure 6.11). Figure 6.11 also shows an example of how the function *useAsCString* was used within the *bind* function and how we used *useAsCStringLen* instead.

The problem in the original code is the fourth parameter to the function *c_sqlite3_bind_blob*. The SQLite's documentation states that "*If the fourth parameter to sqlite3_bind_blob() is negative, then the behavior is undefined*". Instead, the fourth parameter should be the number of *Bytes* for the value that will be inserted. Our code simply passes the number of bytes provided by *useAsCStringLen* as the fourth argument.

Type signatures

```
useAsCString :: ByteString → (CString →  IO a) →  IO a
useAsCStringLen :: ByteString →  (CStringLen →  IO a) → IO a
```

Original code

```
sqlite3BindBlob :: Stmt → Int → BS.ByteString → IO ()
sqlite3BindBlob (Stmt sm) i s = BS.useAsCString s $ \cs → do
   ret ← c_sqlite3_bind_blob sm (fromIntegral i) cs (- 1) nullPtr
   when (ret /= sQLITE_OK) $ sqliteThrow "Cannot bind text" ret
```

New code

```
sqlite3BindBlob :: Stmt → Int → BS.ByteString → IO ()
sqlite3BindBlob (Stmt sm) i s = BS.useAsCStringLen s $ \(cs,bytes) → do
   outPtr ← mallocBytes bytes
   memcpy outPtr cs (fromIntegral bytes)
   ret ← c_sqlite3_bind_blob
            sm (fromIntegral i) (castPtr outPtr) (fromIntegral bytes) nullPtr
   when (ret /= sQLITE_OK) $ sqliteThrow "Cannot bind text" ret
```

Figure 6.11: The solution to remove the additional *Null*

6.4 Memory Allocation Bug

Another bug appeared when QuickCheck was run on the property *prop_bind_value* with the binding *simplest-sqlite*, which led to an exception shown in Figure 6.12. The output indicates that it is a problem with decoding a *Text*, which in our property corresponds to a *String* packed as a *Text*.

```
*** Failed! Exception: 'Cannot decode byte '\x9d':
Data.Text.Internal.Encoding.decodeUtf8: Invalid UTF-8 stream' (after 4813 tests)
```

Figure 6.12: An output when the property *prop_bind_value* was tested on *simplest-sqlite*

There are only two possible places where this problem can appear, either when the value is inserted in the database or when it is retrieved. After the code was examined, we concluded that the bug occurs when the value is inserted.

When the binding *simplest-sqlite* gets a *Text* as input, it converts the *Text* to a *ByteString* and then sends it to the function *sqlite3BindByteString*, which binds the *ByteString* to a SQLite statement. The statement is then executed with the function *step* that performs an insert to the database.

Our fix for the bug and the original code are shown in Figure 6.13. The problem was that the *ByteString* '*cs*' was incorrectly allocated in the memory and was therefore incorrectly inserted in the database. After the *ByteString* was retrieved from the database, the *Text* package tried to decode the incorrect *ByteString* and therefore raised an exception.

Our code first extracts the length of the *Bytestring* (the byte offset where the *Null*-terminator would occur assuming the *Bytestring* was *Null*-terminated) and allocates enough memory to store the *ByteString*. A pointer to the stored *ByteString* is used as the third parameter and the length of the *ByteString* as the fourth parameter to the function *sqlite3_bind_text*. The exception disappears after these changes.

Original code

```
sqlite3BindByteString :: Stmt → Int → BS.ByteString → IO ()
sqlite3BindByteString (Stmt sm) i s = BS.useAsCString s $ \cs → do
  ret ← c_sqlite3_bind_text sm (fromIntegral i) cs (- 1) nullPtr
  when (ret /= sQLITE_OK) $ sqliteThrow "Cannot bind text" ret
```

New code

```
sqlite3BindByteString :: Stmt → Int → BS.ByteString → IO ()
sqlite3BindByteString (Stmt sm) i s = BS.useAsCStringLen s $ \ (cs,bytes) → do
  outPtr ← mallocBytes bytes
  memcpy outPtr cs (fromIntegral bytes)
  ret ← c_sqlite3_bind_text
          sm (fromIntegral i) (castPtr outPtr) (fromIntegral bytes) nullPtr
  when (ret /= sQLITE_OK) $ sqliteThrow "Cannot bind text" ret
```

Figure 6.13: The solution to the memory allocation bug

The changes also fixed another problem that appeared during the testing. An example of it is shown in Figure 6.14. The figure indicates that it was a problem with the encoding or decoding of the *String* since it returned a non-ASCII *Char* (represented with an underscore in the figure) instead of the expected *String*. This was also caused by incorrect memory allocation but the *Text* package managed to decode it but the result is incorrect. However, the problem was solved when the solution to the other problem was implemented.

```
str: _ expected: dJdÃ«KgwALkbgÃ±Ã¶tÃ¡PmibwIpÃ¯ti
*** Failed! Assertion failed (after 734 tests)
```

Figure 6.14: An example of another *String* problem in the *memory allocation bug*

7 Related Work

7.1 RQG

RQG [22] (Random Query Generator) is a tool that can generate random queries based on a grammar file specified by the user or use the predefined test grammars. The generator can generate queries in MySQL databases, or any Perl DBI, JDBC or ODBC-compatible SQL databases such as JavaDB and PostgreSQL. The generated queries are executed by different databases or the same database with different settings. Differences in the output of the database(s) are then compared to detect potential bugs. It also supports 3-way database comparison of MySQL, JavaDB and PostgreSQL, which can compare the output of the three databases at the same time.

RQG's testing approach is similar to our testing since they use random values in their tables and queries. A difference is that RQG mainly focuses on generating random queries and comparing different databases or differences between releases, while our approach is focused on testing bindings for a single database and asserting the expected value. However, we also used differential testing to verify *Double* values for two bindings and it was very useful in order to decide the correctness.

7.2 SQLsmith

SQLsmith [23] is a random SQL query generator that is based on the testing tool Csmith. It focuses on testing the PostgreSQL database, but also supports SQLite. It is possible to use other databases but the user has to implement C++ classes for schema information about the database. SQLsmith uses fuzz testing (checks for crashes) and benchmarking to test databases. During its development, the tool has found 30 bugs in PostgreSQL.

SQLsmith is a great tool for fuzz testing PostgreSQL but does not check the correctness of answers.

7.3 Mysqltest

Mysqltest [24] is a tool that runs test cases on a MySQL or MariaDB database and is able to compare the output with a result file. It supports connection to one or more MySQL servers that it can switch between. To run mysqltest, a test file needs to be written in a special test language that specifies which SQL statements that should be run.

Mysqltest is somewhat similar to our testing approach, since it uses predefined SQL statements. The inputs to mysqltest are also static, while our testing uses randomly generated inputs, which mean that we can test more values of each data type. It also becomes harder to perform more tests since it is more or less a Unit test tool for MySQL, while our approach can use the testing tool QuickCheck to generate a large number of different tests easily. The number of databases is also limited in mysqltest since it can only be used with MySQL and MariaDB. Our testing method can be used with any other databases, but a few modifications are needed in order to test another database. For example, the types used by the database need to be defined, which is described in Future Work in section 8.3.

7.4 Csmith

Csmith [25] is a random test case generator for C compilers that uses differential testing to find bugs. This is necessary because the widely used verification tools need specifications to find bugs, which can be impractical for bugs regarding, for instance, inadequate optimization safety, unsound static analyses and flawed transformations. Csmith differs from other similar testing tools in the generation stage by expressing more of the C language features and use more complex C programming code. When a random C program is generated, it will use several compilers to compile the C program, run the programming code and finally compare the outputs. The tool has been very successful. It has found more than 325 new compiler bugs for the first three years.

Csmith's approach is somewhat similar to our differential SQL testing but it is applied in another area. The approach is shown to have good potential to find bugs.

8 Discussion

The testing of the SQLite bindings only involves the SQL statements *create*, *insert* and *select*. In order to evaluate whether it was beneficial to use more SQL statements, operators and subclauses, some of them such as *delete, update, or* and *join* were used for testing of two of the bindings (*HDBC-sqlite3* and *sqlite*). The result was that it did not find any more bugs. We do not think it is necessary to test them on the other bindings, since bindings can mostly be affected by bugs involving type conversations or memory allocation problems, and these can be found by using *create, insert* and *select* statements. Type conversions problems are only dependent on the data types and specific values that are converted between Haskell and SQLite. This means that by using *insert* we test that the value can be converted correctly from Haskell to SQLite and *select* tests the opposite direction. Any mismatches in the inserted value and the retrieved value will indicate a problem in one of the type conversion paths. Therefore, we chose not to test other statements.

In order to extend our testing approach, we also implemented testing that used random sequences of the queries: *insert*, *select* and *delete*, and was otherwise the same as earlier. The effects of every random query are difficult to specify, as it requires modelling the semantics of SQL. Therefore, differential testing was used to compare the outputs from three bindings. However, it did not find any more bugs and it was decided to not continue to test the other bindings with this testing technique.

8.1 Problems that were not Bugs

Initially, the *HDBC-sqlite3* binding seemed to have many problems. One problem was that the *HDBC-sqlite3* binding seemed to return incorrect *ByteString* values compared to the inserted values. However, in the *HDBC* documentation [26], it is stated that the returned *ByteStrings* contain *Text* encoded using the *UTF-8* encoding, and to extract the *Text*, the function *fromSql* should be used (it converts an SQL value to a Haskell value). The problem disappeared after the property was modified to use *fromSql* instead of using the value directly.

8.2 Other Tools Applicability on our Testing Approach

The mysqltest tool would be infeasible to test binding conversions. The reason is that mysqltest uses static inputs for bindings and since many values should be tested for each data type, makes writing static tests for all values are inefficient

The query generation of RQG and SQLsmith supports more SQL queries than our approach. It would be possible to use RQG's and SQLsmith's random query generators by printing the queries to the STDOUT (standard output) that Haskell can read to create SQL queries in our testing approach.

8.3 Future Work

The testing approach could be applied to other databases than SQLite. Examples of databases that could use this approach are MySQL, Oracle and PostgreSQL. To test another database, the main area that has to be modified is the handling of database types. For instance, MySQL has more data types representing *Integers* than SQLite. There are also other types that SQLite does not use but other databases have that should be added to the property, such as *Time* and *Date*. This means that new generators and column(s) need to be defined for the generation of the SQL queries.

9 Conclusion

Seven different SQLite bindings have been tested and four bugs have been found. The bugs are from three different bindings (*HDBC-sqlite3*, *simplest-sqlite* and *sqlite*), and there are three different data types (*String*, *Double* and *ByteString*) involved in the bugs. The bugs have been found by using QuickCheck with the two created properties *prop_bind_string* and *prop_bind_value*.

The first bug is a conversion bug that occurred (in the *sqlite* binding) since a UTF-8 *String* had not been converted to Unicode code points, which Haskell's *Char* type uses. The second bug appeared since a few *Double* values were incorrectly retrieved after being inserted as *Strings*. The reason was that the binding (*HDBC-sqlite3*) truncates the *Double* values after the first 15 significant digits. The third bug involves a null-termination problem of the type *ByteString* (in the *simplest-sqlite* binding) and it returns a shorter *ByteString* if *\NUL* is included in the inserted *ByteString*. It happens since the binding uses the function *packCString* that cuts the *ByteString* at *\NUL*. The last bug is a memory allocation problem and it occurs since the binding (*simplest-sqlite*) did not allocate the correct amount of memory in the function *sqlite3_bind_text*.

We have realized that non-ASCII characters and values are needed to find type conversion bugs. It is necessary to use a testing approach that executes multiple tests. The tests of the bindings can use a predefined sequence of queries (it means that the correct answer easily can be detected) since type conversion errors appear with specific values and they have to be stored and retrieved from the binding, which can be done in a few SQL statements. It means that more advanced statements are not likely to increase the probability of finding more errors in the bindings; it rather tests the database itself. However, it is important to test the bindings since it is the connection between the library and the preferred programming language.

The testing approach presented in this thesis is an efficient way of finding bugs in bindings, and we sincerely recommend people to use it. It is user-friendly and includes tests of the seven most downloaded SQLite bindings the last 30 days from Hackage (as of September 2016). It also has many possibilities such as extension with other databases. It is free and accessible at our Github repository [27].

Bibliography

[1] W. M. McKeeman, “Differential Testing for Software,” *Digital Technical Journal,* vol. 10, no. 1, pp. 100-107, 1998.

[2] G. Inc, “The sqlite package,” Hackage, 21 April 2008. [Online]. Available: http://hackage.haskell.org/package/sqlite. [Accessed 7 March 2016].

[3] B. O'Sullivan, “Interfacing with C: the FFI,” in *Real World Haskell*, Sebastopol, o'Reilly Media, Inc., 2008, pp. 405-428.

[4] P. Hamill, Unit Test Frameworks: Tools for High-Quality Software Development, Sebastopol: O'Reilly Media, Inc., 2005.

[5] J. H. Koen Claessen, “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs,” *ICFP ’00. ACM,* p. 268–279, 2000.

[6] Hackage, “Hackage,” Hackage, [Online]. Available: https://hackage.haskell.org/. [Accessed 7 March 2016].

[7] SQLite, “How SQLite Is Tested,” SQLite, 2016. [Online]. Available: https://www.sqlite.org/testing.html. [Accessed 10 february 2016].

[8] MySQL, “What to Do If MySQL Keeps Crashing,” Oracle Corporation, 2016. [Online]. Available: http://dev.mysql.com/doc/refman/5.7/en/crashing.html. [Accessed 10 february 2016].

[9] S. Haldar, “Chapter 2. SQLite Overview,” in *SQLite Database System Design and Implementation*, Sunnyvale, Motorola mobility, Inc. , 2015, pp. 35-70.

[10] H.-S. J. T. W. Jerry Zeyu Gao, “Black box testing methods for software components,” in *Testing and Quality Assurance for Component-based Software*, Norwood, Artech House, Inc., 2003, pp. 119-140.

[11] K. A. Saleh, “Black box, white box, and grey box testing,” in *Software Engineering*, Florida, J.Ross Publishing, Inc., 2009, pp. 224-241.

[12] A. S. Mena, “Randomized Testing with QuickCheck,” in *Beginning Haskell: A Project-Based Approach*, New York, Apress, 2014, pp. 364-371.

[13] B. O'Sullivan, “Testing and Quality Assurance,” in *Real World Haskell*, Sebastopol, o'Reilly Media, Inc., 2008, pp. 255- 265.

[14] M. O. Grant Allen, “SQL for SQLite,” in *The Definitive Guide to SQLite, Second Edition*, New York, Springer Science+Business Media, LLC., 2010, pp. 47-86.

[15] M. Snoyman, “The persistent-sqlite package,” Hackage, 22 June 2010. [Online]. Available: http://hackage.haskell.org/package/persistent-sqlite. [Accessed 7 March 2016].

[16] L. P. S. J. H. Bryan O'Sullivan, “The sqlite-simple package,” Hackage, 15 August 2012. [Online]. Available: http://hackage.haskell.org/package/sqlite-simple. [Accessed 7 March 2016].

[17] D. Knapp, “The direct-sqlite package,” Hackage, 13 March 2010. [Online]. Available: http://hackage.haskell.org/package/direct-sqlite. [Accessed 7 March 2016].

[18] B. Lykah, “The groundhog-sqlite package,” Hackage, 16 June 2011. [Online]. Available: http://hackage.haskell.org/package/groundhog-sqlite. [Accessed 7 March 2016].

[19] J. Goerzen, “The HDBC-sqlite3 package,” Hackage, 24 September 2006. [Online].

Available: http://hackage.haskell.org/package/HDBC-sqlite3. [Accessed 7 March 2016].

[20] Y. Jujo, “The simplest-sqlite package,” Hackage, 20 November 2015. [Online]. Available: http://hackage.haskell.org/package/simplest-sqlite. [Accessed 7 March 2016].

[21] J. Goerzen, “The HDBC package,” Hackage, 24 September 2006. [Online]. Available: https://hackage.haskell.org/package/HDBC. [Accessed 7 March 2016].

[22] P. Stoev, “Random Query Generator,” Oracle, 17 July 2012. [Online]. Available: https://github.com/RQG/RQG-Documentation/wiki/RandomQueryGenerator. [Accessed 4 February 2016].

[23] A. Seltenreich, “SQLsmith,” Andreas Seltenreich, 26 April 2015. [Online]. Available: https://github.com/anse1/sqlsmith/releases. [Accessed 5 February 2016].

[24] Oracle, “mysqltest — Program to Run Test Cases,” Oracle, 2016. [Online]. Available: https://dev.mysql.com/doc/mysqltest/2.0/en/mysqltest.html. [Accessed 9 February 2016].

[25] Y. C. E. E. J. R. Xuejun Yang, “Finding and Understanding Bugs in C Compilers,” in *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Jose, ACM, 2011, pp. 283-294.

[26] J. Goerzen, “Database.HDBC,” John Goerzen, 2011. [Online]. Available: https://hackage.haskell.org/package/HDBC-2.4.0.1/docs/Database-HDBC.html. [Accessed 20 April 2016].

[27] V. E. Jimmy Svensson, “QuickChecking_foreign_languages,” GitHub, [Online]. Available: https://github.com/MasterThesis-JimmyVictor/QuickChecking_foreign_languages/tree/master/SQLite. [Accessed 30 January 2017].