dougalstanton.net

# Looking Out To Sea - Testing embedded C++ with Haskell QuickCheck

If you work with C or C++ and are hankering after the testing power of QuickCheck for property tests, this is your lucky day.

If you've got C++ code you have to bear a couple of things in mind. If it's vanilla C then you're okay.

- The code under test needs to be C-compatible, so using `extern C` otherwise the name mangler will prevent your linker from finding the correct symbols.
- You have to manage the compilation steps manually or using an ordinary Makefile. The CABAL system doesn't recognise `.cpp` files so won't process them properly on its own.

These are not grave restrictions by any means. This is how it's done. First, we will see the files we have to start with. These are the ones which we have to write and cannot be autogenerated:

- foo.cpp
- foo.h
- Foo.hsc
- Main.hs

The first two comprise your "system under test". The next two are the Haskell/C bridge file which handles marshalling and the Haskell test executable. As your system gets more complex you might obviously add to these.

I will work through this list of files in order, showing what they contain and how they can be used to generate further dependencies until the test system is complete.

## Some code to test

First we have the C++ code we're testing. I'm going to use the example of data transfer over a network. We’ve got some data structure which we need to transmit or receive. A simple way of testing whether we are internally consistent is making sure we can always decode the same data we encoded. This is the “round trip” test.

$$\mathtt{id} = \mathtt{deserialise} \cdot \mathtt{serialise}$$

So `foo.cpp` contains a simple struct and a pair of functions that should operate as a “round trip”, serialising and then deserialising the input.

```
 #include "foo.h"

size_t serialise (const struct foo * in, uint8_t * out)
{
    out[0] = in->bar & 0x000000ff;
    out[1] = (in->bar & 0x0000ff00) >> 8;
    out[2] = (in->bar & 0x00ff0000) >> 16;
    out[3] = (in->bar & 0xff000000) >> 24;
    out[4] = in->baz;
    out[5] = in->quux[0];
    out[6] = in->quux[1];
    out[7] = in->quux[2];
    out[8] = in->quux[3];
    return 9; // bytes written
}

size_t deserialise (const uint8_t * in, struct foo * out)
{
    out->bar = in[0] | in[1] << 8 | in[2] << 16 | in[3] << 24;
    out->baz = in[4];
    out->quux[0] = in[5];
    out->quux[1] = in[6];
    out->quux[2] = in[6];
    out->quux[3] = in[8];
    return 9; // bytes recovered
}
```

I’ve hard-coded lots of stuff there, all the indices and the bytes read and written. I’ve also put an obvious bug into the indices to show the checker working. The header file is also important, because it defines what the interface is to the Haskell side (or any other code).

```
 #include <stdint.h>
#include <string.h>
```

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct foo
{
    uint32_t bar;
    int8_t   baz;
    int8_t   quux[4];
} foo_t;

size_t serialise   (const struct foo *, uint8_t *);
size_t deserialise (const uint8_t *, struct foo *);

#ifdef __cplusplus
}
#endif
```

In particular the header file should define the structure if we want to tell Haskell how it works.

As I said before, you need to ensure the `extern "C"` stuff is there otherwise you'll fail at the linking stage. This means that polymorphic C++ code won't work directly — you'll have to write a C wrapper and attach to that.

To run Haskell tests on C++ code we need the C++ object file. I'm using the filename `foo.cpp.o` instead of `foo.o` here to highlight an important point. If you're compiling all this in the same directory as the Haskell code, and you've got a `foo.cpp` and a `Foo.hs` then GHC will likely clobber the object file you've already written — it doesn't pay attention to difference in case. So for simplicity's sake, either rename the Haskell source file so it's not the same or rename the C/C++ object file so it's not the same.

```
 $ gcc -o foo.cpp.o -c foo.cpp
```

If you forgot to tell the compiler that these are C-compatible functions with the `extern` keyword then the output will look like this:

```
 $ nm foo.cpp.o
0000000000000148 s EH_frame0
00000000000000b0 T __Z11deserialisePKhP3foo
0000000000000190 S __Z11deserialisePKhP3foo.eh
```

```
0000000000000000 T __Z9serialisePK3fooPh
0000000000000160 S __Z9serialisePK3fooPh.eh
```

whereas we want them to look more like this:

```
 $ nm foo.cpp.o
0000000000000148 s EH_frame0
00000000000000b0 T _deserialise
0000000000000190 S _deserialise.eh
0000000000000000 T _serialise
0000000000000160 S _serialise.eh
```

## Bridging the gap

Next we create a bridge between the two different languages. This is not difficult but can be a bit fiddly. I'm using Hsc2Hs here which is the simplest to understand but requires a bit of boilerplate. Essentially it's a Haskell file with C-style interjections. The `hsc2hs` will process the C-like stuff and replace it with extra Haskell.

```
 {-# LANGUAGE ForeignFunctionInterface #-}
module Foo where

#include "foo.h"

import Foreign hiding (unsafePerformIO)
import Foreign.Marshal.Array
import Foreign.Storable
import System.IO.Unsafe (unsafePerformIO)

data Foo = Foo
    { fooBar  :: #{type uint32_t}
    , fooBaz  :: #{type int8_t}
    , fooQuux :: [#{type int8_t}]
    } deriving (Eq, Show)

instance Storable Foo where
    sizeOf _    = #{size foo_t}
    alignment _ = alignment (undefined :: Word32)

    peek ptr = do
        r1 <- #{peek foo_t, bar} ptr
        r2 <- #{peek foo_t, baz} ptr
        r3 <- peekArray 4 $ #{ptr foo_t, quux} ptr
        return (Foo r1 r2 r3)
    poke ptr (Foo r1 r2 r3) = do
        #{poke foo_t, bar} ptr r1
        #{poke foo_t, baz} ptr r2
        pokeArray (#{ptr foo_t, quux} ptr) r3

type Buffer = [Word8]

foreign import ccall "foo.h serialise"
    c_serialise :: Ptr Foo -> Ptr Word8 -> IO Int
foreign import ccall "foo.h deserialise"
    c_deserialise :: Ptr Word8 -> Ptr Foo -> IO Int

serialise :: Foo -> Buffer
serialise foo = unsafePerformIO $ with foo $
    \foo_ptr -> allocaArray (sizeOf foo) $
    \buf_ptr -> do
        sz <- c_serialise foo_ptr buf_ptr
        peekArray sz buf_ptr

deserialise :: Buffer -> Foo
deserialise buf = unsafePerformIO $ withArray buf $
    \buf_ptr -> allocaBytes (sizeOf (undefined :: Foo)) $
    \foo_ptr -> do
        _ <- c_deserialise buf_ptr foo_ptr
        peek foo_ptr
```

There's a bit of personal choice as to how you interpret some of the datatypes in Haskell. You can consider a boolean value as `Bool` or as `Word8` ; one more closely models the data but the other more closely models the semantics. In this case I use the `#{type}` directive which models the C type completely — the preprocessor can't know that it's a boolean being represented. I think this is better if you're testing C/C++ code because it allows you to pass any valid number rather than only emitting `0` or `1` . If you want to model the semantics there are some utility functions like `toBool` and `fromBool` which will convert the data properly.

See also Magnus Therning's informative blog post about Hsc2Hs. We compile like so:

```
$ hsc2hs Foo.hsc
```

Now we have a C object file created by GCC and a Haskell source file created by Hsc2Hs.

- foo.cpp.o

- Foo.hs

If the marshalling in `Foo.hsc` is defined correctly you should be able to load the generated Haskell file into GHCi alongside the C++ object file to exercise it.

```
$ ghci Foo.hs foo.cpp.o
```

## Some code to test with

All that's needed now is your tests. If the code you are testing does not maintain state then all the better: you can wrap the FFI calls in `unsafePerformIO` to convince the type system that they're pure computations. The Haskell compiler doesn't know the external code is pure so this is your promise to the type system that it *is* pure. If you are in fact lying (or in error) then you may find odd behaviour or crashes.

```
module Main where

import Foo

import Test.QuickCheck

instance Arbitrary Foo where
    arbitrary = do
        foobar  <- arbitrary
        foobaz  <- arbitrary
        fooquux <- vector 4
        return (Foo foobar foobaz fooquux)

prop_roundtrip = property $
    \foo -> foo == deserialise (serialise foo)

main = quickCheck prop_roundtrip
```

Auto-generating tests is much easier now that you've dragged the code over from the dark side. Remember to include your precompiled object file on the command line when you're building your Haskell code.

```
$ ghc --make Main.hs foo.cpp.o -o tests
```

Even better of course is to write a Makefile which will do everything like that for you:

```
tests: Main.hs Foo.hs foo.cpp.o
```

```
        ghc --make Main foo.cpp.o -o tests

foo.cpp.o: foo.cpp foo.h
        gcc -c foo.cpp -o foo.cpp.o

Foo.hs: Foo.hsc foo.h
        hsc2hs -I. Foo.hsc

clean:
        rm -rf *.hi *.o tests Foo.hs
```

Let's see how things fare when we attempt to check our roundtrip property.

```
 $ make
hsc2hs -I. Foo.hsc
gcc -c foo.cpp -o foo.cpp.o
ghc --make Main foo.cpp.o -o tests
[1 of 2] Compiling Foo              ( Foo.hs, Foo.o )
[2 of 2] Compiling Main             ( Main.hs, Main.o )
Linking tests ...

$ ./tests
Falsifiable (after 1 test):
Foo {fooBar = 1, fooBaz = -1, fooQuux = [-1,-1,0,0]}
```

Oh dear! Not a great outcome there… But now you can load up GHCi and manually run that test to see what result you get when serialising and deserialising, examining the intermediate data and ultimately tracking down the bug.