Magnus web site
Random stuff

Top Archive

Using QuickCheck to test C APIs

2015-06-15 - Magnus Therning

Last year at ICFP I attended the tutorial on QuickCheck with John Hughes. We got to use the Erlang implementation of QuickCheck to test a C API. Ever since I've been planning to do the same thing using Haskell. I've put it off for the better part of a year now, but then Francesco Mazzoli wrote about inline-c (Call C functions from Haskell without bindings and I found the motivation to actually start writing some code.

The general idea

Many C APIs are rather stateful beasts so to test it I

1. generate a sequence of API calls (a program of sorts),
2. run the sequence against a model,
3. run the sequence against the real implementation, and
4. compare the model against the real state each step of the way.

The C API

To begin with I hacked up a simple implementation of a stack in C. The "specification" is

```
/**
 * Create a stack.
 */
void *create();

/**
 * Push a value onto an existing stack.
 */
void push (void *, int);

/**
 * Pop a value off an existing stack.
 */
int pop(void *);
```

Using `inline-c` to create bindings for it is amazingly simple:

```
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}
```

```
module CApi
    where

import qualified Language.C.Inline as C
import Foreign.Ptr

C.include "stack.h"

create :: IO (Ptr ())
create = [C.exp| void * { create() } |]

push :: Ptr () -> C.CInt -> IO ()
push s i = [C.exp| void { push($(void *s), $(int i)) } |]

pop :: Ptr () -> IO C.CInt
pop s = [C.exp| int { pop($(void *s)) } |]
```

In the code below I import this module qualified.

Representing a program

To represent a sequence of calls I first used a custom type, but later realised that there really was no reason at all to not use a wrapped list:

```
newtype Program a = P [a]
    deriving (Eq, Foldable, Functor, Show, Traversable)
```

Then each of the C API functions can be represented with

```
data Statement = Create | Push Int | Pop
    deriving (Eq, Show)
```

`Arbitrary` for `Statement`

My implementation of `Arbitrary` for `Statement` is very simple:

```
instance Arbitrary Statement where
    arbitrary = oneof [return Create, return Pop, liftM Push ar
    shrink (Push i) = Push <$> shrink i
    shrink _ = []
```

That is, `arbitrary` just returns one of the constructors of `Statement`, and shrinking only returns anything for the one constructor that takes an argument, `Push`.

Prerequisites of `Arbitrary` for `Program Statement`

I want to ensure that all `Program Statement` are valid, which means I need to define the model for running the program and functions for checking the precondition of a statement as well as for updating the model (i.e. for running the `Statement`).

Based on the C API above it seems necessary to track creation, the contents of the stack, and even if it isn't explicitly mentioned it's probably a good idea to track the popped value. Using `record` (`Record` is imported as `R`, and `Record.Lens` as `RL`) I defined it like this:

```
type ModelContext = [R.r| { created :: Bool, pop :: Maybe Int,
```

Based on the rather informal specification I coded the pre-conditions for the three statements as

```
preCond :: ModelContext -> Statement -> Bool
preCond ctx Create = not $ RL.view [R.l| created |] ctx
preCond ctx (Push _) = RL.view [R.l| created |] ctx
preCond ctx Pop = RL.view [R.l| created |] ctx
```

That is

- `Create` requires that the stack hasn't been created already.
- `Push i` requires that the stack has been created.
- `Pop` also requires that the stack has been created.

Furthermore the "specification" suggests the following definition of a function for running a statement:

```
modelRunStatement :: ModelContext -> Statement -> ModelContext
modelRunStatement ctx Create = RL.set [R.l| created |] True ctx
modelRunStatement ctx (Push i) = RL.over [R.l| stack |] (i :) c
modelRunStatement ctx Pop = [R.r| { created = c, pop = headMay
    where
        c = RL.view [R.l| created |] ctx
        s = RL.view [R.l| stack |] ctx
```

(This definition assumes that the model satisfies the pre-conditions, as can be seen in the use of `tail`.)

`Arbitrary` for `Program Statement`

With this in place I can define `Arbitrary` for `Program Statement` as follows.

```
instance Arbitrary (Program Statement) where
    arbitrary = liftM P $ ar baseModelCtx
        where
            ar m = do
                push <- liftM Push arbitrary
                let possible = filter (preCond m) [Create, Pop,
                if null possible
                    then return []
                    else do
                        s <- oneof (map return possible)
                        let m' = modelRunStatement m s
                        frequency [(499, liftM2 (:) (return s)
```

The idea is to, in each step, choose a valid statement given the provided model and cons it with the result of a recursive call with an updated model. The constant 499 is just an arbitrary one I chose after running `arbitrary` a few times to see how long the generated programs were.

For shrinking I take advantage of the already existing implementation for lists:

```
    shrink (P p) = filter allowed $ map P (shrink p)
        where
            allowed = and . snd . mapAccumL go baseModelCtx
                where
                    go ctx s = (modelRunStatement ctx s, preCon
```

Some thoughts so far

I would love making an implementation of `Arbitrary s`, where `s` is something that implements a type class that contains `preCond`, `modelRunStatement` and anything else needed. I made an attempt using something like

```
class S a where
    type Ctx a :: *

    baseCtx :: Ctx a
    preCond :: Ctx a -> a -> Bool
    ...
```

However, when trying to use `baseCtx` in an implementation of `arbitrary` I ran into the issue of injectivity. I'm still not entirely sure what that means, or if there is something I can do to work around it. Hopefully someone reading this can offer a solution.

Running the C code

When running the sequence of `Statement` against the C code I catch the results in

```
type RealContext = [r| { o :: Ptr (), pop :: Maybe Int } |]
```

Actually running a statement and capturing the output in a `RealContext` is easily done using `inline-c` and `record`:

```
realRunStatement :: RealContext -> Statement -> IO RealContext
realRunStatement ctx Create = CApi.create >>= \ ptr -> return $
realRunStatement ctx (Push i) = CApi.push o (toEnum i) >> retur
    where
        o = RL.view [R.l| o |] ctx
realRunStatement ctx Pop = CApi.pop o >>= \ v -> return $ RL.se
    where
        o = RL.view [R.l| o |] ctx
```

Comparing states

Comparing a `ModelContext` and a `RealContext` is easily done:

```
compCtx :: ModelContext -> RealContext -> Bool
compCtx mc rc = mcC == rcC && mcP == rcP
    where
        mcC = RL.view [R.l| created |] mc
        rcC = RL.view [R.l| o |] rc /= nullPtr
        mcP = RL.view [R.l| pop|] mc
        rcP = RL.view [R.l| pop|] rc
```

Verifying a `Program Statement`

With all that in place I can finally write a function for checking the validity of a program:

```
validProgram :: Program Statement -> IO Bool
validProgram p = and <$> snd <$> mapAccumM go (baseModelCtx, ba
    where
        runSingleStatement mc rc s = realRunStatement rc s >>=

        go (mc, rc) s = do
            ctxs@(mc', rc') <- runSingleStatement mc rc s
            return (ctxs, compCtx mc' rc')
```

(This uses `mapAccumM` from an earlier post of mine.)

The property, finally!

To wrap this all up I then define the property

```
prop_program :: Program Statement -> Property
prop_program p = monadicIO $ run (validProgram p) >>= assert
```

and a main function

```
main :: IO ()
main = quickCheck prop_program
```

Edit 2015-07-17: Adjusted the description of the pre-conditions to match the code.

Tags: haskell, quickcheck, testing

⟸ mapAccum in monad Systemd watchdog ⟹

Jesse McDonald

2015-06-15

I believe the problem with `baseCtx` is that there could be two or more instances of `S` which have the same definition for `Ctx a`, which means that the compiler can't infer `a` (and thus the instance to use) from `Ctx a`. One option would be to add a `Proxy a` parameter to `baseCtx` to select the instance. Another would be to make `S` a multi-parameter type class with a one-to-one functional dependency between the parameters, like so:

```
class S a ctx | a -> ctx, ctx -> a where
    baseCtx :: ctx
    ...
```

Magnus Therning

2015-06-16

I should probably point out that I'm using version 0.3.1.2 of `record`.

The library seems to have seen dramatic changes as of version 0.4.0.0.

Leave a comment