State Machine Testing in Haskell using Hedgehog library

Jan Hrček

29th April 2021

Overview

- Simple example REST api we'll be testing
- Property testing
  - QuickCheck vs. hedgehog
- State machine testing (SMT)
- Hedgehog SMT prerequisites
  - higher kinded polymorphism
- Example code + live coding SMT tests

Example REST api

- Our system under test
- A simple servant based application with 3 endpoints
  1. GET /projects - get list of projects
  2. POST /projects - create a project (returning ID of create project)
  3. DELETE /projects/PROJECT_ID - delete a project
- see `src/App` in this repo

Property testing - key concepts

- Generate **random inputs**
- Assert that function under tests satisfies some property for all generated inputs
- When input is found that doesn't satisfy the property, try to **shrink** it to provide minimal counterexample

Property testing

Advantages (vs. unit testing)

- better coverage with fewer tests written
- better at discovering edge cases/unexpected feature interactions
- forces you to think more abstractly about desired system properties

Disadvantages

- not always easy to come up with useful properties
- need to control the distribution of generated random values

Property testing in Haskell

- Two most popular libraries QuickCheck and Hedgehog
- QuickCheck is the original property testing library by John Hughes
  - state machine testing provided by separate library quickcheck-state-machine
- Hedgehog is more recent
  - integrated skrinking
  - beautiful output with color highlighted code
  - state machine testing support as part of base library

State machine testing (SMT)

- A way to apply property testing to impure code / external system
- The test maintains a simplified model of the state of the tested system
  - e.g. list of users, projects created
- The test generates random sequence of actions that can depend on state
- The test executes generated actions one by one
- The test verifies after each action, that various post-conditions are met
  - e.g. HTTP api returns 200 status reponse
  - If I rename resource to X, it's named X when I get it the next time

State machine testing - Key Requirements

1. action generation should be aware of the system state
   - e.g. generate action to delete resource only if it exists
2. random input generation must be deterministic
   - reproducibility if test fails!

   ```
   This failure can be reproduced by running:
   > recheck (Size 2) (Seed 184616 79431) prop_api_tests
   ```
3. sequences of actions must be shrinkable
   - goal: given long sequence of actions reproducing bug, find a shorter sequence that still reproduces the bug

State machine testing - Key Requirements

- How can we generate **random** sequence of actions that depend on each other (e.g. output of one action is used as input for next action) yet is still **deterministic**??
- Ingenious idea:
  - separate phases of **random input generation** and **action execution**
  - During test generation dependencies between actions are expressed using symbolic variables
  - During test execution symbolic variables are replaced by actual values returned by the tested system

Example input actions generated by the test

```
Var 0 = DeleteNonExistentProjectInput []
Var 1 = CreateProjectInput "Xg"
Var 2 = DeleteExistigProjectInput (Var 1)
Var 3 = GetProjectsInput
Var 4 = CreateProjectInput "GI"
Var 5 = DeleteExistigProjectInput (Var 4)
Var 6 = CreateProjectInput "GA6FX"
Var 7 = CreateProjectFailNameExistsInput "GA6FX"
Var 8 = DeleteNonExistentPr9jectInput [ Var 6 ]
```

Aside: Higher kinded polymorphism

- concrete types
  - no type variables
  - `Maybe Int`, `[String]`, `(Double, Bool)`
- "normal" polymorphic types
  - type variables have kind `Type`
  - `Maybe a`, `[b]`, `(c,d)`
  - intuition: container that can hold different types of things
- higher kinded polymorphism
  - type variables have more complex kinds
  - `f` in `f Int` has kind `Type -> Type`
  - familiar example: `Functor`, `Applicative`, `Monad` are all parametrized by type variable of kind `Type -> Type`
  - intuition: functions that can work with different types of containers (think `fmap`, `traverse`)

Higher kinded polymorphism

- The main point: you can have a datatype which is parametrized by a higher kinded type (like a container)

```
> data Something (container :: Type -> Type)
    = Something (container Int)
> :t Something (Just 1)
Something (Just 1) :: Something Maybe
> :t Something [1]
Something [1] :: Something []
```

What is that `Var` thing?

- `newtype Var a v = Var (v a)`
  - *Variables are the potential or actual result of executing an action.*
  - *They are parameterised by either `Symbolic` or `Concrete` depending on the phase of the test.*
- `Var ProjectId Symbolic`
  - is symbolic representation of `ProjectId` used during test input generation
  - we don't yet know what `ProjectId` the application will return when the action is executed
  - but we need something to be able to create dependencies between actions
- `Var ProjectId Concrete`
  - is concrete representation of `ProjectId` used during test execution
  - can get actual `ProjectId` value out of it (e.g. to call `deleteProject projId`)

Vars are used in State and Input

- You declare your custom `data State` to represent the state of tested system
- For each action you'd like to execute you declare `(Action)Input`
  - represents randomly generated input data
  - used as input for actions that are executed against tested application
- Both `State` and `Input` types have to be parametrized by `v :: Type -> Type`
- Both `State` and `Input` can contain pure data that we generated during tests, as well as references to values returned by external system.
- in our example app
  - When you `createProject` it returns `ProjectId`
  - If you want to store this value in `State` you must wrap it in `Var ProjectId v`

Skeleton state machine test

```haskell
prop_api_tests :: Property
prop_api_tests = property $ do
    let commands =
          [ command1
          , command2
          , ...
          ]
    -- Generate random sequence of 1 - 100 actions
    -- starting from initialState
    actions <- forAll $ Gen.sequential
        (Range.linear 1 100) initialState commands
    executeSequential initialState actions
```

Command - overview

- the key abstraction when writing state machine tests
- one command represents one action that hedgehog can try to run against the tested system

Command skeleton

```haskell
someCommand :: Command gen test state
someCommand = Command
    commandGen       -- 1. when/how to generate inputs?
    commandExecute   -- 2. how to execute input?
    [ Require (...)  -- 3. preconditions
    , Update (...)   -- 4. how to update the model?
    , Ensure (...)   -- 5. assertions - what must be true
                     -- after command executed?
    ]
```

Command parts (1) - Input generator

- `inputGenerator :: state Symbolic -> Maybe (gen (input Symbolic))`
- We're given current state (with Sybmolic vars) and we have to decide:
  - does it make sense to generate the input for this command in this state?
  - if not, we return Nothing
  - if yes, we return Just a generator (that can pick some values in the state)
- example: to generate "Delete project" command we need at leat 1+ project in the state
- the generator can pick random project from the state and return `DeleteProjectInput projectIdVar`

Command parts (2) - execution function

```
commandExecute :: input Concrete -> m output
```

- Given concrete input, execute it (against the tested out) and get the output
- The output can be used to change our model (e.g. add new `ProjectId` to the model)

Command parts (3) - Require callback

```
state Symbolic -> input Symbolic -> Bool
```

- Given this state, does it still make sense to execute this input?
- Used when shrinking
  - after we remove some actions from the sequence, followup actions might no longer make sense

Command parts (4) - Update callback

```
forall v. Ord1 v => state v -> input v -> Var output
v -> state v
```

- How to update the state after the command is executed
- this is perhaps the hardest one to crack, because it must be polymorphic in the `v` argument - i.e. it must work the same for `Symbolic` as well as for `Concrete` variables

Command parts (5) - Ensure callback

```
state Concrete -> state Concrete -> input Concrete ->
output -> Test ()
```

- After each action is executed we're given access to
  - state before the action
  - state after the action
  - input for the action
  - output of the action
- we can do whatever assertions using these values

Useful materials

- Introduction to state machine testing (3 part series from QFPL)
  - Part 1
  - Part 2
  - Part 3)
- Most comprehensive single talk on property testing John Hughes - Building on developers' intuitions
- Good intro to Higher Kinded Data: Chris Penner - Higher Kinded Data Types By Example