Testing REST APIs with QuickCheck

Matthias Fischmann (mf@zerobuzz.net)

zerobuzz, Berlin

Berlin Haskell Users Group / 2014-Feb-06 / hosted by Zanox.

The source

`https://github.com/zerobuzz/webtest.git`

Work in progress for both HTTP and UI testing (selenium / webdriver).

This talk is about HTTP only.

Data types: (application code)

```
data Value = String Text
           | Number Double
           | Object (Map Text Value)
           ...
```

Functions: (productive code)

```
encode :: Value → SBS
decode :: SBS → Maybe Value
```

Properties: (test code)

```
prop_EncodeDecode :: Value → Property
prop_EncodeDecode v = decode (encode v) ≡ Just v
```

These are your (less exotic) options to enhance confidence in:

- formal proofs: expensive or infeasable
- brute-force test of all input data: impractical
- unit testing: pick "relevant" input values by hand (often has significant gaps)
- quickcheck: **randomized** test to approximate brute-force tests

quickcheck does two wonderful things:

- It generates input data for approxmiating brute-force search (**arbitrary**); and it
- If the property fails, it finds a smallest counter-example (**shrink**).

Example

```
*Main> quickCheck prop_EncodeDecode
*** Failed! Falsifiable (after 22 tests and 1 shrink):
String ""
```

Example

Oops?

Example

```
*Main> encode (String "")
"\"\""

*Main> decode it :: Maybe Value
Nothing

*Main> decode "[\"\"]" :: Maybe Value
Just (Array (fromList [String ""]))
```

Example

$$\text{prop\_EncodeDecode} :: \mathit{Value} \rightarrow \mathit{Property}$$
$$\text{prop\_EncodeDecode v} = \text{topLevel v} \Rightarrow \text{decode (encode v)} \equiv \mathit{Just}\ \text{v}$$

$$\text{topLevel} :: \mathit{Value} \rightarrow \mathit{Bool}$$
$$\text{topLevel } (\mathit{Object}\ \_) = \mathit{True}$$
$$\text{topLevel } (\mathit{Array}\ \_) = \mathit{True}$$
$$\text{topLevel } \_ = \mathit{False}$$

```
*Main> quickCheck prop_EncodeDecode
+++ OK, passed 100 tests.
```

```
*Main> :type quickCheck
quickCheck :: Arbitrary a => (a -> Property) -> IO ()

*Main> :info Arbitrary
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
```

Properties

If client misbehaves, server **does not crash (5xx responses)**.

If client "behaves well", server **does not complain (4xx, 5xx responses)**.

If client "behaves well", server **"behaves well"**.
(Example: “Resources are available for GET **at every moment between** the corresponding POST and DELETE requests.”)

The approach

The data under scrutiny is HTTP request lists (scripts).

The approach

```
data Script = ...
```

```
instance Arbitrary Script where
  arbitrary      = ...
  shrink script = ...
```

The approach

Two challenges:

- How to generate **interesting** arbitrary scripts?
- How to write **concise and expressive** properties?

The application

The server stores documents in folders.
Documents consist of titles and paragraph lists.

```
data Content = Doc Text [Ref]
             | Par Text
             | Folder [Ref]
```

References point to further resources that need to be fetched via HTTP:

```
type Ref = String
```

For instance (pseudo-code):

```
HTTP GET "/4871"     ⇒ Doc "Rainbows End"
                           ["/4871/12","/4871/18","/4871/17"]
HTTP GET "/4871/12" ⇒ Par  "..."
```

Scripts

```
data ScriptItem =
  ScriptItemHTTP
    { srqMethod      :: RequestMethod
    , srqBody        :: Either SBS Value
    , srqGetParams   :: [(SBS, SBS)]
    , srqPostParams  :: [(SBS, SBS)]
    , srqHeaders     :: [(SBS, SBS)]
    }
  deriving (Show, Eq, Typeable, Generic)
```

```
newtype Script = Script { scriptItems :: [ScriptItem] }
  deriving (Show, Eq, Typeable, Generic)
```

Instead of *Value*, aeson can handle arbitrary data types as json data transparently:

```
data Content = Doc Text [Ref]
             | Par Text
             | Folder [Ref]
  deriving (Show, Eq, Typeable, Generic)
```

```
instance ToJSON Content where
  toJSON (Doc title refs) = ...
  toJSON (Par text)       = ...
  toJSON (Folder refs)    = ...
```

```
instance FromJSON Content where
  parseJSON s = ...
```

Scripts

```
data ScriptItem app =
  ScriptItemHTTP
    { srqMethod      :: RequestMethod
    , srqBody        :: Either SBS app
    , srqGetParams   :: [(SBS, SBS)]
    , srqPostParams  :: [(SBS, SBS)]
    , srqHeaders     :: [(SBS, SBS)]
    }
  deriving (Show, Eq, Typeable, Generic)
```

```
newtype Script app = Script { scriptItems :: [ScriptItem app] }
  deriving (Show, Eq, Typeable, Generic)
```

... is something missing?

Scripts and URIs

We need to address resources via **URIs**.

"Please GET the object that I have POSTed three items earlier in this Script."

Scripts and URIs

```
data ScriptItem app =
  ScriptItemHTTP
    { srqSerial      :: Ix
    , srqMethod      :: RequestMethod
    , srqBody        :: Either SBS app
    , srqGetParams   :: [(SBS, SBS)]
    , srqPostParams  :: [(SBS, SBS)]
    , srqHeaders     :: [(SBS, SBS)]
    , srqPath        :: Either SBS IxRef
    }
  deriving (Show, Eq, Typeable, Generic)
```

```
newtype Script app = Script { scriptItems :: [ScriptItem app] }
  deriving (Show, Eq, Typeable, Generic)
```

```
type Ix = Int
```

```
data IxRef = IxRef Int | IxRefRoot
  deriving (Show, Eq, Ord, Typeable, Generic)
```

Writing unit tests with Script

webtest package provides:

```
post :: Ix → Content → IxRef → ScriptItem Content
post serial body ref =
  ScriptItemHTTP serial POST (Right body) [] [] [] (Right ref)
```

```
get :: Ix → IxRef → ScriptItem Content
get serial ref =
  ScriptItemHTTP serial GET (Left "") [] [] [] (Right ref)
```

Application tester writes:

```
script :: Script Content
script = Script $
  post 0 (Folder [])                    IxRefRoot :
  post 2 (Doc "On Ontology" []) (IxRef 0) :
  post 3 (Doc "On Nothing" [])  (IxRef 0) :
  get  5                        (IxRef 0) :
  []
```

Arbitrary Scripts

webtest package provides:

```
instance Arbitrary app ⇒ Arbitrary (Script app) where
  arbitrary      = ...
  shrink script = ...
```

Application tester writes:

```
instance Arbitrary Content where
  arbitrary = oneof
    [Doc    ⟨$⟩ arbitrary ⟨∗⟩ arbitrary
    ,Par    ⟨$⟩ arbitrary
    ,Folder ⟨$⟩ arbitrary
    ]
```

Arbitrary Scripts

```
*Main> sample' arbitrary :: IO [Content]
[Doc "xsw" [], Par "", Par "::", Folder ["/81"], ...]

*Main> sample' arbitrary :: IO [Script Content]
[Script [...], ...]

*Main> sample' arbitrary >>= mapM_ (runScript setup)
[Trace [...], ...]
```

Script execution

```
runScript :: ToJSON app ⇒ Setup → Script app → IO (Trace app)
runScript = ...
```

```
data Trace app = Trace [(ScriptItem content, Maybe (Response SBS))]
```

```
data Setup = Setup {host    :: String
                   , port    :: Int
                   , verbose :: Bool}
```

Properties of Scripts

prop_No5xx :: *Trace Content* $\rightarrow$ *Property*
prop_No5xx (*Trace* xs) $=$ all (λ(req, rsp@(rspCode $\rightarrow$ (c, _, _))) $\rightarrow$ c $\not\equiv 5$) xs

dynamicScriptProp :: (*Trace* app $\rightarrow$ *Property*) $\rightarrow$ *Setup* $\rightarrow$ (*Script* app $\rightarrow$ *Property*)
dynamicScriptProp prop setup script $=$
  morallyDubiousIOProperty \$
    prop $\langle$\$$\rangle$ runScript setup script

Now we can run Scripts, compute a given Trace properity on each result, and return the Scripts that fail as counter-examples for shrinking, all with:

```
*Main> quickCheck (dynamicScriptProp prop_No5xx setup)
+++ OK, passed 100 tests.
```

Remember properties

If client misbehaves, server **does not crash (5xx responses)**.

$\Rightarrow$ **done (sort of...)**

If client "behaves well", server **does not complain (4xx, 5xx responses)**.

$\Rightarrow$ **state machines**

If client "behaves well", server **"behaves well"**.

$\Rightarrow$ **next talk :)**

If the domain of input values is too large, it becomes harder to hit **interesting** test cases. Possible application constraints:

- Don't POST to DELETEd URIs;
- don't POST folders to folders (if they are not nestable);
- don't GET paragraphs contained in DELETEd documents (if you have gc);
- always DELETE pars before DELETEing containing docs (if you don't have gc).

$\Rightarrow$ **Need application-specific** *Arbitrary***.**

$\Rightarrow$ **Framework should still give me equipment to write it.**

Finite State Machines

```
data SM sid content = SM {fromSM :: Map sid (State sid content)}
  deriving (Show)
```

```
data State sid content =
  State
    {stateId          :: sid
    ,stateStart       :: Bool
    ,stateTerminal    :: Bool
    ,stateTransitions :: [(Gen (ScriptItem content), sid)]
    }
  deriving (Typeable)
```

```
arbitraryScriptFromSM :: SM sid Content → Gen (Script Content)
arbitraryScriptFromSM = ...
```

Finite State Machines

```
newtype DocFolderScript = DocFolderScript (Script Content)
```

```
instance Arbitrary DocFolderScript where
  arbitrary = scriptFromSM docFolderSM
```

```
data Sid = ...
```

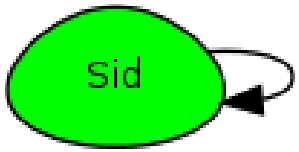
```
docFolderSM :: SM Sid Content
docFolderSM = ...
```

Example

```
data Sid = Sid
```

```
simpleSM :: SM Sid Content
simpleSM = SM $ fromList [(Sid, State Sid True True [(trans, Sid)])]
  where
    trans :: Gen (ScriptItem Content)
    trans = do
      m :: RequestMethod      ← arbitrary
      p  :: Either SBS IxRef  ← arbitrary
      c  :: Either SBS Content ← if m ∈ [POST, PUT]
                                 then Right ⟨$⟩ arbitrary
                                 else Left  ⟨$⟩ pure ""
      return $ mkScriptItemHTTP m c p
```

Example



FSM are too weak

More undecidable than an Finite State Machines (FSM):
We know nodes and transitions statically, but not transition labels!

Still not powerful enough:
When generating new *ScriptItem*, we cannot inspect *Script* prefix.

Give up FSM restriction for a Turing Machine that looks like an FSM!

```
data SM sid content = SM {fromSM :: Map sid (State sid content)}
  deriving (Show)
```

```
data State sid content =
  State
    {stateId          :: sid
    ,stateStart       :: Bool
    ,stateTerminal    :: Bool
    ,stateTransitions :: [Script sid content → Gen (ScriptItem sid content, sid)]
    }
  deriving (Typeable)
```

***SM* are opaque, but *Script* is *SM*-aware!**

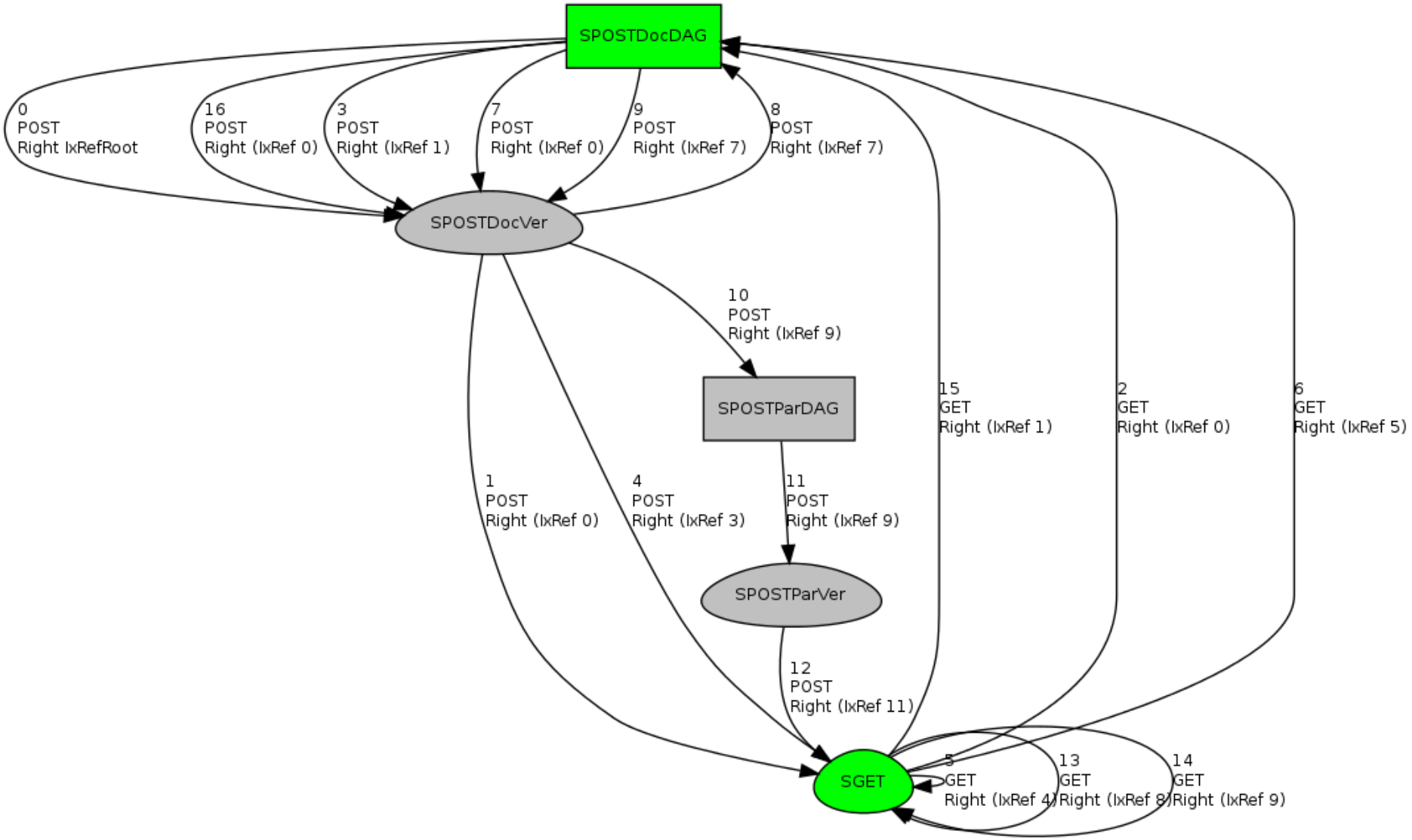
webtest State Machines

```
data ScriptItem sid content =
   ScriptItemHTTP
      { siSerial      :: Ix
      , siFromState   :: State sid content
      , siThisState   :: State sid content
       ...
```

```
newtype Script sid content = Script { scriptItems :: [ScriptItem sid content] }
```

```
scriptToDot :: String → Script sid content → D.Graph
scriptToDot name script = ...
```

Machines are easy to inspect!



Machines are easy to inspect!

```
data Sid = SGET | SPOSTDocDAG | SPOSTDocVer | SPOSTParDAG | SPOSTParVer

docFolderSM :: SM Sid Content
docFolderSM = mkMachine $
  State SGET True True
    [λscript → do
      p ← elements (scriptRefs script)
      next ← elements [SGET, SPOSTDocDAG]
      return (mkScriptItemHTTP NH.GET (Left "") [] p, next)] :
  State SPOSTDocDAG True False
    [λscript → do
      p ← elements (scriptRefs' isPostPool script)
      next ← elements [SPOSTDocVer]
      return (mkScriptItemHTTP NH.POST (new C_IProposalContainer) [] p, next)] :
  State SPOSTDocVer False True
    [λscript → do
      let p = siSerial ∘ last ∘ filter isIProposalContainerPOST ∘ scriptItems $ script
      c ← do
        title ← arbitrary
        descr ← arbitrary
        return $ propSetTitle title ∘ propSetDescr descr ⟨$⟩ new C_IProposal
      next ← elements [SGET, SPOSTDocDAG, SPOSTDocVer, SPOSTParDAG]
      return (mkScriptItemHTTP NH.POST c [] (Right $ IxRef p), next)] :
  State SPOSTParDAG False False
    [λscript → do
      let p = siSerial ∘ last ∘ filter isIProposalContainerPOST ∘ scriptItems $ script
      next ← elements [SPOSTParVer]
      return $ (mkScriptItemHTTP NH.POST (new C_IParagraphContainer) [] (Right $ IxRef p), next)] :
  State SPOSTParVer False True
    [λscript → do
      let p = siSerial ∘ last ∘ filter isIParagraphContainerPOST ∘ scriptItems $ script
      c ← do
        text ← arbitrary
        return $ propSetTitle text ⟨$⟩ new C_IParagraph
      next ← elements [SGET, SPOSTDocDAG, SPOSTParDAG, SPOSTParVer]
      return (mkScriptItemHTTP NH.POST c [] (Right $ IxRef p), next)] :
  []
```

Machines are easy to inspect!

```
 Script {scriptItems = [ScriptItemHTTP {siSerial = 0, siFromState = Just SPOSTDocDAG, siThisState = Just SPOSTDocVer, siM!
!ethod = POST, siBody = Right (Content {getCContentType = C_IProposalContainer, getCPath = Nothing, getCData = fromList [!
!]}), siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = Right IxRefRoot},ScriptItemHTTP {siSerial = 1, si!
!FromState = Just SPOSTDocVer, siThisState = Just SGET, siMethod = POST, siBody = Right (Content {getCContentType = C_IPr!
!oposal, getCPath = Nothing, getCData = fromList []}), siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = !
!Right (IxRef 0)},ScriptItemHTTP {siSerial = 2, siFromState = Just SGET, siThisState = Just SPOSTDocDAG, siMethod = GET, !
!siBody = Left "", siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = Right (IxRef 0)},ScriptItemHTTP {siS!
!erial = 3, siFromState = Just SPOSTDocDAG, siThisState = Just SPOSTDocVer, siMethod = POST, siBody = Right (Content {get!
!CContentType = C_IProposalContainer, getCPath = Nothing, getCData = fromList []}), siGetParams = [], siPostParams = [], !
!siHeaders = [], siHTTPPath = Right (IxRef 1)},ScriptItemHTTP {siSerial = 4, siFromState = Just SPOSTDocVer, siThisState !
!= Just SGET, siMethod = POST, siBody = Right (Content {getCContentType = C_IProposal, getCPath = Nothing, getCData = fro!
!mList []}), siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = Right (IxRef 3)},ScriptItemHTTP {siSerial !
!= 5, siFromState = Just SGET, siThisState = Just SGET, siMethod = GET, siBody = Left "", siGetParams = [], siPostParams !
!= [], siHeaders = [], siHTTPPath = Right (IxRef 4)},ScriptItemHTTP {siSerial = 6, siFromState = Just SGET, siThisState =!
! Just SPOSTDocDAG, siMethod = GET, siBody = Left "", siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = R!
!ight (IxRef 5)},ScriptItemHTTP {siSerial = 7, siFromState = Just SPOSTDocDAG, siThisState = Just SPOSTDocVer, siMethod =!
! POST, siBody = Right (Content {getCContentType = C_IProposalContainer, getCPath = Nothing, getCData = fromList []}), si!
!GetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = Right (IxRef 0)},ScriptItemHTTP {siSerial = 8, siFromSta!
!te = Just SPOSTDocVer, siThisState = Just SPOSTDocDAG, siMethod = POST, siBody = Right (Content {getCContentType = C_IPr!
!oposal, getCPath = Nothing, getCData = fromList []}), siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = !
!Right (IxRef 7)},ScriptItemHTTP {siSerial = 9, siFromState = Just SPOSTDocDAG, siThisState = Just SPOSTDocVer, siMethod !
!= POST, siBody = Right (Content {getCContentType = C_IProposalContainer, getCPath = Nothing, getCData = fromList []}), s!
!iGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = Right (IxRef 7)},ScriptItemHTTP {siSerial = 10, siFromS!
!tate = Just SPOSTDocVer, siThisState = Just SPOSTParDAG, siMethod = POST, siBody = Right (Content {getCContentType = C_I!
!Proposal, getCPath = Nothing, getCData = fromList []}), siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath !
!= Right (IxRef 9)},ScriptItemHTTP {siSerial = 11, siFromState = Just SPOSTParDAG, siThisState = Just SPOSTParVer, siMeth!
!od = POST, siBody = Right (Content {getCContentType = C_IParagraphContainer, getCPath = Nothing, getCData = fromList []}!
!), siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = Right (IxRef 9)},ScriptItemHTTP {siSerial = 12, siF!
!romState = Just SPOSTParVer, siThisState = Just SGET, siMethod = POST, siBody = Right (Content {getCContentType = C_IPar!
!agraph, getCPath = Nothing, getCData = fromList []}), siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = !
!Right (IxRef 11)},ScriptItemHTTP {siSerial = 13, siFromState = Just SGET, siThisState = Just SGET, siMethod = GET, siBod!
!y = Left "", siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = Right (IxRef 8)},ScriptItemHTTP {siSerial!
! = 14, siFromState = Just SGET, siThisState = Just SGET, siMethod = GET, siBody = Left "", siGetParams = [], siPostParam!
!s = [], siHeaders = [], siHTTPPath = Right (IxRef 9)},ScriptItemHTTP {siSerial = 15, siFromState = Just SGET, siThisStat!
!e = Just SPOSTDocDAG, siMethod = GET, siBody = Left "", siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath !
!= Right (IxRef 1)},ScriptItemHTTP {siSerial = 16, siFromState = Just SPOSTDocDAG, siThisState = Just SPOSTDocVer, siMeth!
!od = POST, siBody = Right (Content {getCContentType = C_IProposalContainer, getCPath = Nothing, getCData = fromList []})!
!, siGetParams = [], siPostParams = [], siHeaders = [], siHTTPPath = Right (IxRef 0)}]}
```

Executive summary

Quickcheck approximates **brute-force tests** by randomly testing input values, and it's just thorough enough to be feasible. This is surprisingly effective.

If you want to test REST APIs, **make HTTP request lists the input type** and use quickcheck's *Arbitrary* infrastructure.

State machines are a tool for generating arbitrary scripts that **make sense** from the application point of view.

Acknowledgements and further reading

- Koehn Claessen, John Hughes, *Testing Monadic Code with QuickCheck*
- Adam Curtis, *hackage://webdriver*
- Edward A. Kmett, *hackage://free*
- Andres Löh, *Free Monads*, Haskell eXchange 2013, London
- Edward A. Kmett, Rúnar Bjarnason, *hackage://machines*
- Max Bolingbroke, *hackage://test-framework*

Future work

Extend *ScriptItem* type with **webdriver** (Selenium) requests for combined frontend-backend testing.

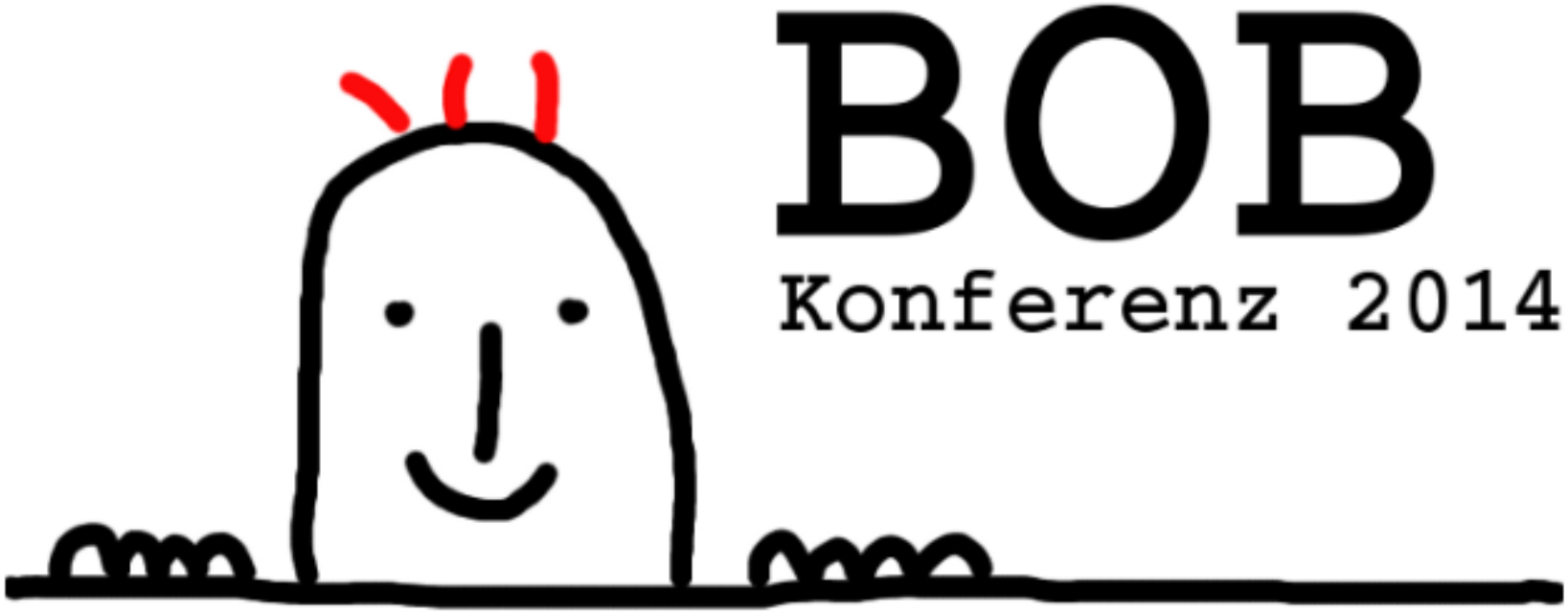
Compile *Script* and shallow-embedded equivalence properties to **python and javascript** (so we can generate Haskell-independent unit-test suites).

Test robustness with **Fuzz class** that make small random changes to valid input data.

Graphical debugger:

- generate state transition graphs from run traces;
- compile not to dot, but to d3js
- make them explorable HTTP debug traces.

Thank you!



Optimales Handwerkszeug für Software-Entwickler

Frameworks - Libraries - Paradigmen - Sprachen - Tools

Active Group GmbH, Filderstadt

factis research GmbH, Freiburg

`funktionale-programmierung.de`

Neue Konferenz 2014!

- Planung noch am Anfang
- funktionale Sprachen
  (Haskell, Clojure, Erlang, Scala)
- Ökosystem um BoB-Technologien
- polyglotte Projekte
- **+ ???: Umfrage ausfüllen!**

Mehr Infos im Februar!

bobkonf.de

@bobkonf