Two Case Studies: QuickCheck and Wash/CGI

Lecture 4,
Designing and Using Combinators
John Hughes

Motivations

• Two case studies (software testing, server-side web scripting) in which a DSEL played an essential rôle.

• Two DSELs with a monadic design.

• Three interesting monads!

QuickCheck: The Research Hypothesis

Formal specifications can be used directly for ***software testing*** in combination with ***random test case generation***.

Needed:

• a language to express formal specifications.

• a way to specify test case generation.

• a tool to carry out tests.

QuickCheck: The Research Hypothesis

Formal specifications can be used directly for ***software testing*** in combination with ***random test case generation***.

Needed:

• a language to express formal specifications.

• a way to specify test case generation.

• a tool to carry out tests.

Solution: a DSEL!

Implemented in 350 lines of code.

A "Demo"

```
prop_PlusAssoc x y z =
    (x + y) + z == x + (y + z)
```

Property encoded as a Haskell function

```
Main> quickCheck prop_PlusAssoc
```

Invoke quickCheck to test it

A "Demo"

```
prop_PlusAssoc x y z =
    (x + y) + z == x + (y + z)
```

```
Main> quickCheck prop_PlusAssoc
ERROR - Unresolved overloading
*** Type      : (Num a, Arbitrary a) => IO ()
*** Expression : quickCheck prop_PlusAssoc
```

A “Demo”

```
prop_PlusAssoc :: Integer -> Integer -> Integer -> Bool
prop_PlusAssoc x y z =
    (x + y) + z == x + (y + z)
```

```
Main> quickCheck prop_PlusAssoc
OK, passed 100 tests.
```

A “Demo”

```
prop_PlusAssoc :: Float -> Float -> Float -> Bool
prop_PlusAssoc x y z =
    (x + y) + z == x + (y + z)
```

```
Main> quickCheck prop_PlusAssoc
Falsifiable, after 0 tests:
2.33333
2.0
-2.0
```

Values for x, y, and z

A “Demo”

```
prop_Insert :: Integer -> [Integer] -> Bool
prop_Insert x xs =
    ordered (insert x xs)

ordered xs = and (zipWith (<=) xs (drop 1 xs))
```

A “Demo”

```
prop_Insert :: Integer -> [Integer] -> Bool
prop_Insert x xs =
    ordered (insert x xs)

ordered xs = and (zipWith (<=) xs (drop 1 xs))
```

```
QuickCheck> quickCheck prop_Insert
Falsifiable, after 2 tests:
-3
[3,-4,3]
```

A “Demo”

```
prop_Insert :: Integer -> [Integer] -> Property
prop_Insert x xs =
    ordered xs ==> ordered (insert x xs)

ordered xs = and (zipWith (<=) xs (drop 1 xs))
```

```
Main> quickCheck prop_Insert
OK, passed 100 tests.
```

A “Demo”

```
prop_Insert :: Integer -> [Integer] -> Property
prop_Insert x xs =
    ordered xs ==>
        collect (length xs) $ ordered (insert x xs)
ordered xs = and (zipWith (<=) xs (drop 1 xs))
```

Investigate test coverage

A “Demo”

```
prop_Insert :: Integer -> [Integer] -> Property
prop_Insert x xs =
    ordered xs ==>
        collect (length xs) $ ordered (insert x xs)
ordered xs = and (zipWith (<=) xs (drop 1 xs))
```

```
Main> quickCheck prop_Insert
OK, passed 100 tests.
46% 0.
26% 1.
19% 2.
8% 3.
1% 4.
```

A “Demo”

```
prop_Insert :: Integer -> [Integer] -> Property
prop_Insert x xs =
    ordered xs ==>
        collect (length xs) $ ordered (insert x xs)
ordered xs = and (zipWith (<=) xs (drop 1 xs))
```

```
Main> quickCheck prop_Insert
OK, passed 100 tests.
46% 0.
26% 1.
19% 2.
8% 3.
1% 4.
```

A random list is unlikely to be ordered unless it is very short!

A “Demo”

```
prop_Insert :: Integer -> Property
prop_Insert x = forAll orderedList $ \xs ->
        collect (length xs) $ ordered (insert x xs)

ordered xs = and (zipWith (<=) xs (drop 1 xs))
```

A “Demo”

```
prop_Insert :: Integer -> Property
prop_Insert x = forAll orderedList $ \xs ->
        collect (length xs) $ ordered (insert x xs)

ordered xs = and (zipWith (<=) xs (drop 1 xs))
```

```
Main> quickCheck prop_Insert
OK, passed 100 tests.
20% 2.
17% 0.
15% 1.
11% 3.
9% 5.
7% 4.
5% 8.
3% 9.
3% 7.
3% 11.
2% 14.
1% 6.
1% 19.
1% 15.
1% 13.
1% 10.
```

Property Language

```
property ::=  boolExp
            | \x -> property
            | boolExp ==> property
            | forAll set $ \x -> property
            | collect expr property

test ::= quickCheck property
```

Set = Test Data Generator

```
orderedList :: (Ord a, Arbitrary a) => Gen [a]
orderedList =
    oneof
        [return [],
         do xs <- orderedList
            n <- arbitrary
            return ((case xs of
                       [] -> n
                       x:_ -> n `min` x)
                    :xs)]
```

Generator: a monad!

Random choice

Type based generation

Set = Test Data Generator

```
orderedList :: (Ord a, Arbitrary a) => Gen [a]
orderedList =
    frequency
        [(1,return []),
         (4,do xs <- orderedList
               n <- arbitrary
               return ((case xs of
                          [] -> n
                          x:_ -> n `min` x)
                       :xs))]
```

Specified distribution

Type Based Generation

```
class Arbitrary a where
    arbitrary :: Gen a
```

instance Arbitrary Integer **where** …

instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b) **where** …

instance Arbitrary a => Arbitrary [a] **where** …

Defines default generation method by recursion over the type!

Type Based Testing

```
class Testable a where
    property :: a -> Property
```

instance Testable Bool **where** …

instance (Arbitrary a, Show a, Testable b) => Testable (a->b)

where property f = forAll arbitrary f

quickCheck :: Testable a => a -> IO ()

Testing by recursion on types.

Generation Language

```
gen ::= return expr
      | do {x <- gen}* gen
      | arbitrary
      | oneof [gen*]
      | frequency [(int,gen)*]
```

How does the Gen monad work?

Random Numbers in Haskell

```
class RandomGen g where
    next :: g -> (Int, g)
    split :: g -> (g, g)
```

A random number seed can be split into two independent seeds.

Idea

Parameterise actions on a random number seed.

>>= supplies independent seeds to its operands.

A Generator Monad Transformer

```
newtype Generator g m a = Generator (g -> m a)

instance (RandomGen g, Monad m) =>
         Monad (Generator g m) where
    return x = Generator $ \g -> return x
    Generator f >>= h = Generator $ \g ->
      let (g1,g2) = split g in
          do a <- f g1
             let Generator f' = h a
             f' g2
```

Representation of Properties

```
newtype Property = Prop (Gen Result)
```

Generates a test result!

```
data Result
  = Result { ok :: Maybe Bool,
             arguments :: [String],
             stamp :: [String]}
```

forAll

collect

Did it Work?

Only 350 lines, but the combination of specifications and random testing proved very effective.

Used by

- Okasaki (Columbia State) to develop data structure library
- Andy Gill to develop a Java (!) pretty-printing library
- Team *Functional Beer* in the ICFP Programming Contest
- Galois Connections, likewise
- Safelogic, to develop transformer for first order logic
- Ported to Mercury

i.e. used in Swedish and US industry

Current Work: Testing Imperative ADTs

- Specify ADT operations by a simple Haskell implementation.

```
type Queue a = [a]
empty = []
add x q = x:q
front (x:q) = x
remove (x:q) = q
```

Current Work: Testing Imperative ADTs

- Specify ADT operations by a simple Haskell implementation.
- Construct imperative implementation.

```
data QueueI r a =
    Queue (r (QCell r a))
          (r (QCell r a))

addI :: RefMonad m r =>
    a -> Queue r a -> m ()
```

Current Work: Testing Imperative ADTs

- Specify ADT operations by a simple Haskell implementation.
- Construct imperative implementation.
- Model a *language* of operations as a datatype, with interpretations on specification and implementation.

```
data Action a =
    Add a | Front | Remove
spec :: [Action a] ->
    Queue a -> Queue a
impl :: RefMonad m r =>
    [Action a] ->
        QueueI r a -> m ()
```

Current Work: Testing Imperative ADTs

- Specify ADT operations by a simple Haskell implementation.
- Construct imperative implementation.
- Model a *language* of operations as a datatype, with interpretations on specification and implementation.
- Define *retrieval* of the implementation state.

```
retrieve ::
    RefMonad m r =>
        QueueI r a ->
            m (Queue a)
```

Current Work: Testing Imperative ADTs

- Specify ADT operations by a simple Haskell implementation.
- Construct imperative implementation.
- Model a *language* of operations as a datatype, with interpretations on specification and implementation.
- Define *retrieval* of the implementation state.
- Compare results after random sequences of actions.

```
prop_Queue = forAll actions $ \as ->
  runST (do
    q <- emptyI
    impl as q
    abs <- retrieve q
    return (abs==spec as empty))
```

Future Work

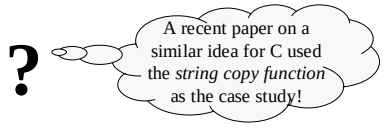
- Use Haskell's *foreign function interface* to test software in other languages.
  - Haskell ==> executable specification language
  - QuickCheck ==> specification based testing system

QuickCheck Summary

- QuickCheck is a state-of-the-art testing tool.
- DSEL approach made the tool *extremely easy to construct and modify*, let us experiment, identify and solve problems not previously addressed.
- Could not have carried out the same research without it!

QuickCheck Summary

- QuickCheck is a state-of-the-art testing tool.
- DSEL approach made the tool *extremely easy to construct and modify*, let us experiment, identify and solve problems not previously addressed.
- Could not have carried out the same research without it!



Wash/CGI: The Goal

Ease the programming of active web pages, implemented using the CGI interface.

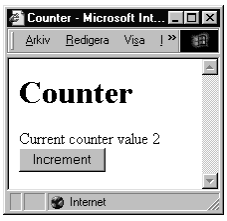
- The CGI interface provides *server side* scripting, via programs which generate HTML running on the server -- in contrast to e.g. Javascript or applets, which run in the browser.
- Most suitable for e.g. querying/updating databases on the server, where instant response is not important.
- An "old" standard, therefore portable: supported by all servers.

Counter Example

```
main = run $ counter 0

counter n = ask $
    standardPage "Counter" $
    makeForm $
     do text "Current counter value "
        text (show n)
        br empty
        submitField (counter (n+1))
          (fieldVALUE "Increment")
```

Counter Example

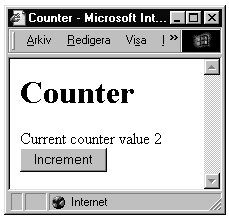


Run function

```
main = run $ counter 0

counter n = ask $
    standardPage "Counter" $
    makeForm $
     do  text "Current counter value "
         text (show n)
         br empty
         submitField (counter (n+1))
           (fieldVALUE "Increment")
```

Counter Example



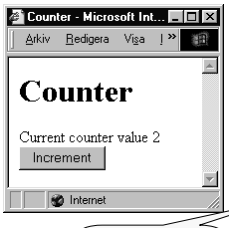
Run function

```
main = run $ counter 0

counter n = ask $
    standardPage "Counter" $
    makeForm $
     do  text "Current counter value "
         text (show n)
         br empty
         submitField (counter (n+1))
           (fieldVALUE "Increment")
```

Create an active page

Counter Example



Run function

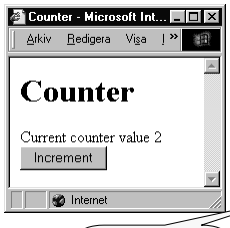
```
main = run $ counter 0

counter n = ask $
    standardPage "Counter" $
    makeForm $
     do  text "Current counter value "
         text (show n)
         br empty
         submitField (counter (n+1))
           (fieldVALUE "Increment")
```

Create an active page

Monad for HTML generation

Counter Example



Run function

```
main = run $ counter 0

counter n = ask $
    standardPage "Counter" $
    makeForm $
     do  text "Current counter value "
         text (show n)
         br empty
         submitField (counter (n+1))
           (fieldVALUE "Increment")
```

Create an active page

Monad for HTML generation

Callback function

Extended Counter

```
counter n = ask $
    standardPage "Counter" $
    makeForm $
     do  text "Current counter value "
         activeInputField counter (fieldVALUE (show n))
         submitField (counter (n+1)) (fieldVALUE "++")
         submitField (counter (n-1)) (fieldVALUE "--")
```

File Uploader

```
main = run$ ask$ page$ makeForm$
   do text "File to upload: "
      file <- fileInputField empty
      submitField (receive (raw file))
           (fieldVALUE "Send file")

receive [f] =
   if take 2 (fileName f) == ".." then htell $ page $ text "Naughty!"
   else do io (writeFile ("uploaded.files/"++fileName f)
                         (fieldContents f))
           htell $ page $ text "File uploaded"
```

File Uploader

Creates an input field and delivers the value input.

```
main = run$ as$ page$ makeForm$
  do text "File to upload: "
     file <- fileInputField empty
     submitField (receive (raw file))
          (fieldVALUE "Send file")

receive [f] =
  if take 2 (fileName f) == ".." then htell $ page $ text "Naughty!"
  else do io (writeFile ("uploaded.files/"++fileName f)
                        (fieldContents f))
          htell $ page $ text "File uploaded"
```

File Uploader

Creates an input field and delivers the value input.

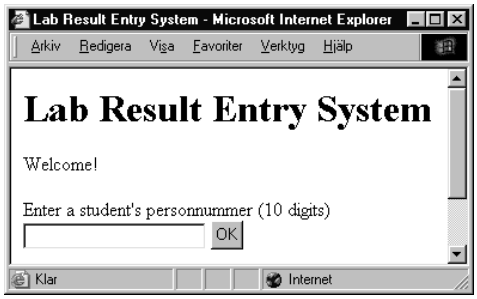
```
main = run$ as$ page$ makeForm$
  do text "File to upload: "
     file <- fileInputField empty
     submitField (receive (raw file))
          (fieldVALUE "Send file")

receive [f] =
  if take 2 (fileName f) == ".." then htell $ page $ text "Naughty!"
  else do io (writeFile ("uploaded.files/"++fileName f)
                        (fieldContents f))
          htell $ page $ text "File uploaded"
```

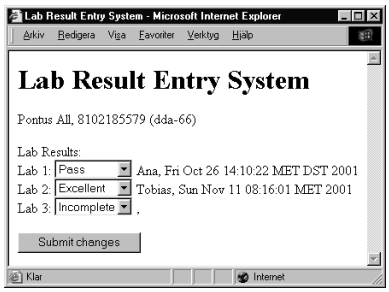
Can do I/O on the server

Lab Result Entry System

Lab Result Entry System



Lab Result Entry System



Lab Result Entry System

```
editPerson pn pr = ask $ page $ makeForm $
  do text (forename pr++" "++aftername pr++", "++
           pn++" ("++email pr++")")
     p empty
     text "Lab Results:"
     br empty
     gs <- sequence (map (editLab (labs pr)) labNames)
     br empty
     submitField (commit pn pr gs)
          (fieldVALUE "Submit changes")
```

Pass name, personal number, and grades to callback

Lab Result Entry System

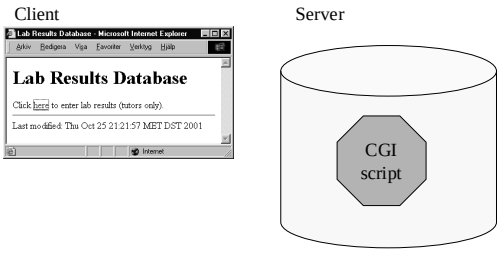
```
commit pn pr gs =
  do io (do d <- getDate
            putRecord (PN pn)
              (pr{labs=updateLabs (labs pr) gs d}))
     mainPage "Database updated"
```

Wash/CGI Paradigm

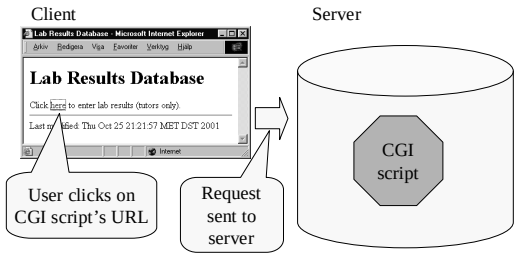
- HTML is generated just by calling a function for each element.
- Input fields just return (a structure containing) the value input.
- Active elements (e.g. submit buttons) just invoke a “callback function”.
- State is recorded by parameter passing, in the usual way.

A very simple and natural paradigm!

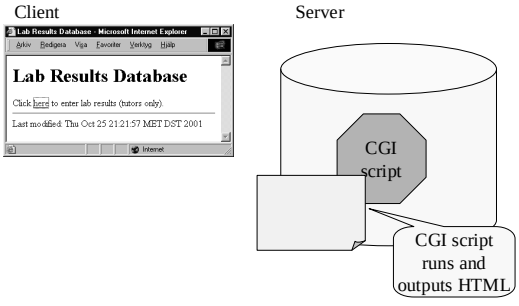
How CGI Works



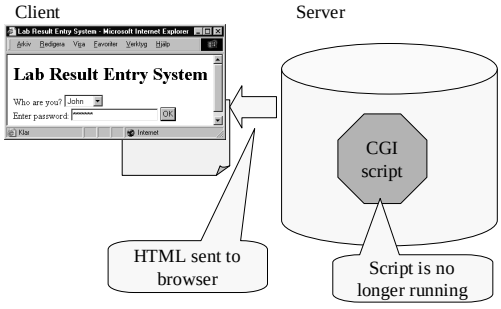
How CGI Works



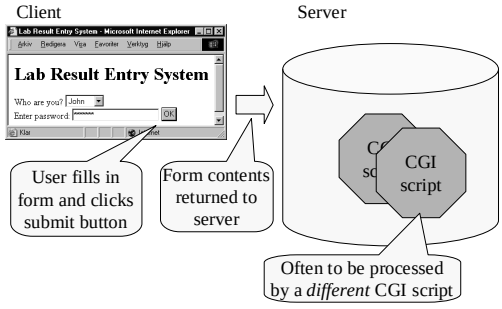
How CGI Works



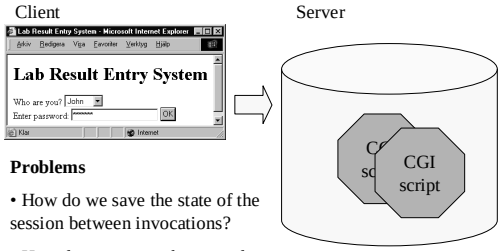
How CGI Works



How CGI Works



How CGI Works



Problems

- How do we save the state of the session between invocations?
- How do we ensure the second CGI script correctly interprets the form generated by the first?

Saving the State

Where should the state be saved?

- On the server?

What if the client just closes the window? -- How long should the state be saved? What if the client presses Back, and continues from a previous state?

- On the client?

Each window saves the state of its session.
Back is easy to handle.
Implement using "hidden fields" in HTML forms.

How Can We Save the State?

- Wash/CGI implements an entire session by *one* CGI script.
- When the script is resumed, it is (of course) reinvoked at the beginning.
- The script decodes state stored in the browser input, and *reruns the computation* up to the point of last suspension.
- Rerunning purely functional code produces the same results -- but input/output might not!

How can we rerun the script, *without* repeating any input/output it performed?

A Monad Transformer for Resumable Actions

- suspend :: Monad m => Resume m ()

Suspend and save state, in a restartable form

- resume :: Monad m =>
  [String] -> Resume m a -> m (Either [String] a)

The "run function": start in a given state, either suspend or terminate

State is a list of strings

- once :: (Monad m, Show a, Read a) =>
  Resume m a -> Resume m a

Do this once only: save result in the state

Example of Resuming

```
liftR m = once (lift (lift m))

example =
 do liftR (putStr "Input? ")
    x <- liftR getLine
    suspend
    liftR (putStr (x++"\n"))
```

```
Main> resume [] example
Input? 23
Left ["()","\"23\"",""]
Main> resume ["()","\"23\"",""] example
23
Right ()
```

Defining Resume

The Resume monad needs two features:

• A *state*, containing

(i) Saved results from previous runs

(ii) Collected results for the next run

• An *exception mechanism* to enable an abrupt stop, delivering the current state.

Defining Resume

```
type Resume m a =
        State ([String],[String]) (Exception [String] m) a
```

Saved results | Generated results | State on suspension

Defining Resume

```
type Resume m a =
        State ([String],[String]) (Exception [String] m) a
```

```
resume :: Monad m =>
    [String] -> Resume m a -> m (Either [String] a)
resume old m = runException $ runState (old,[]) $
     do  a <- m
         return (Right a)
     `handleWith` \new ->
         return (Left new)
```

Defining Resume

```
type Resume m a =
        State ([String],[String]) (Exception [String] m) a
```

```
suspend :: Monad m => Resume m ()
suspend =
     do  (old,new) <- readState
         case old of
            [] -> exception (reverse (":new))
            x:old' -> writeState (old',x:new)
```

Defining Resume

```
type Resume m a =
        State ([String],[String]) (Exception [String] m) a
```

```
once :: (Monad m, Show a, Read a) =>
                Resume m a -> Resume m a
once m =
     do  (old,new) <- readState
         case old of
            [] ->       do  a <- m
                            writeState (old,show a:new)
                            return a
            a:old' -> do writeState (old',a:new)
                            return (read a)
```

The CGI Monad

Wash/CGI defines a monad CGI:

• Provides resumption as described here (but without explicitly using monad transformers)

• Maintains a state to generate unique *field names* in HTML forms

• Handles input fields in forms

How Input Fields Work

Bound to the value input?

```
a <- textInputField empty
…
… value a …
```

Generate HTML for an input field

?

How can we have the input value already?

How Input Fields Work

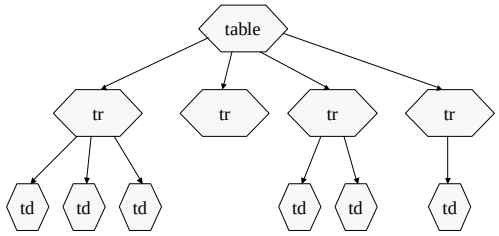
```
a <- textInputField empty
…
… value a …
```

Just a, or Nothing

- Generates HTML and returns Nothing on the *first* run.
- Decodes the value and returns Just a on *subsequent* runs.
- Better not try to use the value until after a suspension!

HTML Generation

HTML is represented as a tree structure:



HTML Generation

HTML is represented as a tree structure:

```
data ELEMENT_ =
   ELEMENT_ { tag :: String
            , attrs :: [ATTR_]
            , elems :: [ELEMENT_]
            }
 | …
```

HTML Monad Transformer

```
type WithHTML m a = State ELEMENT_ m a
```

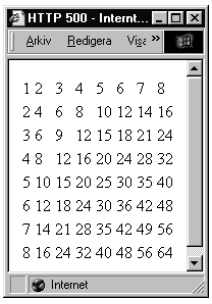
HTML constructors

- add a new sub-ELEMENT_ to the current ELEMENT_
- take as argument a WithHTML action, which adds their own sub-ELEMENT_s

Typical type: WithHTML m a -> WithHTML m a

HTML Example

```
table (sequence
   [tr (sequence
         [td (text (show (x*y)))
         | x <- [1..8]]
   | y <- [1..8]])
```



Wash/CGI Summary

- Wash/CGI brings new power to CGI programmers
- Solves two awkward problems:
  - saving and restoring state across client interactions
  - consistent generation and interpretation of forms
- The DSEL went through many versions: *flexibility* led to a very clean design in the end
- Exploits integration with Haskell through e.g. callback functions