LECTURE 13

Theory and Design of PL (CS 538)

March 4, 2020

PROPERTY-BASED TESTING

UNIT TESTING IS BORING

- Most common kind of test today
- Idea: write test cases one by one
  - Write down one input (and maybe external state)
  - Write down expected output
  - Check if input produces expected output
- Build up a lot of hand-crafted tests
  - Write new tests when bugs are found
  - Keep tests up to date

IDEA: TEST PROPERTIES

- Idea: write down *properties* of programs
- Properties hold for class of inputs, not just one input
- Don't need to write tests one-by-one

Randomly generate test cases!

EXAMPLES

- Applying twice same as applying once (idempotence)
- One function “undoes” another function (inverse)
- Optimized implementation mirrors simple version
- Relationships between insert, delete, lookup, etc.

QUICKCHECK

- Haskell library for property-based testing
  - Write random input generators with combinators
  - Write properties of functions we want to test
- Quickcheck will randomly generate and test
  - “Shrinks” failing test inputs to find minimal ones
- Implementations in at least 40+ other languages

TAKING IT FOR A SPIN

- Install with Cabal (or Stack)

```
cabal v2-install --lib QuickCheck
```

- Import Haskell module

```haskell
import Test.QuickCheck
```

- Documentation available here

QUICKCHECK DEMO

HOW TO TEST A PARSER?

- Parser goes from String to structured data
- How to generate input Strings?
  - Randomly? Almost certainly won't parse…
- Even if parser succeeds, is the answer is right?

USE THE PRETTY-PRINTER (HW3)

- Parser: String to structured data
- Printing: structured data to String
  - This direction is usually easy...

Inverse property: printing data, then parsing it back should give original data!

HOW IS THE LIBRARY DESIGNED?

GEN TYPE

- `Gen a`: something that can generate random `a`'s

```
-- Build generator drawing a's from a list of a's
elements :: [a] -> Gen a

-- Select a random generator from a list
oneof :: [Gen a] -> Gen a

-- Customize distribution of generators
frequency :: [(Int, Gen a)] -> Gen a
```

GEN IS A MONAD

```
instance Monad Gen where
  return :: a -> Gen a

  (>>=) :: Gen a -> (a -> Gen b) -> Gen b
```

- Return: from val of type `a`, build generator that always returns val
- Bind: draw something from first generator (of `a`'s) and use to select the next generator (of `b`'s)

COMBINING GENERATORS

- Combinators: build new generators out of old ones

```haskell
-- Turn generator of a into generator of pairs of a
genPairOf :: Gen a -> Gen (a, a)
genPairOf g = do x <- g
                 y <- g
                 return (x, y)

-- Turn generator of a into generator of lists of a
vectorOf :: Int -> Gen a -> Gen [a]
vectorOf 0 _ = return []
vectorOf n g = do x  <- g
                  xs <- vectorOf (n - 1) g
                  return (x:xs)
```

TYPECLASS: ARBITRARY

- `Arbitrary a`: means `a` is “generatable”
- Concretely: there is something of type `Gen a`

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

USING ARBITRARY

- Typeclass machinery will automatically get generator
- Compare with previous: no need to pass in `Gen a`

```haskell
genPair :: Arbitrary a => Gen (a, a)
genPair = do x <- arbitrary   -- From typeclass constraint
             y <- arbitrary   -- Automatically inferred
             return (x, y)

vector :: Arbitrary a => Int -> Gen [a]
vector 0 = return []
vector n = do x  <- arbitrary
              xs <- vector (n - 1)
              return (x:xs)
```

INSTANCES OF ARBITRARY

- Library has tons of instances for base types
  - Arbitrary Bool, Arbitrary Char, Arbitrary Int, …
- Also has instances for more complex types

```
instance (Arbitrary a, Arbitrary b)
         => Arbitrary (a, b) where          -- products
instance (Arbitrary a, Arbitrary b)
         => Arbitrary (Either a b) where   -- sums
instance Arbitrary a
         => Arbitrary [a] where            -- lists
```

ARBITRARY PRODUCTS

```haskell
instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b) where
  arbitrary = do getA <- arbitrary  -- type: Gen a
                 getB <- arbitrary  -- type: Gen b
                 return (getA, getB)
```

ARBITRARY SUMS

```
instance (Arbitrary a, Arbitrary b) => Arbitrary (Either a b) whe
  arbitrary = oneOf [ do aa <- arbitrary  -- type: Gen a
                         return (Left aa)
                    , do bb <- arbitrary  -- type: Gen b
                         return (Right bb) ]
```

TESTING PROPERTIES

- Combine generator of `a`'s and property of `a`'s

```
forAll :: Show a => Gen a -> (a -> Bool) -> Property

myProp2 = forAll genX $ \x ->
            forAll genY $ \y ->
              fst (x, y) == x
```

ADDITIONAL FEATURES

- Sizes: control “size” of generated things
- Shrinking: given a failing test case, “make it smaller”
  - Search for simplest failing test cases
  - Can customize how to shrink test cases
- Implications: if one prop holds, then other one holds
  - “If input is valid, then function behaves correctly”

QUICK REVIEW: OUR FAVORITE TYPES

ALWAYS THE SAME PATTERN

1. Add a new type
2. Add constructor expressions
3. Add destructor expressions
4. Add typing rules for new expressions
5. Add evaluation rules for new expressions

FUNCTION TYPES

1. Type of the form $s \to t$, where s, t are types
2. Constructing functions: $\lambda x.\ e$
3. Destructing functions: $e\ e'$

FUNCTIONS IN HASKELL

```
-- Function types look like this
myFun :: Int -> String

-- Building functions
myFun = \arg -> "Int: " ++ (show arg)

mySameFun arg = "Int: " ++ (show arg)

-- Using functions
myOtherFun = myFun 42
```

PRODUCT TYPES

1. Type of the form $s \times t$, where s, t are types
2. Constructing pairs: (e_1, e_2)
3. Destructing pairs: $\mathrm{fst}(e)$ and $\mathrm{snd}(e)$
    - Or: pattern match

> *Think: an* $s \times t$ *contains an* s *AND a* t

PRODUCTS IN HASKELL

```haskell
-- Product types look like this:
myPair :: (Bool, Int)

-- Building pairs
myPair = (True, 1234)

-- Using pairs via projections
myFst = fst myPair  -- True
mySnd = snd myPair  -- 1234
```

RECORDS IN HASKELL

- “Record types”: products in disguise

```
-- Declaring a record type
data RecordType = MkRt { getBool :: Bool, getInt :: Int }
myRecord :: RecordType

-- Building records
myRecord = MkRt { getBool = True, getInt = 1234 }

-- Using records via accessors
myBool = getBool myRecord -- True
myInt  = getInt  myRecord -- 1234

-- Using records pattern match
myFoo = case myRecord of
          MkRt { getBool = b, getInt = i } -> ... b ... i ...
```

SUM TYPES

1. Type of the form $s + t$, where s, t are types
2. Constructing sums: $\mathrm{inl}(e_1), \mathrm{inr}(e_2)$
3. Destructing sums: case analysis/pattern match
    - Can't use fst/snd: don't know if it's an s or a t!

> *Think: an $s + t$ contains an s OR a t*

SUMS IN HASKELL

```haskell
-- Sum types look like this:
data BoolPlusInt = Inl Bool | Inr Int

-- Building sums: two ways
myBool = Inl True :: BoolPlusInt
myInt  = Inr 1234 :: BoolPlusInt

-- Using sums: pattern match
myFun :: BoolPlusInt -> String
myFun bOrI = case bOrI of
               Inl b -> "Got bool: " ++ (show b)
               Inr i -> "Got int: "  ++ (show i)
```

THE ALGEBRA OF DATATYPES

WHAT IS AN ALGEBRA?

- A bunch of stuff you can multiply and add together
- Think: high-school algebra, polynomials, etc.
- How can we multiply and add **types**?

More care needed for non-termination
(Course theme: we won’t be careful)

WHEN ARE TWO TYPES “THE SAME”?

- Given two equivalent types t and s:
  - Program (function) converting t to s
  - Program (function) converting s to t
  - Converting back and forth should be identity
- We call such types *isomorphic*, and write $t \cong s$

FINITE TYPES

- A type with no values: 0 (“Void”/“Empty”/“False”)
  - No constructors
- A type with one possible value: 1 (“Unit”)
  - Exactly one constructor: $()$
- A type with two possible values: 2 (“Bool”)
  - Exactly two constructors: $\mathtt{true}$ and $\mathtt{false}$
- A type with three possible values: 3 (“Three”)
- ...

EXPECTED EQUATIONS HOLD!

- Basic arithmetic

$$2 \times 2 \cong 1 + 1 + 1 + 1 \cong 4$$

- Commutativity

$$t \times s \cong s \times t \qquad t + s \cong s + t$$

- Associativity

$$t_1 + (t_2 + t_3) \cong (t_1 + t_2) + t_3$$

EXPONENTIALS

- Write s^t for function types $t \to s$
- Satisfies the expected properties, for instance:
  - Arithmetic: $2^2 \cong 4, t^2 \cong t \times t$
  - Tower rule: $(Z^Y)^X \cong Z^{X \times Y}$
  - $Z^{X+Y} \cong Z^X \times Z^Y$

DERIVATIVES

- High-school calculus: derivative of X^n is $n \times X^{n-1}$
  - Surprisingly: forms the *zipper* of a type!
  - Original type with a “hole” in it
- Example: derivative of pair $t \times t$ is $2 \times t \cong t + t$
  - Left: hole in first component, t in second
  - Right: hole in second component, t in first

INDUCTIVE TYPES

```
data List a = Nil | Cons a (List a)
```

- Reading: should satisfy $\mathrm{List}(\mathrm{t}) \cong 1 + \mathrm{t} \times \mathrm{List}(\mathrm{t})$
- One solution: $1 + \mathrm{t} + \mathrm{t}^2 + \mathrm{t}^3 + \cdots$
  - Reading: either empty, or one t, or two t, ...
- Take derivative: $1 + 2 \times \mathrm{t} + 3 \times \mathrm{t}^2 + \cdots$
  - You’ve programmed with this type before...

HASKELL WRAPUP

HIGHLY EXPERIMENTAL

- Original goal: implement a lazy language
- An academic experiment that escaped from the lab
  - "Avoid success at all costs"
- Remains a testbed for wild and wacky PL ideas
- GHC has a huge list of experimental flags
  - IncoherentInstances, UndecidableInstances, RankNTypes, GADTs, …

HASKELL IS EXTREME

- Extreme control of side-effects: can’t just print a line!
- Pervasive use of monads: hard to avoid
- Style encourages lots of symbol operators
  - Impossible to Google, looks like line noise
- Takes abstraction to an extreme
  - Highly generic and reusable code
  - Very dense: looks small, but unpacks to a lot

TREMENDOUS INFLUENCE

- Popularized many features
  - Typeclasses and polymorphism
  - Algebraic datatypes and pattern matching
  - Higher-order functions
- Showed: strongly-typed languages can be elegant
  - Every language should have type inference
- Changed how people think about programming
  - Got lots of people to learn about monads

DOES ANYONE USE THIS?

- More than you might think
  - Finance: Credit Suisse, DB, JPM, Standard Chartered, Barclays, HFT shops, …
  - Big tech: Microsoft, Facebook, Google, Intel, …
  - R&D: Galois, NICTA, MITRE, …
  - Security: Kaspersky, lots of blockchain, …
  - Startups: too many
- Strengths
  - Anything working with source code
  - Static analysis, transformations, compilers, …
  - Hardware design

CAN THIS STUFF GO FAST?

- Haskell code can be really fast
  - Can be competitive with C, sometimes
  - GHC is highly optimizing, use purity and types
- Performance tuning makes a huge difference
  - Requires very solid understanding of GHC
  - Few resources, somewhat of a dark art
  - You have to know what you’re doing

LAZY VERSUS EAGER

- Laziness is double-edged
- Elegant, simple code via recursion
  - Very natural to work with infinite data
  - Usually don't hit non-termination
- Hard to reason about performance, especially space
  - Things might not be run when you think they are
  - "Space leaks": buildup of suspended computations

LESS RADICAL COUSINS

- Haskell is a member of the ML family of languages
- Same family: OCaml, SML, F#, Purescript, …
  - More popular in industry
  - FB version with curly braces and semicolons
- General features
  - Eager, easier to reason about performance
  - No control of side effects (no monads)
  - No typeclasses (but real modules)
  - Syntax is very similar to Haskell

SCRATCHING THE SURFACE

- Much more to Haskell than we covered
  - Type level programming, dependent types
  - Concurrency and parallelism
  - Generic Haskell (how does `derive` work?)
  - Arbitrarily complex category theory stuff
- Related systems
  - Liquid Haskell: types with custom assertions
  - Agda: computer-aided proof assistant