QuickCheck

- Haskell module for testing properties of Haskell Programs
- Properties are defined as functions of one or more variables, returning a value of type `Bool`
- QuickCheck tests a property (using function `test!`) by generating random test values for each of its arguments, and then evaluating the property.
- It repeats this 100 times, or until a `False` result is returned.
- In the event of a `False` result, it prints out the test values that caused the test to fail, otherwise it states that the test was `Ok`.
- See http://www.cs.chalmers.se/~rjmh/QuickCheck/
- Simply import `Test.QuickCheck` to use it.

Monomorphism Type Restriction

- QuickCheck cannot work with functions of polymorphic type (e.g. `[a]`), but instead must have a fully specified type (monomorphic), (e.g. `[Int]`).
- This is required because QuickCheck needs to know how to generate and print test values, so it needs a concrete type to work with.
- If a concrete type is not specified,
  - if the type has to belong to the `Num` class, it uses `Int`,
  - otherwise, it uses the empty type `()`

Example: Testing List reversal

- The function `reverse :: [a] -> [a]` reverses its list argument.
- If we reverse the list twice, we should get the original back:
  `reverse (reverse xs) == xs`
- We can define such a property for lists as:
  `prop_RevRev xs = reverse (reverse xs) == xs`
- This is just a standard Haskell function definition, so it has type
- This is polymorphic, so when we test it we should give it a concrete type, but let's not bother:
  `test prop_RevRev`
  - DEMO: Testing the Double Reverse Property
List Reversal, continued

- If we reverse a list once, we do not get the original back.
- Let’s test this:
  ```haskell
test prop_rev :: [Int] -> Bool
```  
  DEMO: Testing the Single Reverse Property
- We see that the failure is indicated by an example of a list which is not equal to its reverse.
- What if we test without giving a type?
  ```haskell
test prop_rev
```  
  DEMO: Re-Testing the Single Reverse Property
- The test succeeds ! How come ?

Generic Properties (I)

- Sometimes we want a property that we can use with a variety of operators.
- Consider the property of commutativity — an operator $\oplus$ is commutative if $x \oplus y = y \oplus x$
- We can define the property as a function of the operator:
  ```haskell
pOpComm op x y = x 'op' y == y 'op' x
```  
- To test it we need to supply the operator, and sometimes also type information:
  ```haskell
test (pOpComm (+))
```  
  DEMO: Testing commutativity of $+$

Generic Properties (II)

- Consider operator associativity:
  ```haskell
pOpAssoc op x y z = x 'op' (y 'op' z) == (x 'op' y) 'op' z
```  
- To test if list concatenation is associative
  ```haskell
  ((++) :: [a] -> [a] -> [a])
```  
  we need to type the following:
  ```haskell
test (pOpAssoc (++) :: [Int] -> [Int] -> [Int] -> Bool)
```  
  DEMO: Testing associativity of $++$
QuickCheck: Class Arbitrary

- QuickCheck defines a class called `Arbitrary` that defines two operators for generating random values of a type:
  ```haskell
class Arbitrary a where
    arbitrary :: Gen a
    coarbitrary :: a -> Gen b -> Gen b
  ```
- The type `Gen a` denotes something capable of generating random values of a given type — by a mechanism we shall ignore for now.
- We shall also ignore the `coarbitrary` function, whose role is quite technical.

Type Constructor Gen

- `choose :: Random a => (a,a) -> Gen a`
  - Given type `a` an instance of the standard `Random` class, return a random value within bound of first argument
- `elements :: [a] -> Gen a`
  - Pick a value at random from a list
- `vector :: Arbitrary a => Int -> Gen [a]`
  - Generate a list of arbitrary elements of given length
- `oneof :: [Gen a] -> Gen a`
  - Pick a generator at random from a list, and use that
- `sized :: (Int -> Gen a) -> Gen a`
  - Takes a generator depending on an integer and generates that integer internally (starts small and gets larger over time).

Some Arbitrary Instances

- `instance Arbitrary Bool where`
  - `arbitrary = elements [True, False]`
- `instance Arbitrary Int where`
  - `arbitrary = sized \n -> choose (-n,n)`
- `instance Arbitrary a => Arbitrary [a] where`
  - `arbitrary = sized \n -> choose (0,n) >>= vector`
- This can be re-written using do-notation:
  ```haskell
  instance Arbitrary a => Arbitrary [a] where
  arbitrary = sized mkvector
  mkvector n = do s <- choose (0,n)
                  vector s
  ```
- Yes - you’ve guessed, the generation of the random numbers and the steady increase in `sized` values is managed by a (state) monad.

Function test

- `Function test :: Testable a => a -> IO()` allow us to run tests on any type belonging to the `Testable` class.
- `class Testable a where`
  - `property :: a -> Property`
- We have instances of `Testable` for `Bool`, `Property`, and for `a -> b` where `b` is an instance of `Testable`, and `a` is an instance of `Arbitrary` and `Show`.
- This latter case means we automatically get testable instances of functions of type `a1 -> a2 -> . . . . . -> ab -> Bool`, as long as the `ai` can be generated arbitrarily.
What is type \textit{Property}?

- A problem arises when testing implications:
  \[ P \implies Q \]

  This is always true when \( P \) is false, so benefit \textit{only} arises from test-cases where \( P \) does not hold.

- We have a variation of implication that does not count the case where antecedent fails:
  \((\Rightarrow) :: \text{Testable} \; a \rightarrow \text{Bool} \rightarrow a \rightarrow \text{Property}\)

The \textit{Property} type is best viewed as a special form of \textit{Bool}.

- QuickCheck testing of \textit{Property} looks for 100 cases where antecedent is true, but gives up after 10,000 attempts.

Testing Boolean Implication

- We define our own implication
  \[
  \text{implies} \; \text{False} \_ = \text{True} \\
  \text{implies} \; \text{True} \; q = q
  \]

- We build a test for property \( n = m^2 \implies n \cdot m = m^3 \)
  \[
  \text{prop\_weak} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \\
  \text{prop\_weak} \; n \; m = (n == m*m) \implies (n*m == m*m*m)
  \]

  \text{DEMO: Testing prop\_weak (verbosely)}

Testing Property Implication

- We build a test for property \( n = m^2 \implies n \cdot m = m^3 \) using the property implication
  \[
  \text{prop\_better} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Property} \\
  \text{prop\_better} \; n \; m = (n == m*m) \implies (n*m == m*m*m)
  \]

  \text{DEMO: Testing prop\_better (verbosely)}

- We see that we only count tests where \( n \) is actually the square of \( m \).

- We also see that we do not get 100 good tests

- Also, most successful tests involve \( m = n \in \{0, 1\} \)

- Clearly there must be a better way to test this property?

CS4011 Project 2010

- Objective: implement, in Haskell, a program that plays the game “minesweeper”.

- Really plays it. Without the user needing to make any moves.

- This is a rather more interesting problem than it first looks: Kaye 2000 shows that minesweeper is \text{NP}-Complete\footnote{He encodes logic circuits into minesweeper positions such that determining whether the position is consistent is equivalent to solving the proposition the circuit encodes. Neat!}

- Involves GUI design and implementation

- Start Date: now

- Deadline: Monday December 13

\footnote{see http://web.mat.bham.ac.uk/R.W.Kaye/minesw/}
**Background**

- Minesweeper is a puzzle game. A grid contains a number of hidden mines (the exact density determines the difficulty of the game, a 16x16 with 40 mines might be typical). The player moves by either:
  - Uncovering a square, to reveal:
    - A mine (game over!)
    - No mine (great!), then shows the count of mines in the 8 adjacent squares.
  - Flagging a square (player believes it contains a mine)

**Project Task**

- Necessary (but not sufficient)
  - Distribute mines randomly around the grid
  - Allow the user to uncover and flag mines interactively
  - Detect the endgame
  - Provide QuickCheck properties to test your program

- Sufficient:
  - Whenever the user requests it, play one move automatically, if there is at least one unambiguously safe move available

- Wonderful:
  - If there is *not* a safe move available (this happens sometimes), determine the probability of a mine being present, and report the *safest* squares to uncover.

**Resources, hints, help**

- We will see how to provide some (basic, at least) GUI features, but this project is really about algorithms and datastructures.
- There are some good online resources that discuss how to approach the solver (see links on the course web page).
- We will devote one lab to a “project clinic” (closer to the handin date).