# COMP3141

## Software System Design and Implementation

# Property Based Testing; Lazy Evaluation

Liam O'Connor
CSE, UNSW (and Data61)
Term 2 2019

# Free Properties

Haskell already ensures certain properties automatically with its language design and type system.

1. Memory is accessed where and when it is safe and permitted to be accessed (*memory safety*).

# Free Properties

Haskell already ensures certain properties automatically with its language design and type system.

1. Memory is accessed where and when it is safe and permitted to be accessed (*memory safety*).
2. Values of a certain static type will actually have that type at run time.

# Free Properties

Haskell already ensures certain properties automatically with its language design and type system.

1. Memory is accessed where and when it is safe and permitted to be accessed (*memory safety*).
2. Values of a certain static type will actually have that type at run time.
3. Programs that are well-typed will not lead to undefined behaviour (*type safety*).

# Free Properties

Haskell already ensures certain properties automatically with its language design and type system.

1. Memory is accessed where and when it is safe and permitted to be accessed (*memory safety*).
2. Values of a certain static type will actually have that type at run time.
3. Programs that are well-typed will not lead to undefined behaviour (*type safety*).
4. All functions are *pure*: Programs won't have side effects not declared in the type. (*purely functional programming*)

# Free Properties

Haskell already ensures certain properties automatically with its language design and type system.

1. Memory is accessed where and when it is safe and permitted to be accessed (*memory safety*).
2. Values of a certain static type will actually have that type at run time.
3. Programs that are well-typed will not lead to undefined behaviour (*type safety*).
4. All functions are *pure*: Programs won't have side effects not declared in the type. (*purely functional programming*)

$\Rightarrow$ Most of our properties focus on the *logic of our program*.

# Logical Properties

We have already seen a few examples of logical properties.

# Logical Properties

We have already seen a few examples of logical properties.

**Example (Properties)**

1. `reverse` is an *involution*: `reverse (reverse xs) == xs`
2. right identity for `(++)`: `xs ++ [] == xs`
3. transitivity of `(>)`: `(a > b)` $\wedge$ `(b > c)` $\Rightarrow$ `(a > c)`

# Logical Properties

We have already seen a few examples of logical properties.

**Example (Properties)**

1. `reverse` is an *involution*: `reverse (reverse xs) == xs`
2. right identity for `(++)`: `xs ++ [] == xs`
3. transitivity of `(>)`: `(a > b)` $\wedge$ `(b > c)` $\Rightarrow$ `(a > c)`

The set of properties that capture all of our requirements for our program is called the *functional correctness specification* of our software.

This defines what it means for software to be correct.

# Proofs

Last week we saw some *proof methods* for Haskell programs. We could prove that our implementation meets its functional correctness specification.

# Proofs

Last week we saw some *proof methods* for Haskell programs. We could prove that our implementation meets its functional correctness specification.

Such proofs certainly offer a high degree of assurance, but:

- Proofs must make some assumptions about the environment and the semantics of the software.

# Proofs

Last week we saw some *proof methods* for Haskell programs. We could prove that our implementation meets its functional correctness specification.

Such proofs certainly offer a high degree of assurance, but:

- Proofs must make some assumptions about the environment and the semantics of the software.
- Proof complexity grows with implementation complexity, sometimes drastically.

# Proofs

Last week we saw some *proof methods* for Haskell programs. We could prove that our implementation meets its functional correctness specification.

Such proofs certainly offer a high degree of assurance, but:

- Proofs must make some assumptions about the environment and the semantics of the software.
- Proof complexity grows with implementation complexity, sometimes drastically.
- If software is incorrect, a proof attempt might simply become stuck: we do not always get constructive negative feedback.

# Proofs

Last week we saw some *proof methods* for Haskell programs. We could prove that our implementation meets its functional correctness specification.

Such proofs certainly offer a high degree of assurance, but:

- Proofs must make some assumptions about the environment and the semantics of the software.
- Proof complexity grows with implementation complexity, sometimes drastically.
- If software is incorrect, a proof attempt might simply become stuck: we do not always get constructive negative feedback.
- Proofs can be labour and time intensive ($$$), or require highly specialised knowledge ($$$).

# Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language semantics or operating environment.

# Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language semantics or operating environment.
- Test complexity does not grow with implementation complexity, so long as the specification is unchanged.

# Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language semantics or operating environment.
- Test complexity does not grow with implementation complexity, so long as the specification is unchanged.
- Incorrect software when tested leads to immediate, debuggable counterexamples.

# Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language semantics or operating environment.
- Test complexity does not grow with implementation complexity, so long as the specification is unchanged.
- Incorrect software when tested leads to immediate, debuggable counterexamples.
- Testing is typically cheaper and faster than proving.

# Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language semantics or operating environment.
- Test complexity does not grow with implementation complexity, so long as the specification is unchanged.
- Incorrect software when tested leads to immediate, debuggable counterexamples.
- Testing is typically cheaper and faster than proving.
- Tests care about efficiency and computability, unlike proofs.

# Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language semantics or operating environment.
- Test complexity does not grow with implementation complexity, so long as the specification is unchanged.
- Incorrect software when tested leads to immediate, debuggable counterexamples.
- Testing is typically cheaper and faster than proving.
- Tests care about efficiency and computability, unlike proofs.

We lose some assurance, but gain some convenience ($$$).

# Property Based Testing

**Key idea**: Generate random input values, and test properties by running them.

**Example (QuickCheck Property)**

```
prop_reverseApp xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs
```

# Property Based Testing

**Key idea**: Generate random input values, and test properties by running them.

**Example (QuickCheck Property)**

```
prop_reverseApp xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Haskell's *QuickCheck* is the first library ever invented for property-based testing. The concept has since been ported to Erlang, Scheme, Common Lisp, Perl, Python, Ruby, Java, Scala, F#, OCaml, Standard ML, C and C++.

# PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  $\Rightarrow$ Less testing code

# PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  $\Rightarrow$ Less testing code
- Property-based testing heavily depends on test data generation:

# PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  $\Rightarrow$ Less testing code
- Property-based testing heavily depends on test data generation:
  - Random inputs may not be as informative as hand-crafted inputs
    $\Rightarrow$ use shrinking

# PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  $\Rightarrow$ Less testing code
- Property-based testing heavily depends on test data generation:
  - Random inputs may not be as informative as hand-crafted inputs
    $\Rightarrow$ use shrinking
  - Random inputs may not cover all necessary corner cases:
    $\Rightarrow$ use a coverage checker

# PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  $\Rightarrow$ Less testing code
- Property-based testing heavily depends on test data generation:
  - Random inputs may not be as informative as hand-crafted inputs
    $\Rightarrow$ use shrinking
  - Random inputs may not cover all necessary corner cases:
    $\Rightarrow$ use a coverage checker
  - Random inputs must be generated for user-defined types:
    $\Rightarrow$ QuickCheck includes functions to build custom generators

# PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  $\Rightarrow$ Less testing code
- Property-based testing heavily depends on test data generation:
  - Random inputs may not be as informative as hand-crafted inputs
    $\Rightarrow$ use shrinking
  - Random inputs may not cover all necessary corner cases:
    $\Rightarrow$ use a coverage checker
  - Random inputs must be generated for user-defined types:
    $\Rightarrow$ QuickCheck includes functions to build custom generators
- By increasing the number of random inputs, we improve code coverage in PBT.

# Test Data Generation

Data which can be generated randomly is represented by the following type class:

```
class Arbitrary a where
  arbitrary :: Gen a  -- more on this later
  shrink :: a -> [a]
```

Most of the types we have seen so far implement `Arbitrary`.

# Test Data Generation

Data which can be generated randomly is represented by the following type class:

```
class Arbitrary a where
  arbitrary :: Gen a  -- more on this later
  shrink :: a -> [a]
```

Most of the types we have seen so far implement `Arbitrary`.

**Shrinking**

The `shrink` function is for when test cases fail. If a given input `x` fails, QuickCheck will try all inputs in `shrink x`; repeating the process until the smallest possible input is found.

# Testable Types

The type of the `quickCheck` function is:

```
                                    -- more on IO later
quickCheck :: (Testable a) => a -> IO ()
```

# Testable Types

The type of the `quickCheck` function is:

```
                                        -- more on IO later
quickCheck :: (Testable a) => a -> IO ()
```

The `Testable` type class is the class of things that can be converted into properties.

# Testable Types

The type of the `quickCheck` function is:

```
                                    -- more on IO later
quickCheck :: (Testable a) => a -> IO ()
```

The `Testable` type class is the class of things that can be converted into properties. This includes:

- `Bool` values
- QuickCheck's built-in `Property` type
- Any function from an `Arbitrary` input to a `Testable` output:

  ```
  instance (Arbitrary i, Testable o)
        => Testable (i -> o) ...
  ```

# Testable Types

The type of the `quickCheck` function is:

```
                                    -- more on IO later
quickCheck :: (Testable a) => a -> IO ()
```

The `Testable` type class is the class of things that can be converted into properties. This includes:

- `Bool` values
- QuickCheck's built-in `Property` type
- Any function from an `Arbitrary` input to a `Testable` output:

```
instance (Arbitrary i, Testable o)
      => Testable (i -> o) ...
```

Thus the type `[Int] -> [Int] -> Bool` (as used earlier) is `Testable`.

# Simple example

Is this function reflexive?

```
divisible :: Integer -> Integer -> Bool
divisible x y = x `mod` y == 0

prop_refl :: Integer -> Bool
prop_refl x = divisible x x
```

# Simple example

Is this function reflexive?

```
divisible :: Integer -> Integer -> Bool
divisible x y = x `mod` y == 0

prop_refl :: Integer -> Bool
prop_refl x = divisible x x
```

- Encode pre-conditions with the (==>) operator:

  ```
  prop_refl :: Integer -> Property
  prop_refl x = x > 0 ==> divisible x x
  ```

  (but may generate a lot of spurious cases)

## Simple example

Is this function reflexive?

```haskell
divisible :: Integer -> Integer -> Bool
divisible x y = x `mod` y == 0

prop_refl :: Integer -> Bool
prop_refl x = divisible x x
```

- Encode pre-conditions with the (==>) operator:

  ```haskell
  prop_refl :: Integer -> Property
  prop_refl x = x > 0 ==> divisible x x
  ```

  (but may generate a lot of spurious cases)

- or select different generators with modifier `newtypes`.

  ```haskell
  prop_refl :: Positive Integer -> Bool
  prop_refl (Positive x) = divisible x x
  ```

  (but may require you to define custom generators)

# Words and Inverses

**Example (Inverses)**

```
words   :: String -> [String]
unwords :: [String] -> String
```

# Words and Inverses

**Example (Inverses)**

```haskell
words   :: String -> [String]
unwords :: [String] -> String
```

We might expect `unwords` to be the inverse of `words` and vice versa. Let's find out!

# Words and Inverses

**Example (Inverses)**

```
words   :: String -> [String]
unwords :: [String] -> String
```

We might expect `unwords` to be the inverse of `words` and vice versa. Let's find out!

**Lessons**: Properties aren't always what you expect!

# Merge Sort

**Example (Merge Sort)**

Recall merge sort, the sorting algorithm that is reliably $\mathcal{O}(n \log n)$ time complexity.

- If the list is empty or one element, return that list.
- Otherwise, we:
  1. Split the input list into two sublists.
  2. Recursively sort the two sublists.
  3. Merge the two sorted sublists into one sorted list in linear time.

Applying our bottom up design, let's posit:

```
split :: [a] -> ([a],[a])
merge :: (Ord a) => [a] -> [a] -> [a]
```

# Split

```
split :: [a] -> ([a],[a])
```

What is a good specification of split?

# Split

```haskell
split :: [a] -> ([a],[a])
```

What is a good specification of split?

- Each element of the input list occurs in one of the two output lists, the same number of times.

# Split

```haskell
split :: [a] -> ([a],[a])
```

What is a good specification of split?

- Each element of the input list occurs in one of the two output lists, the same number of times.
- The two output lists consist only of elements from the input list.

# Split

```haskell
split :: [a] -> ([a],[a])
```

What is a good specification of split?

- Each element of the input list occurs in one of the two output lists, the same number of times.
- The two output lists consist only of elements from the input list.

Because of its usefulness later, we'll define this in terms of a permutation predicate.

# Merge

```haskell
merge :: (Ord a) => [a] -> [a] -> [a]
```

What is a good specification of merge?

# Merge

```haskell
merge :: (Ord a) => [a] -> [a] -> [a]
```

What is a good specification of merge?

- Each element of the output list occurs in one of the two input lists, the same number of times.

# Merge

```haskell
merge :: (Ord a) => [a] -> [a] -> [a]
```

What is a good specification of merge?

- Each element of the output list occurs in one of the two input lists, the same number of times.
- The two input lists consist solely of elements from the output list.

# Merge

```
merge :: (Ord a) => [a] -> [a] -> [a]
```

What is a good specification of merge?

- Each element of the output list occurs in one of the two input lists, the same number of times.
- The two input lists consist solely of elements from the output list.
- **Important:** If the input lists are sorted, then the output list is sorted.

# Overall

```haskell
mergesort :: (Ord a) => [a] -> [a]
```

What is a good specification of mergesort?

# Overall

```
mergesort :: (Ord a) => [a] -> [a]
```

What is a good specification of mergesort?

- The output list is sorted.

# Overall

```haskell
mergesort :: (Ord a) => [a] -> [a]
```

What is a good specification of mergesort?

- The output list is sorted.
- The output list is a permutation of the input list.

# Overall

```haskell
mergesort :: (Ord a) => [a] -> [a]
```

What is a good specification of mergesort?

- The output list is sorted.
- The output list is a permutation of the input list.

We can prove this as a consequence of the previous specifications which we tested. Do this if time permits.
We can also just write integration properties that test the composition of these functions together. Also do this if time permits.

# Redundant Properties

Some properties are technically redundant (i.e. implied by other properties in the specification), but there is some value in testing them anyway:

- They may be more efficient than full functional correctness tests, consuming less computing resources to test.

# Redundant Properties

Some properties are technically redundant (i.e. implied by other properties in the specification), but there is some value in testing them anyway:

- They may be more efficient than full functional correctness tests, consuming less computing resources to test.
- They may be more fine-grained to give better test coverage than random inputs for full functional correctness tests.

# Redundant Properties

Some properties are technically redundant (i.e. implied by other properties in the specification), but there is some value in testing them anyway:

- They may be more efficient than full functional correctness tests, consuming less computing resources to test.
- They may be more fine-grained to give better test coverage than random inputs for full functional correctness tests.
- They provide a good sanity check to the full functional correctness properties.

# Redundant Properties

Some properties are technically redundant (i.e. implied by other properties in the specification), but there is some value in testing them anyway:

- They may be more efficient than full functional correctness tests, consuming less computing resources to test.
- They may be more fine-grained to give better test coverage than random inputs for full functional correctness tests.
- They provide a good sanity check to the full functional correctness properties.
- Sometimes full functional correctness is not easily computable but tests of weaker properties are.

# Redundant Properties

Some properties are technically redundant (i.e. implied by other properties in the specification), but there is some value in testing them anyway:

- They may be more efficient than full functional correctness tests, consuming less computing resources to test.
- They may be more fine-grained to give better test coverage than random inputs for full functional correctness tests.
- They provide a good sanity check to the full functional correctness properties.
- Sometimes full functional correctness is not easily computable but tests of weaker properties are.

These redundant properties include unit tests. We can (and should) combine both approaches!

# Redundant Properties

Some properties are technically redundant (i.e. implied by other properties in the specification), but there is some value in testing them anyway:

- They may be more efficient than full functional correctness tests, consuming less computing resources to test.
- They may be more fine-grained to give better test coverage than random inputs for full functional correctness tests.
- They provide a good sanity check to the full functional correctness properties.
- Sometimes full functional correctness is not easily computable but tests of weaker properties are.

These redundant properties include unit tests. We can (and should) combine both approaches!

What are some redundant properties of `mergesort`?

# Test Quality

How good are your tests?

- Have you checked that every special case works correctly?
- Is all code exercised in the tests?
- Even if all code is exercised, is it exercised in all contexts?

Coverage checkers are useful tools to partially quantify this.

# Types of Coverage

**Function Coverage**
All functions executed?

# Types of Coverage

**Function Coverage**
All functions executed?

**Entry/Exit Coverage**
All function calls executed?

# Types of Coverage

**Entry/Exit Coverage**
All function calls executed?

**Function Coverage**
All functions executed?

**Statement/Expression Coverage**
All expressions executed?

# Types of Coverage

**Branch/Decision Coverage**
All conditional branches executed?

**Entry/Exit Coverage**
All function calls executed?

**Function Coverage**
All functions executed?

**Statement/Expression Coverage**
All expressions executed?

# Types of Coverage

**Branch/Decision Coverage**
All conditional branches executed?

**Entry/Exit Coverage**
All function calls executed?

**Function Coverage**
All functions executed?

**Statement/Expression Coverage**
All expressions executed?

**Path Coverage**
All behaviours executed?
**very hard!**

# Haskell Program Coverage

Haskell Program Coverage (or `hpc`) is a GHC-bundled tool to measure function, branch and expression coverage.
Let's try it out!

**For Stack:** Build with the `--coverage` flag, execute binary, produce visualisations with `stack hpc report`.

**For Cabal:** Build with the `--enable-coverage` flag, execute binary, produce visualisations with `hpc report`.

# Sum to $n$

```haskell
sumTo :: Integer -> Integer
sumTo 0 = 0
sumTo n = sumTo (n-1) + n
```

This crashes when given a large number. Why?

# Sum to $n$, redux

```
sumTo' :: Integer -> Integer -> Integer
sumTo' a 0 = a
sumTo' a n = sumTo' (a+n) (n-1)

sumTo = sumTo' 0
```

This still crashes when given a large number. Why?

# Sum to $n$, redux

```
sumTo' :: Integer -> Integer -> Integer
sumTo' a 0 = a
sumTo' a n = sumTo' (a+n) (n-1)

sumTo = sumTo' 0
```

This still crashes when given a large number. Why?

This is called a space leak, and is one of the main drawbacks of Haskell's lazy evaluation method.

# Lazy Evaluation

Haskell is lazily evaluated, also called call-by-need.
This means that expressions are only evaluated when they are needed to compute a result for the user.

# Lazy Evaluation

Haskell is lazily evaluated, also called call-by-need.
This means that expressions are only evaluated when they are needed to compute a result for the user.
We can force the previous program to evaluate its accumulator by using a bang pattern, or the primitive operation seq:

```haskell
sumTo' :: Integer -> Integer -> Integer
sumTo' !a 0 = a
sumTo' !a n = sumTo' (a+n) (n-1)
```

```haskell
sumTo' :: Integer -> Integer -> Integer
sumTo' a 0 = a
sumTo' a n = let a' = a + n in a' `seq` sumTo' a' (n-1)
```

# Advantages

Lazy Evaluation has many advantages:

- It enables equational reasoning even in the presence of partial functions and non-termination.

[1] J. Hughes, "Why Functional Programming Matters", Comp. J., 1989

# Advantages

Lazy Evaluation has many advantages:

- It enables equational reasoning even in the presence of partial functions and non-termination.
- It allows functions to be decomposed without sacrificing efficiency, for example: `minimum = head . sort` is, depending on sorting algorithm, possibly $\mathcal{O}(n)$. John Hughes demonstrates $\alpha\beta$ pruning from AI as a larger example.[1]

[1] J. Hughes, "Why Functional Programming Matters", Comp. J., 1989

# Advantages

Lazy Evaluation has many advantages:

- It enables equational reasoning even in the presence of partial functions and non-termination.
- It allows functions to be decomposed without sacrificing efficiency, for example: `minimum = head . sort` is, depending on sorting algorithm, possibly $\mathcal{O}(n)$. John Hughes demonstrates $\alpha\beta$ pruning from AI as a larger example.[1]
- It allows for circular programming and infinite data structures, which allow us to express more things as pure functions.

**Problem**

In one pass over a list, replace every element of the list with its maximum.

[1] J. Hughes, "Why Functional Programming Matters", Comp. J., 1989

# Infinite Data Structures

Laziness lets us define data structures that extend infinitely. Lists are a common example, but it also applies to trees or any user-defined data type:

```
ones = 1 : ones
```

# Infinite Data Structures

Laziness lets us define data structures that extend infinitely. Lists are a common example, but it also applies to trees or any user-defined data type:

```
ones = 1 : ones
```

Many functions such as `take`, `drop`, `head`, `tail`, `filter` and `map` work fine on infinite lists!

# Infinite Data Structures

Laziness lets us define data structures that extend infinitely. Lists are a common example, but it also applies to trees or any user-defined data type:

```
ones = 1 : ones
```

Many functions such as `take`, `drop`, `head`, `tail`, `filter` and `map` work fine on infinite lists!

```
naturals = 0 : map (1+) naturals
--or
naturals = map sum (inits ones)
```

# Infinite Data Structures

Laziness lets us define data structures that extend infinitely. Lists are a common example, but it also applies to trees or any user-defined data type:

```
ones = 1 : ones
```

Many functions such as `take`, `drop`, `head`, `tail`, `filter` and `map` work fine on infinite lists!

```
naturals = 0 : map (1+) naturals
--or
naturals = map sum (inits ones)
```

How about fibonacci numbers?

# Infinite Data Structures

Laziness lets us define data structures that extend infinitely. Lists are a common example, but it also applies to trees or any user-defined data type:

```
ones = 1 : ones
```

Many functions such as `take`, `drop`, `head`, `tail`, `filter` and `map` work fine on infinite lists!

```
naturals = 0 : map (1+) naturals
--or
naturals = map sum (inits ones)
```

How about fibonacci numbers?

```
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

# Homework

1. Next programming exercise is due in a week.
2. Last week's quiz is due this Friday. Make sure you submit your answers.
3. This week's quiz is also up, due the following Friday.