COMP3141

Liam O'Conn
University of Edinburgh LFCS (an
Term 2 2020

Free Properties

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

1. Memory is a
   (*memory s*

Free Properties

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

1. Memory is a
   (*memory s*

2. Values of a ce

Free Properties

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

1. Memory is a
   (*memory s*

2. Values of a ce
3. Programs that are well-typed will not lead to undefined b *ype safety*).

Free Properties

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

1. Memory is a (*memory s*

2. Values of a ce
3. Programs that are well-typed will not lead to undefined b *ype safety*).

4. All functions are *pure*: Programs won't have side eff (*purely functional programming*)

Free Properties

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

1. Memory is a
   (*memory s*

2. Values of a ce
3. Programs that are well-typed will not lead to undefined b *ype safety*).

4. All functions are *pure*: Programs won't have side eff
   (*purely functional programming*)

⇒ 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 (Prop

1. `reverse`
2. right identi
3. transitivity of `(>)`: `(a > b)` $\wedge$ `(b > c)` $\Rightarrow$

Logical Properties

We have already seen a few examples of logical properties.

Example (Prop

1. `reverse`
2. right identi
3. transitivity of `(>)`: `(a > b)` $\wedge$ `(b > c)` $\Rightarrow$

The set of properties that capture all of our requirements for our p *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 certai

- Proofs mus f the softwar

Proofs

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

Such proofs certai

- Proofs mus f the softwar

- Proof complexity grows with implementation compl .

Proofs

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

Such proofs certai

- Proofs mus f the softwar
- Proof complexity grows with implementation compl .
- If software is *incorrect*, a proof attempt might simply bec 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 certai

- Proofs mu f the softwar
- Proof complexity grows with implementation compl .
- If software is incorrect, a proof attempt might simply bec 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 se

Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language se
- Test comp
  specificati

Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language se
- Test comp specificati
- Incorrect software when tested leads to immediate, de

Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language se
- Test comp
  specificati

- Incorrect software when tested leads to immediate, de
- 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 se
- Test comp specificati
- Incorrect software when tested leads to immediate, de
- Testing is typically cheaper and faster than proving.
- Tests care about efficiency and computability, unlike

Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language se
- Test comp
  specificati
- Incorrect software when tested leads to immediate, de
- Testing is typically cheaper and faster than proving.
- Tests care about efficiency and computability, unlike

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

Property Based Testing

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

Example (Quic

```
prop_revers
    reverse (
```

Property Based Testing

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

Example (Quic

```
prop_revers
    reverse (
```


Haskell's *QuickCheck* is the first library ever invented for p concept has since been ported to Erlang, Scheme, Common Li Java, Scala, F#, OCaml, Standard ML, C and C++.

PBT vs. Unit Testing

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

PBT vs. Unit Testing

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

PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  ⇒ Less testing code
- Property-
  - Rand
    ⇒ us

PBT vs. Unit Testing

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

  - Rand
    $\Rightarrow$ us
  - Random inputs may not cover all necessary corner case
    $\Rightarrow$ use a coverage checker

PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  ⇒ Less testing code
- Property-
  - Rand
    ⇒ us

  - Random inputs may not cover all necessary corner case
    ⇒ use a coverage checker
  - Random inputs must be generated for user defined ty
    ⇒ QuickCheck includes functions to build custom g

PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  ⇒ Less testing code
- Property-
  - Rand
    ⇒ us

  - Random inputs may not cover all necessary corner case
    ⇒ use a coverage checker
  - Random inputs must be generated for user defined ty
    ⇒ QuickCheck includes functions to build custom g

- 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 :
  shrink :: a -> [a
```


Most of the types we

Test Data Generation

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

```
class Arbitrary a where
  arbitrary :
  shrink :: a -> [a
```


Most of the types we

Shrinking

The `shrink` function is for when test cases fail. If a given input ickCheck will try all inputs in `shrink x`; repeating the process u 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:


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

The Testable

Testable Types

The type of the quickCheck function is:


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

The Testable

This includes:

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


```
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

This includes:

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


```
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 :: In
prop_refl x = div
```

Simple example

Is this function reflexive?

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

prop_refl :: In
prop_refl x = div
```

- Encode pre

```
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 :: In
prop_refl x = div
```


- Encode pre

  ```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 -> [S
unwords :: [Str
```

Words and Inverses

Example (Inverses)

```
words   :: String -> [S
unwords :: [Str
```


We might expect unwords to be the inverse of ut!

Words and Inverses

Example (Inverses)

```
words   :: String -> [S
unwords :: [Str
```


We might expect `unwords` to be the inverse of ut!

Lessons: Properties aren't always what you expect!

Merge Sort

Example (Merge Sort)

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

- If the list is emp
- Otherwise
  1. Split th
  2. Recursively sort the two sublists.
  3. Merge the two sorted sublists into one sorted list in linear t

Applying our bottom up design, let's posit:

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

Split


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

What is a good speci

Split


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

What is a good speci

- Each eleme
  number of times.

Split


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

What is a good speci

- Each eleme
  number of times.
- The two output lists consist only of elements from the inpu

Split


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

What is a good speci

- Each eleme
number of times.
- The two output lists consist only of elements from the inpu

Because of its usefulness later, we'll define this in terms of a permu

Merge


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

What is a good speci

Merge


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

What is a good speci

- Each eleme
  number of times.

Merge


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

What is a good speci

- Each eleme number of times.
- The two input lists consist solely of elements from the outp

Merge


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

What is a good speci

- Each eleme
  number of times.
- The two input lists consist solely of elements from the outp
- **Important:** If the input lists are sorted, then the outpu

Overall


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

What is a good speci

Overall


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

What is a good speci

- The output li

Overall


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

What is a good speci

- The output li
- The output li

Overall


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

What is a good speci

- The output li
- The output li

We can prove this as a consequence of the previous specificatio
We can also just write integration properties that test the comp functions together.

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 computin

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 computin
- They may be
  for full functi

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 computin
- They may be
  for full functi

- They provide a good sanity check to the full functional corr

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 computin
- They may be for full functi

- They provide a good sanity check to the full functional corr
- Sometimes full functional correctness is not easily com 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 computin
- They may be

  for full functi
- They provide a good sanity check to the full functional corr
- Sometimes full functional correctness is not easily com

  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 computin
- They may be for full functi

- They provide a good sanity check to the full functional corr
- Sometimes full functional correctness is not easily com 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 t

- Have you ch
- Is all code exer
- Even if all code is exercised, is it exercised in all contexts?

Coverage checkers are useful tools to partially quantify this.

Types of Coverage

Func

All functions executed?

Types of Coverage

Func

All functions executed?

Types of Coverage

Func

All functions executed?

Statement Cov

All expressions executed?

Types of Coverage

Branch/Decision Coverage

All conditional branches executed?

Func

All functions executed?

Statement Cov

All expressions executed?

Types of Coverage

Branch/Decision Coverage
All conditional branches executed?

Func
All functions executed?

Statement Cov
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


```
sumTo :: Intege
sumTo 0 = 0
sumTo n = sumTo (n-
```


This crashes when given a large number. Why?

Sum to n, redux


```haskell
sumTo' :: Integer -> Integer -> Integer
sumTo' a 0 = a
sumTo' a n = sumTo' (

sumTo = sumTo' 0
```


This still crashes when given a large number. Why?

Sum to n, redux


```haskell
sumTo' :: Integer -> Integer -> Integer
sumTo' a 0 = a
sumTo' a n = sumTo' (

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 pre

pattern, or the pri


```
sumTo' :: Integ
sumTo' !a 0 = a
sumTo' !a n = sumTo' (a+n) (n-1)
```



```
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 fun ... :
  `minimum = he` ... $\mathcal{O}(n)$.[1]
  John Hugh

[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 fun :
  `minimum` = he $\mathcal{O}(n)$.
  John Hugh [1]

- It allows for circular programming and infinite data stru
  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 s [illegible] ork 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 s[illegible]ork 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:

```haskell
ones = 1 : ones
```


Many functions s[illegible] ork fine on infinite lists!

```haskell
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 s[illegible]ork 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. First progr
2. Second exe
3. Last week's
4. This week's quiz is also up, due the following Friday.