CS 410/510: Advanced Programming

Lecture 4: Lists, Tests, and Laws

Mark P Jones
Portland State University

Why Study Lists?

_lists are a heavily used data structure in many functional programs_
_special syntax is provided to make programming with lists more convenient_
_lists are a special case / an example of:_
  _an algebraic datatype_
  _A parameterized datatype_
  _A monad_

What is a List?

_an ordered collection (multiset) of values_
_[1,2,3,4], [4,3,2,1], [1,1,2,2,3,3,4,4] are distinct lists of integers_
_A list of type [T] contains zero or more elements of type T_
  _[True, False] :: [Bool]_
  _[1,2,3] :: [Integer]_
  _['a', 'b', 'c'] :: [Char]_
  _[[], [1], [1,2], [1,2,3]] :: [[Integer]]_
_all elements have the same type:_
  _[True, 2, 'c'] is not a valid list_

Naming Convention:

_We often use a simple naming convention:_
  _If a typical value in a list is called x, then a typical list of such values might be called xs (i.e., the plural of x)_
  _... and a list of lists of values called x might be called xss_
  _A simple convention, minimal clutter, and a useful mnemonic_

How do you Make a List?

_The empty list, [], which has type [a] for any (element) type a_
_Enumerations: [e₁, e₂, e₃, e₄]_
_Arithmetic Sequences:_
  _[elem₁ .. elem₃]_
  _[elem₁, elem₂ .. elem₃]_
  _Only works for certain element types: integers, booleans, characters, ..._
  _(omit last element to specify an “infinite list”)_.

Lists ...

... continued:

- Using list **comprehensions**:
  - \[ 2^x + 1 | x \text{ <-} [1,3,7,11] \]

- Using **constructor functions**:
  - \([]\) and \((::)\) ("nil" and "cons")

- Using **prelude/library functions**:
  - ...

---

**Prelude Functions:**

- 

  \( (++) :: [a] \rightarrow [a] \rightarrow [a] \)

  \( \text{reverse} :: [a] \rightarrow [a] \)

  \( \text{take} :: \text{Int} \rightarrow [a] \rightarrow [a] \)

  \( \text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a] \)

  \( \text{takeWhile} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \)

  \( \text{dropWhile} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \)

  \( \text{zip} :: [a] \rightarrow [b] \rightarrow [(a,b)] \)

  \( \text{replicate} :: \text{Int} \rightarrow a \rightarrow [a] \)

  \( \text{iterate} :: (a \rightarrow a) \rightarrow a \rightarrow [a] \)

  \( \text{repeat} :: a \rightarrow [a] \)

  ...

---

**map:**

- \( \text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \)

- \( \text{map } f \text{ } x\text{ } s \text{ } \text{produces a new list by applying} \)
  \( \text{the function } f \text{ to each element in the list } x\text{s} \)

- \( \text{map } (1+) [1,2,3] = [2,3,4] \)

- \( \text{map } \text{even} [1,2,3] = \text{[False, True, False]} \)

- \( \text{map } \text{id} \text{ } x\text{s} = x\text{s}, \text{for any list } x\text{s} \)

- We can also think of \( \text{map} \) as a function that turns functions of type \((a \rightarrow b)\)

  into list transformers of type \(([a] \rightarrow [b])\)

---

**filter:**

- \( \text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \)

- \( \text{filter } \text{even} \text{ } [1..10] = [2,4,6,8,10] \)

- \( \text{filter } (\textless 5) \text{ } [1..100] = [1,2,3,4] \)

- \( \text{filter } (\textless 5) \text{ } [100,99..1] = [4,3,2,1] \)

- We can think of \( \text{filter} \) as mapping premises/functions of type \((a \rightarrow \text{Bool})\), to

  list transformers of type \([a] \rightarrow [a] \)

---

**Testing:**

- Testing can confirm expectations about how

  things work

- Conversely, testing can set expectations about

  how things should work

- It can be dangerous to generalize from tests

  "Testing can be used to show the presence of bugs, but

  never to show their absence" [Edsger Dijkstra, 1969]

- But testing does help us to find & avoid:

  - Bugs in the things we build

  - Bugs in the claims we make about those things

---

... Tests ...

---
Making Tests Executable:

test1 = filter even [1..10] == [2,4,6,8,10]
test2 = filter (<5) [1..100] == [1,2,3,4]
test3 = filter (<5) [100,99..1] == [4,3,2,1]

tests = test1 && test2 && test3

Making Tests Executable:

test1 = filter even [1..10] == [2,4,6,8,10]
test2 = filter (<5) [1..100] == [1,2,3,4]
test3 = filter (<5) [100,99..1] == [4,3,2,1]

tests = and [test1, test2, test3]

Making Tests Executable:

test1 = filter even [1..10] == [2,4,6,8,10]
test2 = filter (<5) [1..100] == [1,2,3,4]
test3 = filter (<5) [100,99..1] == [4,3,2,1]

tests = and [test1, test2, test3]

Enter HUnit:

- A library for unit testing
- Written in Haskell
  (Or from http://hackage.haskell.org)
- Built-in to recent versions of Hugs and GHC
- Just “import Test.HUnit” and you’re ready!

Issues:

- Want to see results for all tests
- Text to identify individual tests (especially useful when a test fails)
- Summary statistics
- Handle more complex behavior (e.g., testing code that performs I/O actions)
- Support tests for code that is supposed to fail (e.g., raise an exception)
Define Tests:

```haskell
import Test.HUnit

test1 = testCase (assertEqual
    "filter even [1..10]"
    (filter even [1..10])
    [2,4,6,8,10])

test2 = ...
test3 = ...
tests = TestList [test1, test2, test3]
```

Running Tests:

```haskell
Main> runTestTT tests
Cases: 3 Tried: 3 Errors: 0 Failures: 0
Main>
```

Detecting Faults:

```haskell
import Test.HUnit

test1 = testCase (assertEqual
    "filter even [1..10]"
    (filter even [1..10])
    [2,4,6,8,10])

test2 = ...
test3 = ...
tests = TestList [test1, test2, test3]
```

Using HUnit:

```haskell
Main> runTestTT tests
### Failure in: 0
filter even [1..10]
expected: [2,4,6,8,10]
but got: [2,4,6,9,10]
Cases: 3 Tried: 3 Errors: 0 Failures: 1
Main>
```

Labeling Tests:

```haskell
...

tests = TestLabel "filter tests"
    $ TestList [test1, test2, test3]
```

Using HUnit:

```haskell
Main> runTestTT tests
### Failure in: filter tests:0
filter even [1..10]
expected: [2,4,6,8,10]
but got: [2,4,6,9,10]
Cases: 3 Tried: 3 Errors: 0 Failures: 1
Main>
```
The Test and Assertion Types:

```haskell
data Test = TestCase Assertion
           | TestList [Test]
           | TestLabel String Test
```

```haskell
runTestTT :: Test -> IO Counts
```

```haskell
assertFailure :: String -> Assertion
assertBool :: String -> Bool -> Assertion
assertEqual :: (Eq a, Show a) => String -> a -> a -> Assertion
```

Problems:

- Finding and running tests is a manual process (easily skipped/overlooked)
- Can be hard to trim tests from distributed code
- Can’t solve the halting problem 😞

Example: merge

Let’s develop a `merge` function for combining two sorted lists into a single sorted list:

```haskell
merge :: [Int] -> [Int] -> [Int]
merge = undefined
```

What about test cases?

Merge Tests:

- Simple examples:
  ```haskell```
  ```haskell
  merge [1,5,9] [2,3,6,10] == [1,2,3,5,6,9,10]
  ```
- One or both arguments empty:
  ```haskell```
  ```haskell
  merge [] [1,2,3] == [1,2,3]
  merge [1,2,3] [] == [1,2,3]
  ```
- Duplicate elements:
  ```haskell```
  ```haskell
  merge [2] [1,2,3] == [1,2,3]
  merge [1,2,3] [2] == [1,2,3]
  ```

Capturing the Tests:

```haskell
mergeTests
  = TestLabel "merge tests"
  $ TestList [simpleTests, emptyTests, dupTests]

simpleTests
  = TestLabel "simple tests"
  $ TestCase (assertEqual "merge [1,5,9] [2,3,6,10]"
             (merge [1,5,9] [2,3,6,10])
             [1,2,3,5,6,9,10])
```

```haskell
emptyTests
  = ...
```

Capturing the Tests:

```
Main> runTestTT mergeTests
Cases: 6 Tried: 0 Errors: 0 Failures: 0
Program error: Prelude.undefined
```

```
Main>
```
Refining the Definition (1):

Let’s provide a little more definition for `merge`:

```
merge :: [Int] -> [Int] -> [Int]
merge xs ys = []
```

What happens to the test cases now?

Refining the Definition (2):

Let’s provide a little more definition for `merge`:

```
merge :: [Int] -> [Int] -> [Int]
merge xs ys = xs
```

What happens to the test cases now?

Refining the Definition (3):

Use type information to break the definition down into multiple cases:

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge (x:xs) ys = ys
```

Refining the Definition (4):

Repeat ...

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge (x:xs) [] = x:xs
merge (x:xs) (y:ys) = x:xs
```
Refining the Definition (5):  

Use guards to split into cases:

merge :: [Int] -> [Int] -> [Int]  
merge [] ys = ys  
merge (x:xs) [] = x:xs  
merge (x:xs) (y:ys)  
  | x<y = x : merge xs (y:ys)  
  | otherwise = y : merge (x:xs) ys

Back to the Tests:  

Main> runTestTT mergeTests  
### Failure in: merge tests:2:duplicate elements:0  
merge [2] [1,2,3]  
expected: [1,2,2,3]  
but got: [1,2,3]  
### Failure in: merge tests:2:duplicate elements:1  
merge [1,2,3] [2]  
expected: [1,2,2,3]  
but got: [1,2,3]  
Cases: 6 Tried: 6 Errors: 0 Failures: 2  

Main>

Refining the Definition (6):  

Use another guards to add another case:

merge :: [Int] -> [Int] -> [Int]  
merge [] ys = ys  
merge (x:xs) [] = x:xs  
merge (x:xs) (y:ys)  
  | x<y = x : merge xs (y:ys)  
  | y<x = y : merge (x:xs) ys  
  | x==y = x : merge xs ys

Back to the Tests:  

Main> runTestTT mergeTests  
Cases: 6 Tried: 6 Errors: 0 Failures: 0  

Main>

Modifying the Definition:  

Suppose we decide to modify the definition:

merge :: [Int] -> [Int] -> [Int]  
merge (x:xs) (y:ys)  
  | x<y = x : merge xs (y:ys)  
  | y<x = y : merge (x:xs) ys  
  | x==y = x : merge xs ys  
merge xs ys = xs ++ ys

Is this still a valid definition?

Back to the Tests:  

Main> runTestTT mergeTests  
Cases: 6 Tried: 6 Errors: 0 Failures: 0  

Main>
Lessons Learned:

- Writing tests (even before we’ve written the code we want to test) can expose key details / design decisions
- A library like HUnit can help to (partially) automate the process
- Development alternates between coding and testing
- Bugs are expensive, running tests is cheap
- Good tests can last a long time; continuing use as code evolves

Lawful Programming:

How can we give useful information about a function without necessarily having to give all the details of its definition?

- Informal description:
  “map applies its first argument to every element in its second argument …”
- Type signature:
  \[ \text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \]
- Laws:
  - Normally in the form of equalities between expressions ...

Algebra of Lists:

- \( (++) \) is associative with unit \([\cdot]\)
  \[ xs ++ (ys ++ zs) = (xs ++ ys) ++ zs \]
  \[ [] ++ xs = xs = xs ++ [] \]
- \( \text{map} \) preserves identities, distributes over composition and concatenation:
  \[ \text{map } \text{id} = \text{id} \]
  \[ \text{map } (f \cdot g) = \text{map } f \cdot \text{map } g \]
  \[ \text{map } f (xs ++ ys) = \text{map } f \cdot xs ++ \text{map } f \cdot ys \]

... continued:

- \( \text{filter} \) distributes over concatenation
  \[ \text{filter } p (xs ++ ys) = \text{filter } p \cdot xs ++ \text{filter } p \cdot ys \]
- Filters and maps:
  \[ \text{filter } p \cdot \text{map } f = \text{map } f \cdot \text{filter } (p \cdot f) \]
- Composing filters:
  \[ \text{filter } p \cdot \text{filter } q = \text{filter } r \]
  where \( r \cdot x = q \cdot x \land p \cdot x \)

Aside: Lambda Notation

- The syntax \( \vars \rightarrow \expr \) denotes a function that takes arguments \( \vars \) and returns the corresponding value of \( \expr \)
- Referred to as a lambda expression after the corresponding construct in \( \lambda \)-calculus
- Examples:
  - \( (\x \rightarrow x + 1) \)
  - \( \text{filter } p \cdot \text{filter } q = \text{filter } (\x \rightarrow q \cdot x \land p \cdot x) \)
  - \( (\x \rightarrow 1 + 2^x) \)
  - \( (\x \cdot y \rightarrow (x + y) \ast (x - y)) \)
Laws Describe Interactions:

- A lot of laws describe how one operator interacts with another
- Example: interactions with reverse:
  - reverse . map f = map f . reverse
  - reverse . filter p = filter p . reverse
  - reverse (xs ++ ys) = reverse ys ++ reverse xs
  - reverse . reverse = reverse
- Caution: stating a law doesn’t make it true! (e.g., the last two laws for reverse ...)

Uses for Laws:

Laws can be used:
- To capture/document deep intuitions about program behavior
- To support reasoning about program behavior
- To optimize or transform programs (either by hand, or in a compiler)
- As properties to be tested
- As properties to be proved

Laws for Merge:

What laws might we formulate for merge?
- If xs and ys are sorted, then merge xs ys is sorted
- merge (sort xs) (sort ys) should be sorted
- merge xs ys == merge ys xs
- merge xs (merge ys zs) == merge (merge xs ys) zs
- merge [] ys == ys and merge xs [] == xs
- merge xs xs == xs
- length (merge xs ys) <= length xs + length ys
- xs is a subset/subsequence of merge xs ys

From Laws to Functions:

mergeProp1 :: [Int] -> [Int] -> Bool
mergeProp1 xs ys = sorted xs ==> sorted ys ==> sorted (merge xs ys)

(==> ) :: Bool -> Bool -> Bool
x ==> y = not x || y

sorted :: [Int] -> Bool
sorted xs = and [ x <= y | (x,y) <- zip (tail xs) ]

Testing mergeProp1:

Main> mergeProp1 [1,4,7] [2,4,6]
True
Main> mergeProp1 [1,4,7] [2,4,1]
True
Main> sorted [1,4,7]
True
Main> sorted [2,4,1]
False
Main> Question: to test merge, I wrote more code ...

If I don’t trust my programming skills, why am I writing even more (untrustworthy) code?

Formulate More Tests!

import List(sort)

sortSorts :: [Int] -> Bool
sortSorts xs = sorted (sort xs)

sortedEmpty :: Bool
sortedEmpty = sorted []

sortIdempotent :: [Int] -> Bool
sortIdempotent xs = sort (sort xs) == sort xs
More Laws to Functions:

mergePreservesOrder :: [Int] -> [Int] -> Bool
mergePreservesOrder xs ys
    = sorted (merge (sort xs) (sort ys))

mergeCommutes :: [Int] -> [Int] -> Bool
mergeCommutes xs ys
    = merge us vs == merge vs us
    where us = sort xs
          vs = sort ys

etc...

Testing mergeProp1:

Main> mergeCommutes [1,4,7] [2,4,6]
True
Main> mergeCommutes [1,4,7] [2,4,1]
True
Main> mergePreservesOrder [1,4,7] [2,4,6]
True
Main> mergePreservesOrder [1,4,7] [2,4,1]
True
Main>

Automated Testing:

- Of course, we can run as many individual test cases as we like:
  - Pick a test case
  - Execute the program
  - Compare actual result with expected result

- Wouldn't it be nice if the environment could help us to go directly from properties to tests?

- Wouldn't it be nice if the environment could run the tests for us automatically too?

QuickCheck:

- This is a job for QuickCheck!

- “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs” by Koen Claessen and John Hughes, Chalmers University, Sweden. (Published at ICFP 2000)

- In Hugs: import Test.QuickCheck

Understand Before you Code:

- Haskell programmers write types first ...
  - ... type checking might find bugs.

- Extreme programmers write tests first ...
  - ... running the tests might find bugs.

- Very few programmers write laws first ...
  - ... because nothing encourages or rewards them for writing laws.

Wanted! Reward!

- In the short-term, programmers won’t see any reward for writing laws ...

- ... so they won’t write them.

- If programmers can derive some benefit from writing laws, then perhaps they will do it ...
Lawful Programming:

reverse :: [a] -> [a]
reverse xs = …

{- reverse satisfies the following:
  reverse (xs ++ ys)
  ==
  reverse ys ++ reverse xs
-}
The Testable Class:

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

instance Testable Bool where ...

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

where ...

Generating Arbitrary Values:

class Arbitrary a where arbitrary :: Gen a

instance Arbitrary ()
instance Arbitrary Bool
instance Arbitrary Int
instance Arbitrary Integer
instance Arbitrary Float
instance Arbitrary Double
instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b)
instance Arbitrary a => Arbitrary [a]

Quantified or Parameterized?

Main> quickCheck prop_revApp
OK, passed 100 tests.

Main> quickCheck (prop_revApp [1,2,3])
OK, passed 100 tests.

Main>

If you don’t give a specific value for an argument, quickCheck will generate arbitrary (i.e. random) values for you.

QuickCheck-ing merge:

Main> quickCheck mergeCommutes
OK, passed 100 tests.

Main> quickCheck mergePreservesOrder
OK, passed 100 tests.

Main>

So far, so good …

Continued ...

mergeProp1 :: [Int] -> [Int] -> Bool
mergeProp1 xs ys = sorted xs ==> sorted ys ==> sorted (merge xs ys)

What happens?

Main> quickCheck mergeProp1
Falsifiable, after 7 tests:

[-1,5,5,4,3,-5] True
[5,-6,2,6,-6,0] Huh?

Main>

What went wrong?

Main> sorted [-1,-5,5,4,3,-5]
False
Main> sorted [5,-6,2,6,-6,0]
False
Main> sorted (merge [-1,-5,5,4,3,-5] [5,-6,2,6,-6,0])
False
Main> False ==> False ==> False
False
Main> False ==> (False ==>) False
True
Main>
A Fix! (in fact, infix)

infixr =>
(=>) :: Bool -> Bool -> Bool
x => y = not x || y

What happens?
Main> quickCheck mergeProp1
OK, passed 100 tests.

Main>
Hooray!!!

Are we Happy Now?

mergeProp1 :: [Int] -> [Int] -> Bool
mergeProp1 xs ys = sorted xs =>>
sorted ys =>>
sorted (merge xs ys)

100 tests passed!

But how many of them were trivial (i.e., one or both arguments unsorted)?

Understanding Test Results:

- Use the collect combinator:
  mergeProp1sorted xs ys
  = collect (sorted xs, sorted ys) (mergeProp1 xs ys)

- Testing:
  Main> quickCheck mergeProp1sorted
  OK, passed 100 tests.
  45% (False,False).
  25% (True,True).
  20% (True,False).
  10% (False,True).

  Main>

Understanding ==>:

- The real (==>) operator is not a standard "implies" function of type Bool -> Bool -> Bool
- When we test a property p ==> q, QuickCheck will try to find 100 test cases for which p is true, and will test q in each of those 100 cases
- If it tries 1000 candidates without finding enough solutions, then it will give up:
  Main> quickCheck (\b -> (b == not b) ==> b)
  Arguments exhausted after 0 tests.

- QuickCheck can be configured to use different numbers of tests/attempts

Understanding Test Results:

- Or use the classify combinator:
  mergeProp1long xs ys
  = classify (length xs > 10) "long"
  $ classify (length xs <= 5) "short"
  $ mergeProp1 xs ys

- Testing:
  Main> quickCheck mergeProp1long
  OK, passed 100 tests.
  49% short.
  29% long.

  Main>

Writing Custom Generators:

Instead of generating random values and selecting only some, we can try to generate the ones we want directly:

sortedList :: Gen [Int]
sortedList = do ns <- arbitrary
                 return (sort ns)
More Examples:

Now we can use QuickCheck’s `forAll` combinator to define:

\[
\text{prop\_mergePreservesOrder} = \text{forAll sortedList } \forall xs -> \\
\text{forAll sortedList } \forall ys -> \\
\text{sorted (merge } xs \text{ ys)}
\]

\[
\text{prop\_mergeCommutes} = \text{forAll sortedList } \forall xs -> \\
\text{forAll sortedList } \forall ys -> \\
\text{merge } xs\ ys == \text{merge } ys\ xs
\]

\[
\text{prop\_mergeIdempotent} = \text{forAll sortedList } \forall xs -> \\
\text{merge } xs\ xs == \text{xs}
\]

Lessons Learned:

- QuickCheck is a useful and lightweight tool that encourages and rewards the lawful programmer!
- There is a script that automatically runs `quickCheck` on all of the properties in a file that have names of the form `prop_XXX`
- Interpreting test results may require some care …
- "Good" (random) test data can be hard to find …