COMP3141

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

Recap: Induction

Suppose we want t s n.
Remember that t

Definition of Na

1. 0 is a natural number.
2. For any natural number n, $n+1$ is also a natural nu

Recap: Induction

Therefore, to show $P(n)$ for all n, it suffices to show:

1. $P(0)$ (the *base case*), and
2. assuming $P(k)$ (the *inductive hypothesis*),
   $\Rightarrow P(k+$

Example

Show that $f(n) = n^2$ for all $n \in \mathbb{N}$, where:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2n - 1 + f(n-1) & \text{if } n > 0 \end{cases}$$

(done on iPad)

Induction on Lists

Haskell lists can be defined similarly to natural numbers.

Definition of Ha

1. [] is a list.
2. For any list

This means, if we want to prove that a property ls, it suffices to show:

1. $P(\texttt{[]})$ (the base case)
2. $P(\texttt{x:xs})$ for all items x, assuming the inductive hypothesis $P(\texttt{xs})$.

Induction on Lists: Example


```
sum :: [Int] -> Int
sum []     = 0            -- 1
sum (x:xs) = x + sum xs   -- 2
```



```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = x `f` foldr f z xs
```


Example

Prove for all `ls`:

```
sum ls == foldr (+) 0 ls
```

(done on iPad)

Custom Data Types

So far, we have seen type synonyms using the type keyword. For a graphics library, we might define:


```
type Point   = (Float
type Vector  = (Flo
type Line    = (Point, Po
type Colour  = (Int, Int, Int, Int) -- RGBA

movePoint :: Point -> Vector -> Point
movePoint (x,y) (dx,dy) = (x + dx, y + dy)
```


But these definitions allow Points and Vectors to be used interchangeably, increasing the likelihood of errors.

Product Types

We can define our own compound types using the `data` keyword:

Type name

Constructor name

Constructor argument types

```haskell
data Point = Poin[illegible]
             deriving (Show, Eq)

data Vector = Vector Float Float
              deriving (Show, Eq)

movePoint :: Point -> Vector -> Point
movePoint (Point x y) (Vector dx dy)
   = Point (x + dx) (y + dy)
```

Records

We could define Colour similarly:

```
data Colour = Colour Int Int Int Int
```

But this has so many

Haskell lets us decl tion style on the previous sli


```
data Colour = Colour { redC     :: Int
                     , greenC   :: Int
                     , blueC    :: Int
                     , opacityC :: Int
                     } deriving (Show, Eq)
```

Here, the code `redC (Colour 255 128 0 255)` gives 255.

Enumeration Types

Similar to enums in C and Java, we can define types to have one of a set of predefined values:

```
data LineStyl
```



```
              deriving (Show, Eq)
```



```
data FillStyle = SolidFill | NoFill
              deriving (Show, Eq)
```

Types with more than one constructor are called *sum types*.

Algebraic Data Types

Just as the Point constructor took two Float arguments, constructors for sum types can take parameters too, allowing us to model different kinds of shape:


```
data PictureO
    = Path    [Point]       Colour L
    | Circle  Poi
    | Polygon [Point]       Colour LineStyle FillStyle
    | Ellipse Point Float Float Float
              Colour LineStyle FillStyle
    deriving (Show, Eq)

type Picture = [PictureObject]
```

Live Coding: Cool Graphics

Example (Ellipses and Curves)

Recursive and Parametric Types

Data types can also be defined with parameters, such as the well known Maybe type, defined in the stan

```
data Maybe a = Just a | N
```


Types can also be re
could define them ourselves:

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


We can even define natural numbers, where 2 is encoded as ro)):

```
data Natural = Zero | Succ Natural
```

Types in Design

Sage Advice

An old adage due to Yaron Minsky (of Jane Street) is:

Make illegal states unrepresentable.

Choose types that [illegible] lure scenarios are elim[illegible]

Example (Contact Details)

```
data Contact = C Name (Maybe Address) (Maybe Email)
```


is changed to:

```
data ContactDetails = EmailOnly Email | PostOnly Address
                    | Both Address Email
data Contact = C Name ContactDetails
```

What failure state is eliminated here? Liam: also talk about other famous screwups

Partial Functions

Failure to follow Yaron's excellent advice leads to partial functions.

Definition

A *partial function* is a function not defined for all possible inputs.

Examples: `hea`

Partial functio

undefined cases a

To eliminate partiality, we must either:

- enlarge the codomain, usually with a Maybe


```
safeHead :: [a] -> Maybe a -- Q: How is th
safeHead (x:xs) = Just x
safeHead []     = Nothing
```

- Or we must constrain the domain to be more specific:

```
safeHead' :: NonEmpty a -> a -- Q: How to define?
```

Type Classes

You have already seen functions such as:

- `compare`
- `(==)`
- `(+)`
- `show`

that work on multiple types, and their corresponding constra `Ord`, `Eq`, `Num` and `Show`.

These constraints are called *type classes*, and can be t which certain operations are implemented.

Show

The `Show` type class is a set of types that can be converted to strings. It is defined like:

```
class Show a where -- nothing to do with OOP
  show :: a -> String
```


Types are added to t

```
instance Show B
  show True  = "Tru
  show False = "False"
```


We can also define instances that depend on other instances:

```
instance Show a => Show (Maybe a) where
  show (Just x) = "Just " ++ show x
  show Nothing  = "Nothing"
```


Fortunately for us, Haskell supports automatically `deriving` instances for some classes, including `Show`.

Read

Type classes can also overload based on the type returned, unlike similar features like Java's interfaces

```
class Read a where
  read :: Strin
```


Some examples:

- `read "34" :: Int`
- `read "22" :: Char` Runtime error!
- `show (read "34") :: String` Type error!

Semigroup

Semigroups

A *semigroup* is a pair of a set S and an operation $\bullet : S \to S \to S$ where the operation $\bullet$ is *associative*.

Associativity is de

Haskell has a type class for semigroups! The associativity law is e [illegible] programmer discipline:

```
class Semigroup s where
  (<>) :: s -> s -> s
  -- Law: (<>) must be associative.
```

What instances can you think of?

Semigroup

Lets implement additive colour mixing:


```
instance Semigroup Colour where
  Colour r1 g1 b1 a
     = Colour


          (mix b1 b2)
          (mix a1 a2)
    where
     mix x1 x2 = min 255 (x1 + x2)
```


Observe that associativity is satisfied.

Monoid

Monoids

A *monoid* is a semigroup $(S, \bullet)$ equipped with a special *identity element* $z : S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y.


```
class (Semigr
  mempty :: a
```


For colours, the ide

```
instance Monoid Colour where
  mempty = Colour 0 0 0 0
```


For each of the semigroups discussed previously:

- Are they monoids?
- If so, what is the identity element?

Are there any semigroups that are not monoids?

Non-empty lists, maximum

Newtypes

There are multiple possible monoid instances for numeric types like `Integer`:

- The operation `(+)` is associative, with identity element `0`
- The operati

Haskell doesn't us
class in the entire pr

A common technique is to define a separate type that is represen
original type, but can have its own, different type class instance

In Haskell, this is done with the `newtype` keyword.

Newtypes

A `newtype` declaration is much like a `data` declaration except that there can be only one constructor and it must take exactly one argument:


```
newtype Score = S Integer

instance Semi
  S x <> S y = S (x + y)

instance Monoid Score where
  mempty = S 0
```


Here, `Score` is represented identically to `Integ` ... alty is incurred to convert between them.

In general, newtypes are a great way to prevent mistakes. Use them frequently!

Ord

Ord is a type class for inequality comparison:


```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should i

For all x, y, and

1. *Reflexivit* x <= x
2. *Transitivity*: If x <= y and y <= z then x <
3. *Antisymmetry*: If x <= y and y <= x then

4. *Totality*: Either x <= y or y <= x

Relations that satisfy these four properties are called *total orders*. Without the fourth (totality), they are called *partial orders*.

Eq

Eq is a type class for equality or equivalence:

```
class Eq a where
  (==) :: a -> a -> Bool
```


What laws should i

For all x, y, and

1. *Reflexivit*
2. *Transitivity*: If `x == y` and `y == z` then `x =`
3. *Symmetry*: If `x == y` then `y == x`

Relations that satisfy these are called *equivalenc*

Some argue that the Eq class should be only for *equality*, requiring stricter laws like:

If `x == y` then `f x == f y` for all functions `f`

But this is debated.

Types and Values

Haskell is actually

- The *value* et, 3 etc
- The *type-* like `String`, and type *constructors* like `Maybe`, `(->)`, `[ ]`

This type level language itself has a type system!

Kinds

Just as terms in the v

language are give

The most basic kin

- Types such a
- Seeing as `Maybe` is parameterised by one argume `* -> *`: given a type (e.g. `Int`) it will return a type (

Lists

Suppose we have a function:

```
toString :: Int -> String
```

And we also have a fu


```
getNumbers :: S
```

How can I compose `toString` with `getNumbers` to get a function `f` of type `Seed -> [String]`?

Answer: we use `map`:

```
f = map toString . getNumbers
```

Maybe

Suppose we have a function:


```
toString :: Int -> String
```

And we also have a fu

```
tryNumber :: Se
```


How can I compose of type Seed -> Maybe String?

We want a map function but for the Maybe type

```
f = maybeMap toString . tryNumber
```

Let's implement it.

Functor

All of these functions are in the interface of a single type class, called `Functor`.


```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Unlike previous t ctor is over types of kind `* -> *`.

Instances for:

- Lists
- `Maybe`
- Tuples (how?)
- Functions (how?)

Demonstrate in live-coding

Functor Laws

The functor type class must obey two laws:

Functor Laws

1. `fmap id == id`
2. `fmap f . fmap g == f`

In Haskell's type system it's impossible to make a total atisfies the first law but violates the second.

This is due to *parametricity*, a property we will return to in Week 8 or 9

Homework

1. Do the first pr
   be due in exac
2. Last week's
3. This week's quiz is also up, due next friday (the friday after th