COMP3141

Software System Design and Implementation

Induction, Data Types and Type Classes

Dr. Liam O'Connor
University of Edinburgh LFCS (and UNSW)
Term 2 2020

Recap: Induction

Suppose we want to prove that a property $P(n)$ holds for all natural numbers n.
Remember that the set of natural numbers $\mathbb{N}$ can be defined as follows:

Definition of Natural Numbers

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

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*),
   $\Rightarrow P(k+1)$ (the *inductive case*).

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 Haskell Lists

1. `[]` is a list.
2. For any list `xs`, `x:xs` is also a list (for any item `x`).

This means, if we want to prove that a property $P(\texttt{ls})$ holds for all lists `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

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

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

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

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

But these definitions allow `Point`s and `Vector`s 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 = Point Float Float
             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 parameters, it's hard to tell which is which.
Haskell lets us declare these types as *records*, which is identical to the declaration style on the previous slide, but also gives us projection functions and record syntax:

```
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 LineStyle = Solid
               | Dashed
               | Dotted
               deriving (Show, Eq)

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

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

Algebraic Data Types

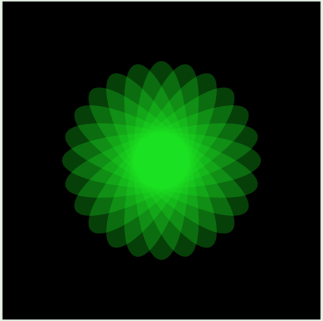
Just as the Point constructor took two Float arguments, constructors for sum types can take parameters too, allowing us to model different kinds of shape:

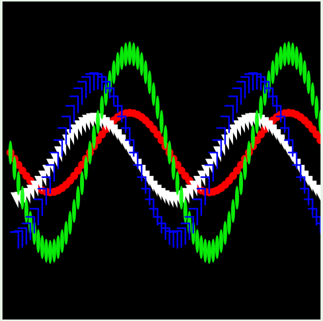
```
data PictureObject
    = Path    [Point]       Colour LineStyle
    | Circle  Point Float   Colour LineStyle FillStyle
    | Polygon [Point]       Colour LineStyle FillStyle
    | Ellipse Point Float Float Float
              Colour LineStyle FillStyle
    deriving (Show, Eq)

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 standard library:

```
data Maybe a = Just a | Nothing
```

Types can also be recursive. If lists weren't already defined in the standard library, we could define them ourselves:

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

We can even define natural numbers, where 2 is encoded as `Succ(Succ Zero)`:

```
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 *constrain* your implementation as much as possible. Then failure scenarios are eliminated automatically.

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: `head`, `tail`, `(!!)`, division

Partial functions are to be avoided, because they cause your program to crash if undefined cases are encountered.
To eliminate partiality, we must either:

- enlarge the codomain, usually with a `Maybe` type:

```
safeHead :: [a] -> Maybe a -- Q: How is this safer?
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 constraints on type variables `Ord`, `Eq`, `Num` and `Show`.

These constraints are called *type classes*, and can be thought of as a set of types for 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 the type class as an *instance* like so:

```
instance Show Bool where
  show True  = "True"
  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 :: String -> a
```

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 defined as, for all a, b, c:

$$(a \bullet (b \bullet c)) = ((a \bullet b) \bullet c)$$

Haskell has a type class for semigroups! The associativity law is enforced only by programmer discipline:

```haskell
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 a1 <> Colour r2 g2 b2 a2
      = Colour (mix r1 r2)
               (mix g1 g2)
               (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 (Semigroup a) => Monoid a where
  mempty :: a
```

For colours, the identity element is transparent black:

```
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 operation `(*)` is associative, with identity element `1`

Haskell doesn't use any of these, because there can be only one instance per type per class in the entire program (including all dependencies and libraries used).

A common technique is to define a separate type that is represented identically to the original type, but can have its own, different type class instances.

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 Semigroup Score where
  S x <> S y = S (x + y)

instance Monoid Score where
  mempty = S 0
```

Here, `Score` is represented identically to `Integer`, and thus no performance penalty 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 instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x <= x`.
2. *Transitivity*: If `x <= y` and `y <= z` then `x <= z`.
3. *Antisymmetry*: If `x <= y` and `y <= x` then `x == y`.
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 instances satisfy?

For all x, y, and z:

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

Relations that satisfy these are called *equivalence relations*.

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 comprised of two languages.

- The *value-level* language, consisting of expressions such as `if`, `let`, `3` etc.
- The *type-level* language, consisting of types `Int`, `Bool`, synonyms like `String`, and type *constructors* like `Maybe`, `(->)`, `[ ]` etc.

This type level language itself has a type system!

Kinds

Just as terms in the value level language are given types, terms in the type level language are given *kinds*.

The most basic kind is written as `*`.

- Types such as `Int` and `Bool` have kind `*`.
- Seeing as `Maybe` is parameterised by one argument, `Maybe` has kind `* -> *`: given a type (e.g. `Int`), it will return a type (`Maybe Int`).

Lists

Suppose we have a function:

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

And we also have a function to give us some numbers:

```
getNumbers :: Seed -> [Int]
```

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 function that may give us a number:

```
tryNumber :: Seed -> Maybe Int
```

How can I compose `toString` with `tryNumber` to get a function `f` 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 type classes we've seen like `Ord` and `Semigroup`, `Functor` 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 == fmap (f . g)`

In Haskell's type system it's impossible to make a total `fmap` function that satisfies 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 programming exercise, and ask us on Piazza if you get stuck. It will be due in exactly 1 week from the start of this lecture.
2. Last week's quiz is due this friday. Make sure you submit your answers.
3. This week's quiz is also up, due next friday (the friday after this one).