COMP3141

Software System Design and Implementation

Induction, Data Types and Type Classes Practice

Christine Rizkallah
CSE, UNSW (and Data61)
Term 2 2019

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

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.

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*).

Recap: 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`).

Recap: 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 $P(\texttt{xs})$.

Recap: Type Classes

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*.

Recap: Type Classes

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:

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

Recap: Type Classes

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.

Recap: Type Classes

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.

Example

```
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

List Monoid Example

```haskell
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys              -- 1
(++) (x:xs) ys = x : xs ++ ys    -- 2
```

List Monoid Example

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys             -- 1
(++) (x:xs) ys = x : xs ++ ys   -- 2
```

Example (Monoid)

Prove for all `xs, ys, zs`:

```
((xs ++ ys) ++ zs) = (xs ++ (ys ++ zs))
```

List Monoid Example

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys              -- 1
(++) (x:xs) ys = x : xs ++ ys   -- 2
```

Example (Monoid)

Prove for all `xs`, `ys`, `zs`:

```
((xs ++ ys) ++ zs) = (xs ++ (ys ++ zs))
```

Additionally Prove

1. for all `xs`:

   ```
   [] ++ xs == xs
   ```

2. for all `xs`:

   ```
   xs ++ [] == xs
   ```

(done on iPad)

List Reverse Example

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys                 -- 1
(++) (x:xs) ys = x : xs ++ ys       -- 2

reverse :: [a] -> [a]
reverse []     = []                 -- A
reverse (x:xs) = reverse xs ++ [x]  -- B
```

Example

Prove for all `ls`:

`reverse (reverse ls) == ls`

(done on iPad)

List Reverse Example

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys                  -- 1
(++) (x:xs) ys = x : xs ++ ys        -- 2

reverse :: [a] -> [a]
reverse []     = []                  -- A
reverse (x:xs) = reverse xs ++ [x]   -- B
```

Example

Prove for all `ls`:

`reverse (reverse ls) == ls`

(done on iPad) stuck!

List Reverse Example

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys                  -- 1
(++) (x:xs) ys = x : xs ++ ys        -- 2

reverse :: [a] -> [a]
reverse []     = []                  -- A
reverse (x:xs) = reverse xs ++ [x]   -- B
```

Example

To Prove for all `ls`:

`reverse (reverse ls) == ls`

First Prove for all `ys`:

`reverse (ys ++ [x]) = x:reverse ys`

(done on iPad)

Recap: Product Type Examples

```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)
```

Recap: Record Example

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

Recap: Algebraic Data Types Example

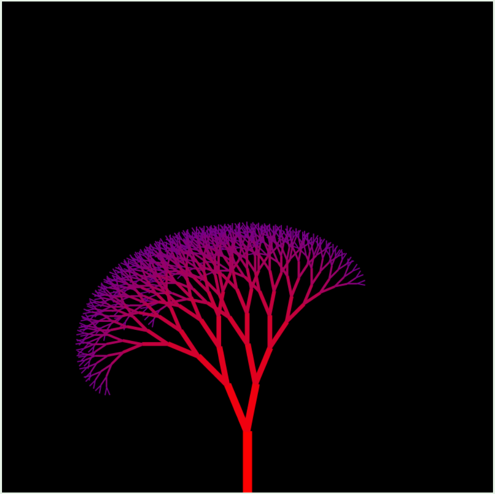
Just as the `Point` constructor took two `Float` arguments, constructors for sum types can take parameters too, allowing us to model different kinds of shape:

```haskell
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: More Cool Graphics

Example (Fractal Trees)



Homework

1. Do the first programming exercise, and ask us on Piazza if you get stuck. It is due in 6 days.
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 (9 days away).