COMP3141

Software System Design and Implementation

Induction, Data Types and Type Classes Practice

Curtis Millar
CSE, UNSW
11 June 2021

Product 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

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

Sum Types

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

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

Constructors

Constructors are how an value of a particular type is created.

```
data Bool = True | False
data Int = .. | -1 | 0 | 1 | 2 | 3 | ..
data Char = 'a' | 'b' | 'c' | 'd' | 'e' | ..
```

Custom Constructors

```
data PointType = PointConstructor Float Float
           deriving (Show, Eq)

data VectorType = VectorConstructor Float Float
            deriving (Show, Eq)
```

Here, `PointConstructor` and `VectorConstructor` are both constructors.
`PointConstructor` has the type `Float -> Float -> PointType` and
`VectorConstructor` has the type `Float -> Float -> VectorType`

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]
```

Here, type creates a *type alias* which provides only an alternate name that refers to an existing type.

Patterns in Function Definitions

1. Patterns are used to deconstruct an value of a particular type.
2. A pattern can be a binding to a hole (_), a name, or a constructor of the type.
3. When defining a function, each argument is bound using a separate pattern.

Patterns in Function Definitions

```
if' :: Bool -> a -> a -> a
if' True  then' _      = then'
if' False _      else' = else'
```

Patterns in Function Definitions

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Patterns in Function Definitions

```
isVowel :: Char -> Bool
isVowel 'a' = True
isVowel 'e' = True
isVowel 'i' = True
isVowel 'o' = True
isVowel 'u' = True
isVowel _   = False
```

Records and Accessors

```haskell
data Colour = Colour { redC      :: Int, greenC   :: Int
                     , blueC     :: Int, opacityC :: Int
                     }

-- Is equivalent to

data Color = Color Int Int Int Int

redC     (Color r _ _ _) = r
greenC   (Color _ g _ _) = g
blueC    (Color _ _ b _) = b
opacityC (Color _ _ _ o) = o
```

Patterns in Expressions

```
factorial :: Int -> Int
factorial x =
  case x of
    0 -> 1
    n -> n * factorial (n - 1)
```

Newtype

`newtype` allows you to encapsulate an existing type to add constraints or properties without adding runtime overhead.

```
newtype Kilometers = Kilometers Float
newtype Miles = Miles Float

kilometersToMiles :: Kilometers -> Miles
kilometersToMiles (Kilometers kms) = Miles $ kms / 1.60934

milesToKilometers :: Miles -> Kilometers
milesToKilometers (Miles miles) = Kilometers $ miles * 1.60934
```

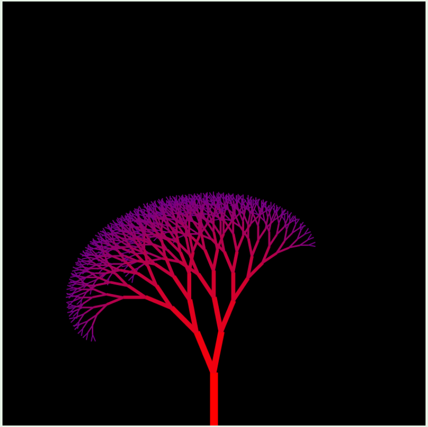
Natural Numbers

```haskell
data Nat = Zero
         | Succ Nat

add :: Nat -> Nat -> Nat
add Zero     n = n
add (Succ a) b = add a (Succ b)

zero = Zero
one = Succ Zero
two = add one one
```

1. `Nat` is recursive as it has the (Succ) constructor which takes a `Nat`.
2. `Nat` has the `Zero` constructor which does not recurse and acts like a *base case*.

More Cool Graphics

Example (Live Coding of Fractal Trees)



Type Classes

1. A type class has nothing to do with OOP classes or inheritance.
2. Type classes describe a set of behaviours that can be implemented for any type.
3. A function or type class instance can operate on a type variable constrained by a type class instead of a concrete type.
4. A type class is similar to an OOP interface.
5. When creating an instance of a type class with *laws*, you must ensure the laws are held manually (they cannot be checked by the compiler).
6. When using a type class with *laws* you can assume that all laws hold for all instances of the type class.

Show

`Show` simply allows us to take a type and represent it as a string.

Haskell Definition

```
class Show a where
        show :: a -> [Char]
```

This is implemented for all of the built-in types such as `Int`, `Bool`, and `Char`

Read

Effectively the 'dual' of `Show`, `Read` allows us to take a string representation of a value and decode it.

You can *think* of read as having the following definition, but it is actually somewhat more complex.

Definition

```
class Read a where
       read :: [Char] -> a
```

This is implemented for all of the built-in types such as `Int`, `Bool`, and `Char`

Ord

`Ord` allows us to compare two values of a type for a *partial* or *total inequality*.

Haskell Definition

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

1. **Tranisitivity**: $x \leq y \wedge y \leq z \rightarrow x \leq z$
2. **Reflexivity**: $x \leq x$
3. **Antisymmetry**: $x \leq y \wedge y \leq x \rightarrow x = y$
4. **Totality** (total order): $x \leq y \vee y \leq x$

Eq

`Eq` allows us to compare two values of a type for an *equivalence* or *equality*.

Haskell Definition

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

1. **Reflexivity**: $x = x$
2. **Symmetry**: $x = y \to y = x$
3. **Tranisitivity**: $x = y \wedge y = z \to x = z$
4. **Negation** (equality): $x \neq y \to \neg(x = y)$
5. **Substitutivity** (equality): $x = y \to f\ x = f\ y$

Derived Instances

When defining a new type we can have the compiler generate instances of `Show`, `Read`, `Ord`, or `Eq` with the `deriving` statement at the end of the definition.

Haskell Example

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

Derived instances of `Ord` will be total orders and will order by fields in the order they appear in a product type and will order constructors in the same order they are defined. Derived instances of `Eq` will be strict equalities.

Kinds of Types

1. Just as values and functions in the *runtime language* of Haskell have *types*, types in the *type language* of Haskell have *kinds*.
2. The kind of a *concrete type* is `*`.
3. Just as *functions* exist over values (with the type `a -> b`), *type constructors* exist for types.
4. `* -> *` is a type constructor that takes a concrete type and produces a concrete type.

Maybe

Haskell Definition

```haskell
-- Maybe :: * -> *
data Maybe a = Just a
             | Nothing
-- Maybe Int :: *
```

1. `Maybe` is a type constructor that takes a type and produces a type that may or may not hold a value.
2. `Maybe Int` is a concrete type that may or may not hold an `Int`.

List

Haskell Definition

```haskell
-- List :: * -> *
data List a = Cons a (List a)
            | Nil

-- List Int :: *
```

1. `List a` is recursive as it has the (Cons) constructor which takes a `List a`.
2. `List a` has the `Nil` constructor which does not recurse and acts like a *base case*.
3. `List` is a type constructor that takes a type and produces a type that holds zero or more of a value.
4. `List Int` is a concrete type that zero or more values of type `Int`.

Haskell List

Definition

```
-- [ ] :: * -> *
data [a] = a : (List a)
         | []
```

1. `[a, b, c]` is syntactic sugar for the constructor `(a : (b : (c : [])))`.
2. `"abc"` is syntactic sugar for the constructor `('a' : ('b' : ('c' : [])))`.
3. Both can also be used as patterns.

Tree

Haskell Definition

```haskell
-- Tree :: * -> *
data Tree a = Node a (Tree a) (Tree a)
            | Leaf
```

1. `Tree a` is recursive in the same manner as `List a`.
2. `Tree` is a type constructor that takes a type and produces a type that holds zero or more of a value in a tree.
3. `Tree Int` is a concrete type that holds zero or more values of type `Int` in a tree.

Semigroup

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

Haskell Definition

```haskell
class Semigroup a where
    (<>) :: a -> a -> a
```

1. **Associativity**: $(a \bullet (b \bullet c)) = ((a \bullet b) \bullet c)$

Example

```haskell
instance Semigroup [a] where
    (<>) = (++)
```

Monoid

A *monoid* is a semigroup $(S, \bullet)$ equipped with a special *identity element*.

Haskell Definition

```
class (Semigroup a) => Monoid a where
    mempty :: a
```

1. **Identity**: $(mempty \bullet x) = x = (x \bullet mempty)$

Example

```
instance Monoid [a] where
  mempty = []
```

Inductive Proofs

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*).

Natural Numbers Example

```haskell
data Nat = Zero
         | Succ Nat

add :: Nat -> Nat -> Nat
add Zero     n = n
add (Succ a) b = add a (Succ b)

one = Succ Zero
two = Succ (Succ Zero)
```

Example (1 + 1 = 2)

Prove `one `add` one = two` (done in editor)

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 $P(\texttt{xs})$.

List Monoid Example

```haskell
(++) :: [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 in editor)

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`
(done in editor)
First Prove for all `ys`: `reverse (ys ++ [x]) = x:reverse ys`
(done in editor)

Graphics and Artwork

```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]
```

Homework

1. Last week's quiz is due before on Friday. Make sure you submit your answers.
2. Do the first programming exercise, and ask us on Discourse if you get stuck. It is due by the start if my next lecture (in 7 days).
3. This week's quiz is also up, it's due next Friday (in 7 days).