Curtis Millar
CSE, UNSW (and Data61)
10 June 2020

Product Types


```haskell
data Point = Point Float Float
```



```haskell
data Vector = Vec
```



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

Records


```
data Colour = Col
```



```
              , opacityC :: Int
              } deriving (Show, Eq)
```

Sum Types


```haskell
data LineStyle = Solid
```



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

Constructors

Constructors ar


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

Custom Constructors


```
data Point = Poin
```



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

Here, `Point` and `Vector` are both constructors.

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 L
    | Circle  Poi
    | Polygon [
    | 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 ar
2. A pattern ca _ pe.
3. When defining a function, each argument is bound usin

Patterns in Function Definitions


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

Patterns in Function Definitions


```
factorial :: In
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Patterns in Function Definitions


```
isVowel :: Char -
isVowel 'a' = Tru
isVowel 'e' = Tru
isVowel 'i' = Tru
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


```haskell
factorial :: In
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.

```haskell
newtype Kilom
newtype Miles = M
```



```haskell
kilometersToMiles :: Kilometers -> Miles
kilometersToMiles (Kilometers kms) = Miles $ kms / 1.6
```



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

Natural Numbers


```
data Nat = Zero
         | Succ Nat

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

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 o
   type class ins
4. A type class is similar to an OOP interface.
5. When creating an instance of a type class with ws are
   held manually (they cannot be checked by the compiler)
6. When using a type class with *laws* you can ass
   instances of the type class.

Show

Show simply allo

Haskell Definit


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

This is implemented for all of the built-in types such as har

Read

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

You can *think*
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 Definit

```
class Ord a where
      (<=
```


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
        (==
```


1. **Reflexivity**: $x = x$
2. **Symmetry**: $x = y \to y = x$
3. **Tranisitivity**: $x = y \land y = z \to x = z$
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 Examp

```
data Colour = Col

                 , 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 *typ*
2. The kind of a

3. Just as *fu* *onstructors* exist for types.
4. `* -> *` is a type constructor that takes a concrete type a type.

Maybe

Haskell Definition

```haskell
-- Maybe :: * -> *
data Maybe a = Just a
```


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

List

Haskell Definition

```
-- List :: * -> *
data List a = Cons a (L
```


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 r *base case*.
3. `List` is a type constructor that takes a type and produce 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 .
2. `"abc"` is syntactic sugar for the constructor .
3. Both can also be used as patterns.

Tree

Haskell Definition

```
-- Tree :: * -> *
data Tree a = Node a (T
```


1. `Tree a` is recursive in the same manner as
2. `Tree` is a type constructor that takes a type and produce 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 Definit

```
class Semigr
    (<>) :: a -> a -> a
```


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

Example

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

Monoid

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

Haskell Definition

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


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

Example

```haskell
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 Na

1. 0 is a natural nu
2. For any natu

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 -> Na
add Zero     n = n
add (Succ a) b = add a (S

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 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 $P(\texttt{xs})$.

List Monoid Example


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


Example (Mo

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


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



```haskell
reverse :: [a] -> [a]
reverse [] = []
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


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



```
type Picture = [PictureObject]
```

Homework

1. Last week's
2. Do the first p
   by the start if
3. This week's quiz is also up, it's due next Friday (in 9 days).