Contents

Records in Haskell

A record is really just a standard data type with one constructor:

```
data Pair a b = Pair a b
```

To use a type like this you might provide some "accessor" functions:

```
first :: (Pair a b) -> a
first (Pair a _) = a

second :: (Pair a b) -> b
second (Pair _ b) = b
```

Defining the data type as a record type is just a syntactic convenience for creating those accessors, or "fields":

```
newtype Pair a b = Pair {
                         first :: a,
                         second :: b
                        }
```

Note that two different record declarations must have disjoint access names.

We can either create values using the urual constructor, or using the named fields:

```haskell
f = Pair 1 'a'

g = Pair { second = 'b', first = 2 }
```

(notice the order doesn't matter when we use the fields)

We can look them up using the field names:

```haskell
h :: (Pair int b) -> Int
h p = (first p) + 1
```

There's also a convenient syntax for creating a new value based on an old one:

```haskell
h = g { second = 'B' }

> h
Pair 2 'B'
```

Consider the following plan to treat `Int`s differently, by having a show instance that produces roman numerals, plus some code to do additions.

```haskell
newtype Roman = Roman Int
```

We will want a way to get at the underlying `Int` inside a `Roman`:

```haskell
unRoman (Roman i) = i
```

We can then define addition:

```haskell
rAdd r1 r2 = Roman (unRoman r1 + unRoman r2)
```

We can now define our `Show` instance:

```haskell
romanShow i
  | i < 4 = replicate i 'I'
  | i == 4 = "IV"
  | i < 9 = 'V' : replicate (i-5) 'I'
  | i == 9 = "IX"
  | i < 14 = 'X' : replicate (i-10) 'I'
  | otherwise = "my head hurts"
```

newtypes using Records

From the Roman example we had:

```
newtype Roman = Roman int
unRoman (Roman i) = i
```

Using the record idea, we can do this all in the `newtype` declaration (lets use `R2/unR2` to avoid clashing with the `Roman/unRoman`)

```
newtype R2 = r2 { unR2 :: Int }
```

So we can code show and addition as

```
instance Show R2 where show = romanShow . unR2
r2Add r1 r2 = R2 (unR2 r1 + unR2 r2)
```

Standard (Prelude) Classes in Haskell

A wide range of classes and instances are provided as standard in Haskell

- Relations
  - `Eq`, `Ord`
- Enumeration
  - `Enum`, `Bounded`
- Numeric
  - `Num`, `Real`, `Integral`, `Factional`, `Floating`, `RealFrac`, `RealFloat`
- Textual
  - `Show`, `Read`
- Higher-Order
  - `Functor`, `Applicative`, `Monad`, `Monoid`, `Foldable`, `Traversable`

Eq a Class

- Class Members

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

- Intances

```
(), Bool, Char, Int, Integer, Float, Double, Ordering, IOError, Maybe, Either, [a], (a, 
```

- Comments
  - There is no instance of `Eq` for any function type
  - Function equality is undecidable
  - `f = g` iff `f x = g x` for all `x` in input type of `f` and `g`

Ord a Class

- Class Members

```
compare :: a -> a -> Ordering
(<), (<=), (>=), (>) :: a -> a -> Bool
max, min :: a -> a -> a
```

- Instances

```
(), Bool, Char, Int, Integer, Float, Double, Ordering, Maybe, Either, [a], (a, b), (a, b
```

- Comments
  - Requires `Eq`
  - Almost everything with the equality also has ordering defined
  - (except `IOError`)

The Ordering Type

- It is straightforward `data` definition

```
data Ordering = LT | EQ | GT
```

- It represents the three possible outcomes of an order comparison

Enum a Class

- Class Members

```
succ, pred     :: a -> a
toEnum         :: Int -> a
fromEnum       :: a -> Int
enumFrom       :: a -> [a]
enumFromThen   :: a -> a -> [a]
enumFromTo     :: a -> a -> [a]
enumFromThenTo :: a -> a -> a -> [a]
```

- Instances

```
Bool, Char, Int, Integer, Float, Double
```

- Comments
  - Basically types for which notation `[stard .. end]` makes sense
  - The `Float` and `Double` instances for `Enum` should be used with care (rounding errors)

Bounded a Class

- Class Members

```
minBound :: a
maxBound :: a
```

- Instances

```
(), Bool, Char, Int, Ordering, (a, b), (a, b, c)
```

- Comments
  - Types that have a (natural) minimum and maximum value

Num a Class

- Class Members

```
(+), (-), (*) :: a -> a -> a
negate        :: a -> a
abs, signum   :: a -> a
fromInteger   :: Integer -> a
```

- Instances

```
Int, Integer, Float, Double
```

- Comments
  - Required: `Eq, Show`
  - Most general notation of number available
  - (Note lack of any form of division)

The Rational Type

- The Rational type, defined in library module `Data.Ratio`

```
type Rational = Ratio Integer
```

- The Ratio type constructor forms a pair

```
data Ratio a = a :% a
```

- We can build Ratio values using infix data constructor `:%`

Real a Class

- Class Members

```
toRational :: a -> Rational
```

- Instances

```
Int, Integer, Float, Double
```

- Comments
  - Required `Num, Ord`
  - A strange class, basically those numbers that can be expressed as rationals (ratio of type numbers)

Integral a Class

- Class Members

```
quot, rem       :: a -> a -> a
div, mod        :: a -> a -> a
quotRem, divMod :: a -> a -> (a,a)
toInteger       :: a -> Integer
```

- Instances

```
Int, Integer
```

- Comments
  - Requires `Real, Enum`
  - Number types that suuport integer-division in various forms

Fractional a Class

- Class Members

```
(/)          :: a -> a -> a
recip        :: a -> a
fromRational :: Rational -> a
```

- Instances

```
Float, Double
```

- Comments
  - Requires `Num`
  - Number types that handle real-number division

Floating a Class

- Class Members

```
pi                  :: a
exp, log, sqrt      :: a -> a
(**), logBase       :: a -> a -> a
sin, cos, tan       :: a -> a
asin, acos, atan    :: a -> a
sinh, cosh, tanh    :: a -> a
asinh, acosh, atanh :: a -> a
```

- Instances

```
Float, Double
```

- Comments
  - Requires `Fractional`
  - All the well-known floating-point functions

RealFrac a

- Class Members

```
properFraction  :: (Integral b) => a -> (b, a)
truncate, round :: (Integral b) => a -> b
ceiling, floor  :: (Integral b) => a -> a
```

- Instances

```
Float, Double
```

- Comments
  - Requires `Real`, `Fractional`
  - Numbers supporting conversions from real to integral forms

RealFloat a Class

- Class Members

```
floatRadix        :: a -> Integer
floatDigits       :: a -> Int
floatRange        :: a -> (Int, Int)
decodeFloat       :: a -> (Integer, Int)
encodeFloat       :: Integer -> Int -> a
exponent          :: a -> Int
significand       :: a -> a
scaleFloat        :: Int -> a -> a
isNaN, isInfinite, isDenormalised,
isNeative, isIEEE :: a -> Bool
atan2             :: a -> a -> a
```

- Instances

```
Float, Double
```

- Comments
  - Requires `RealFrac, Floating`
  - Numbers supporting a floating-point representation plus from IEEE 754 support

Show a Class

- Class Members

```
showsPrec :: Int -> a -> ShowS
show      :: a -> String
showList  :: [a] -> ShowS
```

- Instances

```
(), Bool, Char, Int, Integer, Float, Double, IOError Maybe, Either, Ordering [a], (a, b)
```

- Comments
  - Ways to produce a textual display of a type
  - `showPrec` tables an intiial precedence argument for pretty-printing
  - There is no instance of `Show` for functions

The ShowS type

- Define as

```
type ShowS = String -> String
```

- Using string building functions is often more efficient than directly generating strings

Read a Class

- Class Members

```
readsPrec     :: Int -> ReadS a
readList      :: ReadS [a]
readPrec      :: ReadPrec a
readListPrec :: ReadPrec [a]
```

- Instances

```
(), Bool, Char, Int, Integer, Float, Double, IOError, Maybe, Either, Ordering [a], (a, b
```

- Comments
  - These are parsers that parse the output of show

The ReadS and ReadPrec types

- Define as:

```
type ReadS a = String -> [(a, String)]
newtype ReadPrec = P (Prec -> Read P)
```

- The first function takes a string and return a list of successful parses
- The second is defined in a parser library, that is not part of the Prelude. (We won't discuss this further)

Functor f Class

- Class Members

```
fmap :: (a -> b) -> f a -> f b
```

- Instances

```
Maybe, IO, [], (Either e), ((->) r)
```

- Comments
  - Basically any datatype where mapping a *single-argument* function makes sense
  - This is in a type-*constructor* class

Applicative f Class

- Class Members

```
pure  :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
(*>)  :: f a -> f b -> f b
(<*)  :: f a -> f b -> f a
```

- Instances

```
Maybe, IO, [], (Either e), ((->) r)
```

- Comments
  - Requires `Functor`
  - Basically any datatype where mapping a *multiple-argument* function makes sense
  - This is a type-*constructor* class

Monad m Class

- Class Members

```
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
return :: a -> m a
fail   :: String -> m a
```

- Instances

```
Maybe, IO, [], (Either e), ((->) r)
```

- Comments
  - Requires `Applicative`
  - Types where sequencing of "actions" makes sense
  - Like `Functor`, this is also a type-*constructor* class

Monoid a Class

- Class Members

```
mempty  :: a
mappend :: a -> a -> a
mconcat :: [a] -> a
```

- Instances

```
[], Ordering
```

- Comments
  - Basically any datatype with an associate binary operation that has a unit element
  - This is a type-class

Foldable t Class

- Class Members

```
fold            :: Monoid m => t m -> m
foldMap         :: Monoid m => (a -> m) -> t a -> m
foldr, foldr'   :: (a -> b -> b) -> b -> t a -> b
foldl, foldl'   :: (b -> a -> b) -> b -> t a -> b
foldr1, foldl1 :: (a -> a -> a) -> t a -> a
toList          :: t a -> [a]
null            :: t a -> Bool
length          :: t a -> Int
elem            :: Eq a => a -> t a -> Bool
maximum         :: Ord a => t a -> a
minimum         :: Ord a => t a -> a
sum             :: Num a => t a -> a
product         :: Num a => t a -> a
```

- Instances

```
[], Maybe, (Either a)
```

- Comments
  - Basically any datatype where the concept of folding makes sense
  - This is a type-constructor class
  - It doesn't require `Monoid`, but some of its functions expect it!
  - So we can use `foldl` on a type withoit a `Monoid` instance, because it takes the binary operator and unit element as explicit arguments
  - The use of `fold` requires a type `Monoid` instance so it can its `mempty` and `mappend` values

Traversable t Class

- Class Members

```
traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
sequenceA :: Applicative f => t (f a) -> f (t a)
mapM      :: Monad m => (a -> m b) -> t a -> m (t b)
sequence  :: Monad m => t (m a) -> m (t a)
```

- Instances

```
[], Maybe, (Either a)
```

- Comments
  - Requires `Functor, Foldable`
  - This allows traversing a structure with functions that might fail, and lest failure be handled gracefully
  - This is a type-constructor class

A Key Principle

- Haskell execution replaces sub-expressions, by ones defined to be equal (but hopefully simpler)
- This is an example of a general principle that is very desirable in functional languages - *Referential Transparency*
- A language is *Referentually Transparent* if
  - replacing an expression by another equal expression does not change the meaning/value of the program as a whole
  - e.g. Given program `2 * sum (3:2:1:[]) + x`, then the following are all equivalent programs:

```
2 * (3 + sum (2:1:[])) + x
2 * (3 + 2 + 1 + 0) + x
2 * 6 + x
12 + x
```

Refential Transparency

- Referentially Transaprent
  - A function whose output depends only on its inputs
  - Expressions built from standard arithmetic operators
  - None of the above have and "side-effects"
- Referentially Opaque
  - A function whose value depends on some gloval variable elsewhere
  - A procedure/function that modifies global state
  - The assignment statement
  - A function that performs I/O, it depends on the gloval state of "real world" and modifies it
  - Most of the above are examples of "side-effects"

Why Referential Transparency matters

- Reasoning about program behaviour is easier "substituting equals for equals"
- Code optimization is much simpler
- Scope for code optimization is much greater
- A programming language where every construct is refentially transparent, w.r.t to the "obvious" semantics, is calling "pure"
  - Haskell (and Clean) are pure functional languages
  - ML, Scheme, LISP are generally considered impure functional languages (they have explicit assignment and I/O side-effects), but this is w.r.t. a simple functional semantics for such languages

What Referential Transparency isn't

- Referential Transparency does *not* mean:
  - The language is functional
  - The language has no side-effects
- Referential Transparency is a property relating a language and its semantics
  - Most languages can be given a semantics that makes them referentially transparent
  - This issue is one of degree: such a semantics may be very complex
  - *Pure* functional languages are referentially transparent w.r.t *a relatively simple and obvious semantics*
  - An imperative language with a *full* semantics is also referentially transparent

I/O in Haskell

- We are now going to explore how Haskell supports I/O with all its side effects, whilst also maintaining referential transparency w.r.t the "natural" functional semantics
- First we shall look at a few file operations to note their (destructive) side-effects
- Then we shall introduce the Haskell *IO* type constructor

File I/O Operations

Open/Close

- `openFile`
  - Input: pathname, opening mode(read/write)
  - Effect: modifies filesystem by creating new file
  - Return value: handle to new file
- `hClose`
  - Input: hile handle
  - Effect: closes file indicated by handle, modifying filesystem
  - Return value: none
- Real-world items affected: filesystem, file status
- Opening Modes: `ReadMode, WriteMode, ...`

Put/Get

- `hPutChar`
  - Input: file handle, character
  - Effect: modifies file by appending the character
  - Return value: none
- `hGetChar`
  - Input: file handle
  - Effect: reads character from file current-position, which is then incremented
  - Return value: character read
- Read world items affected: contents of and positions in open files

I/O in Haskell

- Functions that do I/O (as a side-effect) use a special abstract datatype: `IO a`
- Type `IO a` denotes a "value":
  - whose evaluation produces an I/O side-effect
  - which returns a value of a type `a` when evaluated
  - such `values` are called "I/O-actions"
- I/O-actions that don't return a value hav type `IO ()`

- Type `()` is the singleton (aka "unit") type
- It has only one value, also written `()`

- I/O-actions are usually invoked using special syntax ("`do`-notation")

Other File I/O Types

- File opening mode

```
data IOMode = ReadMode | WriteMode | ...
```

- File Pathname - just a string

```
type FilePath = String
```

- File Handles - *pointers* to open files

```
data Handle = ...
```

- This are no types to represent file themselves, or the file-system
- I/O in Haskell works by hiding references to external data that is destructively updated