

Modelling & Datatypes

John Hughes

Software

Software = Programs + Data

Modelling Data

- A big part of designing software is *modelling the data* in an appropriate way
- Numbers are not good for this!
- We model the data by defining *new* types

Modelling a Card Game

- Every card has a *suit*



H hearts,
Whist,
Plump,
Bridge, ...

- Model by a *new* type:

```
data Suit = Spades | Hearts | Diamonds | Clubs
```

The new
type

The values
of this type

Investigating the new type

```
Main> :i Suit
-- type constructor
data Suit

-- constructors:
Spades :: Suit
Hearts :: Suit
Diamonds :: Suit
Clubs :: Suit

Main> :i Spades
Spades :: Suit -- data constructor
```

The new type

The new values
-- *constructors*

Types and
constructors
start with a
capital letter

Printing Values

```
Main> Spades  
ERROR - Cannot find "show" function for:  
*** Expression : Spades  
*** Of type    : Suit
```

Needed to print
values

```
Main> :i show  
show :: Show a => a -> String -- class member
```

- **Fix**

```
data Suit = Spades | Hearts | Diamonds | Clubs  
         deriving Show
```

```
Main> Spades  
Spades
```

The Colours of Cards

- Each suit has a colour – *red* or *black*
- Model colours by a type

```
data Colour = Black | Red
deriving Show
```

- Define functions by *pattern matching*

```
colour :: Suit -> Colour
colour Spades = Black
colour Hearts = Red
colour Diamonds = Red
colour Clubs = Black
```

```
Main> colour Hearts
Red
```

One equation per value

The Ranks of Cards

- Cards have ranks: 2..10, J, Q, K, A

Numeric ranks

- Model by a new type

























```
data Rank = Numeric Integer | Jack | Queen | King | Ace
deriving Show
```

Numeric ranks *contain*
an Integer

```
Main> :i Numeric
Numeric :: Integer -> Rank -- data constructor
Main> Numeric 3
Numeric 3
```


Rank Beats Rank

- When does one rank beat another?

























A					
K					
Q					
J					
m	m>n				
	n	J	Q	K	A

Rank Beats Rank

```
rankBeats :: Rank -> Rank -> Bool
```

























Rank Beats Rank

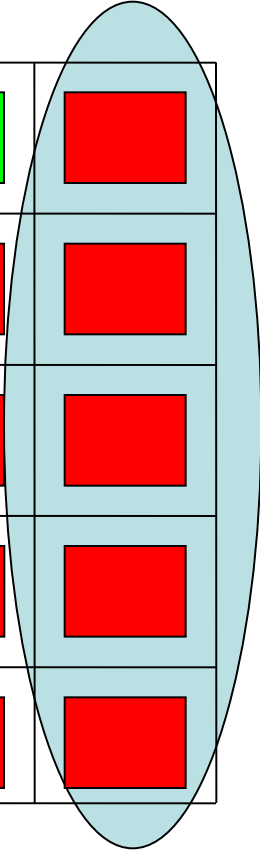
- When does one rank beat another?

A					
K					
Q					
J					
m	m>n				
	n	J	Q	K	A

Rank Beats Rank

- When does one rank beat another?

A					
K					
Q					
J					
m	m>n				
	n	J	Q	K	A



Rank Beats Rank

```
rankBeats :: Rank -> Rank -> Bool
```




















```
rankBeats _ Ace = False
```

Nothing beats an Ace

Matches
anything at all

Rank Beats Rank

- When does one rank beat another?

A					
K					
Q					
J					
m	m>n				
	n	J	Q	K	A

Rank Beats Rank

```
rankBeats :: Rank -> Rank -> Bool
```

```
rankBeats _ Ace = False
```
















```
rankBeats Ace _ = True
```

An Ace beats anything else

Used only if the first equation does not match.

Rank Beats Rank

- When does one rank beat another?





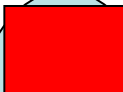



A					
K					
Q					
J					
m	m>n				
	n	J	Q	K	A

Rank Beats Rank

```
rankBeats :: Rank -> Rank -> Bool
rankBeats _ Ace = False
rankBeats Ace _ = True
rankBeats _ King = False
rankBeats King _ = True
```

Rank Beats Rank

- When does one rank beat another?

A					
K					
Q					
J					
m	m > n				
	n	J	Q	K	A

Rank Beats Rank

```
rankBeats :: Rank -> Rank -> Bool
rankBeats _ Ace = False
rankBeats Ace _ = True
rankBeats _ King = False
rankBeats King _ = True
rankBeats _ Queen = False
rankBeats Queen _ = True
rankBeats _ Jack = False
rankBeats Jack _ = True
```

Rank Beats Rank

- When does one rank beat another?

A					
K					
Q					
J					
m	m > n				
	n	J	Q	K	A

Rank Beats Rank

```
rankBeats :: Rank -> Rank -> Bool
rankBeats _ Ace = False
rankBeats Ace _ = True
rankBeats _ King = False
rankBeats King _ = True
rankBeats _ Queen = False
rankBeats Queen _ = True
rankBeats _ Jack = False
rankBeats Jack _ = True
rankBeats (Numeric m) (Numeric n) = m > n
```

Match Numeric 7,
for example

Names the number
in the rank

Examples

```
Main> rankBeats Jack (Numeric 7)
```

```
True
```

```
Main> rankBeats (Numeric 10) Queen
```

```
False
```

Testing

We can write tests in GHCi, or we can *automate* tests

```
import Test.QuickCheck

prop_RankBeats a b =
  rankBeats a b || rankBeats b a
```

```
*Main> quickCheck prop_RankBeats
*** Failed! Falsifiable (after 12 tests):
Jack
Jack
```

Correcting the Property

In this case the *test* is wrong:

```
import Test.QuickCheck

prop_RankBeats a b =
  a/=b ==> rankBeats a b || rankBeats b a
```

If $a \neq b$ then...

Used *only* in
QuickCheck
tests

```
*Main> quickCheck prop_RankBeats
+++ OK, passed 100 tests.
```


Modelling a Card

- A Card has both a Rank and a Suit

```
data Card = Card Rank Suit  
deriving Show
```

- Define functions to inspect both

```
rank :: Card -> Rank  
rank (Card r s) = r  
  
suit :: Card -> Suit  
suit (Card r s) = s
```

A Useful Abbreviation

- Define type and inspection functions together, as follows

```
data Card = Card {rank :: Rank, suit :: Suit}  
deriving Show
```

When does one card beat another?

- When both cards have the same suit, and the rank is higher

can be written
down simpler...

```
cardBeats :: Card -> Card -> Bool
cardBeats c c'
  | suit c == suit c' = rankBeats (rank c) (rank c')
  | otherwise         = False
```

```
data Suit = Spades | Hearts | Diamonds | Clubs
deriving (Show, Eq)
```

When does one card beat another?

- When both cards have the same suit, and the rank is higher

```
cardBeats :: Card -> Card -> Bool
cardBeats c c' = suit c == suit c'
                && rankBeats (rank c) (rank c')
```

Intermezzo: Figures

- Modelling geometrical figures
 - triangle
 - rectangle
 - circle

```
data Figure = Triangle ...  
          | Rectangle ...  
          | Circle ...
```

```
circumference :: Figure -> Double  
circumference = ...
```

Intermezzo: Figures

```
data Figure = Triangle Double Double Double  
           | Rectangle Double Double  
           | Circle { radius:: Double}
```

```
circumference :: Figure -> Double  
circumference (Triangle a b c) = a + b + c  
circumference (Rectangle x y) = 2 * (x + y)  
circumference c                = 2 * pi * radius c
```

Intermezzo: Figures

```
data Figure = Triangle Double Double Double
           | Rectangle Double Double
           | Circle Double

-- types
Triangle :: Double -> Double -> Double -> Figure
Rectangle :: Double -> Double -> Figure
Circle :: Double -> Figure
```

```
square :: Double -> Figure
square s = Rectangle s s
```

Modelling a Hand of Cards

From 2017 and onwards we will be simply using lists to model a hand of cards type `Hand = [Card]`

- A hand may contain any number of cards from zero up!

```
data Hand = Cards Card ... Card  
deriving Show
```

We can't use
...!!!

- The solution is... *recursion!*

Modelling a Hand of Cards

- A hand may contain any number of cards from zero up!
 - A hand may be empty
 - It may consist of a *first card* and the
 - The rest is another hand of cards!

very much like a list...

```
data Hand = Empty | Add Card Hand  
deriving Show
```

A recursive type!

Solve the problem of modelling a hand with one fewer cards!

When can a hand beat a card?

- An empty hand beats nothing
- A non-empty hand can beat a card if the first card can, *or* the rest of the hand can!

```
handBeats :: Hand -> Card -> Bool
handBeats Empty    card = False
handBeats (Add c h) card =
    cardBeats c card || handBeats h card
```

- *A recursive function!*

Let's automate choosing a card...

```
chooseCard :: Card -> Hand -> Card
```

The card to beat

The card we play

How will I test it?

```
prop_chooseCardWinsIfPossible c h =  
  handBeats h c == cardBeats (chooseCard c h) c
```

LIVE CODING!!!

What Did We Learn?

- Modelling the problem using datatypes with components
- Using *recursive datatypes* to model things of varying size
- Using *recursive functions* to manipulate recursive datatypes
- An introduction to testing with properties