Modelling & Datatypes

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*

- 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

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

Needed to print values

- **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

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

Rank Beats Rank

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

Nothing beats an Ace

Matches anything at all

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

```
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

```
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
```

Matches `Numeric 7`, for example

Names the number in the rank

Examples

```
Main> rankBeats Jack (Numeric 7)
True
Main> rankBeats (Numeric 10) Queen
False
```

Further reading exercise: possible to make a much simpler definition by getting Haskell to derive the ordering relations <, <= etc. between cards.

- Find out more about "deriving Ord"...

A Property

- Either a beats b or b beats a

```
prop_rankBeats a b = rankBeats a b || rankBeats b a
```

```
Main> quickCheck prop_rankBeats
ERROR - Cannot infer instance
*** Instance   : Arbitrary Rank
*** Expression : quickCheck prop_rankBeats
```

QuickCheck doesn't know how to choose an arbitrary Rank!

QuickCheck Generators

- Test data is chosen by a *test data generator*
- Writing generators we leave for the future

Testing the Property

```
prop_rankBeats a b = rankBeats a b || rankBeats b a
```

```
Main> quickCheck prop_rankBeats
Falsifiable, after 9 tests:
King
King
```

Provided they're not equal

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

data Rank = Numeric Integer | Jack | Queen | King | Ace
  deriving (Show, Eq)
```

Define == for ranks

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

- The previous type and function definitions can be written in an equivalent abbreviated form:

```
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 more simply...

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

Modelling a Hand of Cards

- 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 rest
    - The rest is another hand of cards!

```
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!

Trickier Example: Choose a card to play

- Given
  - Card to beat
  - The hand
- Beat the card if possible!

Strategy

- If the hand is only one card, play it
- If there is a choice,
  - Select the best card from the *rest* of the hand
  - Choose between it and the first card
- Principles
  - Follow suit if possible
  - Play lowest *winning* card if possible
  - Play lowest *losing* card otherwise

The Code

```
-- chooseCard beat hand chooses a smallest card from hand to
-- play and beat is the card to be beaten
chooseCard :: Card -> Hand -> Hand
chooseCard beat (Add c Empty) = c
chooseCard beat (Add c rest)
    | suit c==suit beat && suit c'/= suit beat  = c
    | suit c/=suit beat  && suit c'==suit beat = c'
    | rankBeats (rank c) (rank c')             = c'
    | otherwise                                = c
   where c' = chooseCard beat rest
```

Properties of chooseCard

- Complicated code with great potential for errors!
- Possible properties:
  - chooseCard returns a card from the hand ("no cards up the sleeve")
  - chooseCard follows suit if possible ("no cheating")
  - chooseCard always wins if possible

Testing chooseCard

```
prop_chooseCardWinsIfPossible c h =
   h/=Empty ==>
      handBeats h c
      ==
      cardBeats (chooseCard c h) c
```

```
Main> quickCheck prop_chooseCardWinsIfPossible
Falsifiable, after 3 tests:
Card{rank=Numeric 8,suit=Diamonds}
Add Card{rank=Numeric 4,suit=Diamonds} (Add
   Card{rank=Numeric 10,suit=Spades} Empty)
```

What went wrong?

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
- Writing properties of more complex algorithms

Reminder: Modelling a Hand

- A Hand is either:
  - An empty hand
  - Formed by *adding a card* to a smaller hand

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

- Discarding the first card:

```
discard :: Hand -> Hand
discard (Add c h) = h
```

Lists
-- how they work

Lists: recap

- Can represent 0, 1, 2, … things
  - [], [3], [”apa”,”katt”,”val”,”hund”]
- They all have the same type
  - [1,3,True,”apa”] is not allowed
- The order matters
  - [1,2,3] /= [3,1,2]
- Syntax
  - 5 : (6 : (3 : [])) == 5 : 6 : 3 : [] == [5,6,3]
  - ”apa” == [’a’,’p’,’a’]

Can we define Lists as a datatype?

```
data List = Empty | Add ?? List
```

- Our attempt at a ”home made” list is either:
  - An empty list
  - Formed by *adding an element* to a smaller list
- What to put on the place of the ??

Lists

```
data List a = Empty | Add a (List a)
```

A type parameter

- Add 12 (Add 3 Empty) :: List Int
- Add ”apa” (Add ”bepa” Empty) :: List String
- Haskell’s built-in lists can be thought of as a syntactic shorthand for this datatype

Lists

```
data List a = Empty | Add a (List a)
```

- Empty :: List Integer
- Empty :: List Bool
- Empty :: List String
- ...

More on Types

- Functions can have ”general” types:
  - *polymorphism*
  - reverse :: [a] -> [a]
  - (++) :: [a] -> [a] -> [a]
- Sometimes, these types can be restricted
  - Ord a => … for comparisons (<, <=, >, >=, …)
  - Eq a => … for equality (==, /=)
  - Num a => … for numeric operations (+, -, *, …)

Do's and Don'ts

```
isBig :: Integer -> Bool
isBig n | n > 9999  = True
        | otherwise = False
```

guards and boolean results

```
isBig :: Integer -> Bool
isBig n = n > 9999
```

Do's and Don'ts

```
resultIsSmall :: Integer -> Bool
resultIsSmall n = isSmall (f n) == True
```

comparison with a boolean constant

```
resultIsSmall :: Integer -> Bool
resultIsSmall n = isSmall (f n)
```

Do's and Don'ts

```
resultIsBig :: Integer -> Bool
resultIsBig n = isSmall (f n) == False
```

comparison with a boolean constant

```
resultIsBig :: Integer -> Bool
resultIsBig n = not (isSmall (f n))
```

Writing Code

- Beautiful code
  - readable
  - not overly complicated
  - no repetitions
  - no ”junk” left
- For
  - you
  - other people