Functional Programming

Type definitions

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2022/23

Type aliases

Explaining the meaning of data in comments is bad!
Introduce new, self explaining types.

```haskell
type Name = String
type Title = String
type Year = Int
type Age = Int

type User = (Name, Year)
-- name, year of birth
type Film = (Title, Age)
-- ^ fsk
type Purchase = (Name -- use name
                , Title -- item name
                , Year) -- date of purchase
users :: [User]
```

Datatypes

Example scenario

- model a card game (hearts)
- represent the game items!
- define game logic on the representations!

Intermezzo: The game

Microsoft Hearts (Wikipedia link)

- computer game based on card game "Hearts"
- included in Windows 3.1 through Windows 7
- discontinued

Gameplay

- four players (three simulated)
- trick-taking game
- each player plays one card to a trick
- trick won by highest card of the suit led; no Trump!
- suit must be followed
- Heart cannot lead until
  - either Heart has been broken — a player played Heart
  - or the leading player has only Heart
- points are scored by any Hearts (1 point) and the Queen of Spades (13 points)

Gameplay

- four players (three simulated)
- trick-taking game
- each player plays one card to a trick
- trick won by highest card of the suit led; no Trump!
- suit must be followed
- Heart cannot lead until
  - either Heart has been broken — a player played Heart
  - or the leading player has only Heart
- points are scored by any Hearts (1 point) and the Queen of Spades (13 points)

Objective

- Avoid gaining points **or** gain all 26 points

Data model for card games

- A card has a **Suit** and a **Rank**
- A card beats another card if it has the same suit, but higher rank
- Todo:
  - represent cards
  - define when one card beats another
  - define a function that chooses a beating card from a hand of cards, if possible

Model using algebraic datatypes

A card has a Suit

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

Explanation

- define an *algebraic datatype*
- new type consisting of (exactly) four values
- Suit: the name of the new type
- Spades, Hearts, ...: the names of its **constructors**
- constructors can be used in expressions and patterns
- names of types and constructors must be capitalized

Printing algebraic datatypes

```
Main> Spades

<interactive>:3:1:
  No instance for (Show Suit) arising from
  a use of ‘print’
  Possible fix: [...]
```

Oops!

- Haskell does not know how to print a `Suit`
- but we can ask for a default (or write our own printer)

Printing derived

```haskell
data Suit = Spades | Hearts | Diamonds | Clubs
  deriving (Show) -- makes 'Suit' printable
```

Defines a function **show** for Suit, which is automatically called by Haskell's printer

```
Main> Spades
Spades
Main> show Spades
"Spades"
Main> :t show
show :: Show a => a -> String
```

Remark

- **Show** is a *type class*
- a type class associates one or more functions with a type; in case of **Show**, the function is **show**

Functions on data types

Each suit has a color:

```
data Color = Black | Red
  deriving (Show)
```

Define a color function by pattern matching

```
color :: Suit -> Color
color = undefined
```

More data

A card has a suit and a **rank**:

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

The constructor **Numeric** is different: it takes an argument.

```
Main> :t Numeric
Numeric :: Integer -> Rank
```

Comparing ranks

Situation

- Let `r2` be the highest rank on the table
- Let `r1` be the card played
- Assuming the suits match, does `r1` get the trick?

Comparing ranks

Situation

- Let `r2` be the highest rank on the table
- Let `r1` be the card played
- Assuming the suits match, does `r1` get the trick?

Need an ordering of ranks

```haskell
-- |rankBeats r1 r2
-- returns True, if r1 beats r2
-- i.e. r1 is strictly greater than r2
rankBeats :: Rank -> Rank -> Bool
rankBeats r1 r2 = undefined
```

Ordering ranks by pattern matching

```
-- rankBeats r1 r2 returns True, if r1 beats r2
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 n1) (Numeric n2) = n1 > n2
-- pattern match on Numeric constructor yields its argument
```

Letting Haskell order the ranks

- definition of `rankBeats` is repetitive
- boilerplate code
- let Haskell generate it for us!

Deriving an order

The comparison operators <=, < etc are overloaded and can be extended to new types

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

rankBeats' r1 r2 = r1 > r2
```

Remark

- **Ord** is another type class that governs <, <=, etc

Oops. . .

```
M04Cards.hs:17:27: error:
    • No instance for (Eq Rank)
        arising from the 'deriving' clause of a data type decl
      Possible fix:
        use a standalone 'deriving instance' declaration,
          so you can specify the instance context yourself
    • When deriving the instance for (Ord Rank)
```

Oops...

```
M04Cards.hs:17:27: error:
    • No instance for (Eq Rank)
        arising from the 'deriving' clause of a data type decl
      Possible fix:
        use a standalone 'deriving instance' declaration,
          so you can specify the instance context yourself
    • When deriving the instance for (Ord Rank)
```

Explanation

Type class **Ord** defines < and then

```
x <= y = x < y || x == y
```

but how do we compare two ranks for equality?

Equality of ranks

- could be defined by pattern matching, but
- let Haskell generate this boilerplate code for us!

Deriving equality

==, /= are overloaded and can be extended to new types

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

rankBeats' r1 r2 = r1 > r2
```

Equality of ranks

- could be defined by pattern matching, but
- let Haskell generate this boilerplate code for us!

Deriving equality

==, /= are overloaded and can be extended to new types

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

rankBeats' r1 r2 = r1 > r2
```

Are they the same?

How do we know that rankBeats = rankBeats'? Let's defer that.

Cards, finally

A card has a Suit and a Rank

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

rank :: Card -> Rank
rank (Card r s) = r

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

- Card has single constructor with two parameters
- (in principle, a tuple with a special name)
- rank, suit are **selector functions**

Alternative definition of Cards

There is a way to define the type along with its selector functions using Haskell's (hated) records types:

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

- defines type `Card` and its constructor `Card`
- defines selectors `rank :: Card -> Rank` and `suit :: Card -> Suit`
- we can use *record notation* to construct values:

```haskell
queenOfSpades = Card{ rank= Queen, suit= Spades }
```

- and *record updates*:

```haskell
queenOfHearts = queenOfSpades { suit= Hearts }
```

Comparing Cards

A card **beats** another card, if it has the same suit, but a higher rank

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

Hand of Cards

```haskell
type Hand = [Card]

chooseCard :: Card -> Hand -> Card
chooseCard givenCard h = undefined
```

To develop `chooseCard` refine `h` by pattern matching

Choose a card

```
type Hand = [Card]

chooseCard :: Card -> Hand -> Card
chooseCard givenCard [] = undefined -- ???
chooseCard givenCard (x:xs) = undefined
```

- What should we do if the hand is empty?
- Avoid by defining only non-empty hands!

Non-empty hands

```
data Hand = Last Card | Next Card Hand
  deriving (Show, Eq)
```

- Recursive datatype definition
- Last Card is the *base case*

Get card from non-empty hand

- A Hand is never empty
- Thus we can always obtain a card

```
topCard :: Hand -> Card
topCard (Last c) = c
topCard (Next c _) = c
```

Choosing from non-empty hand

```
-- choose a beating card, if possible
chooseCard :: Card -> Hand -> Card
chooseCard = undefined
```

Choosing from non-empty hand

```haskell
-- choose a beating card, if possible
chooseCard :: Card -> Hand -> Card
chooseCard gc (Last c) = c -- may beat, or not
chooseCard gc (Next c h) | cardBeats gc c = c
                         | otherwise = chooseCard gc h
```

Questions?

?