Functional Programming / Lean

Type definitions

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2026/27

Type aliases

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

```
1  abbrev Name := String
2  abbrev Title := String
3  abbrev Year := Int
4  abbrev Age := Int
5
6  abbrev User := Name × Year
7  --           name   year of birth
8
9  abbrev Film := Title × Age
10 --                  FSK age rating
11
12 abbrev Purchase := Name × Title × Year
13 --                 user   item    year of purchase
14 def users : List User :=
15   [("Alice", 1990), ("Bob", 1985)]
16 def purchase : Purchase := ("Alice", "Casablanca", 2026)
```

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

```
1 inductive Suit where | spades | hearts | diamonds | clubs
```

Explanation

- define an *inductive 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
- convention: names of types in UpperCamlCase, constructors in lowerCamlCase

Printing and Equality derived

```
1 inductive Suit where | spades | hearts | diamonds | clubs
2   deriving BEq
```

Defines functions for `Suit`, which are automatically called by equality

Remark

- `BEq` is a *type class*
- a type class associates one or more functions with a type; in case of `BEq`, the function is $==$, which returns a `Bool`.

Functions on data types

Each suit has a color:

```
inductive Color where | black | red
  deriving Repr, DecidableEq
```

Define a color function by pattern matching

```
def color : Suit := sorry
```

More data

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

```
1 inductive Rank where
2   | numeric (val : Nat) | jack | queen | king | ace
```

The constructor `numeric` is different: it takes an argument.

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

```
1 -- |rankBeats r1 r2
2 -- returns True, if r1 beats r2
3 -- i.e. r1 is strictly greater than r2
4 def rankBeats : Rank -> Rank -> Bool := sorry
```

Ordering ranks by pattern matching

```
1  -- rankBeats r1 r2 returns True, if r1 beats r2
2  def rankBeats : Rank → Rank → Bool
3    | .numeric n, .numeric m => n > m
4    | .ace, _ => true
5    | _, .ace => false
6    | .jack, _ => true
7    | _, .jack => false
8    | .queen, _ => true
9    | _, .queen => false
10   | .king, _ => true
11   | _, .king => false
```

Cards, finally

A card has a `Suit` and a `Rank`

```
1 structure Card where
2   suit : Suit
3   rank : Rank
```

- `Card` has single constructor with two parameters
- `Card.rank`, `Card.suit` are **selector functions**
- alternatively, use dot notation: if `card : Card`, then `card.suit` etc

Comparing Cards

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

```
1 def cardBeats : Card -> Card -> Bool := sorry
```

Hand of Cards

```
1 abbrev Hand := List Card
2
3 def chooseCard : Hand -> Card := sorry
```

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

Choose a card

```
1 abbrev Hand := List Card
2
3 def chooseCard : Hand -> Card
4   | [] => sorry
5   | c :: _ => c
```

- Can't finish! What should we do if the hand is empty?
- Avoid by defining only non-empty hands!

Non-empty hands

```
1 inductive Hand' where
2   | last : Card -> Hand'
3   | next : Card -> Hand' -> Hand'
```

Get card from non-empty hand

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

```
1 def topCard : Hand' -> Card
2   | .last c => c
3   | .next c _ => c
```

Choosing from non-empty hand

```
1 -- choose a beating card, if possible
2 def chooseCard' : Card -> Hand' -> Card := sorry
```

Choosing from non-empty hand

```
1 def chooseCard' : Card → Hand' → Card
2   | c, .last c' => c'
3   | c, .next c' h' => if cardBeats c' c then c' else chooseCard' c h'
```

Questions?

