COMP90048: Workshop 2

Eleanor McMurtry, University of Melbourne

Using Haskell Types for Great Good

Intro

- One of Haskell's greatest strengths is an *expressive type system*.

- Problems can be massively simplified by creating appropriate types.

The `data` keyword

```
data Coin = Heads | Tails
```

The `data` keyword

```
data Coin = Heads | Tails

data Die = One | Two | Three | Four | Five | Six
```

The `data` keyword

```
data Coin = Heads | Tails

data Die = One | Two | Three | Four | Five | Six

data Student = Student String Integer
```

The `data` keyword

```
data Coin = Heads | Tails
```

union, **enum-like**, or **sum** type

```
data Die = One | Two | Three | Four | Five | Six
```

```
data Student = Student String Integer
```

struct-like or **product** type

The `data` keyword (“algebraic data types”)

```
data Coin = Heads | Tails
```

union, **enum-like**, or **sum** type

```
data Die = One | Two | Three | Four | Five | Six
```

```
data Student = Student String Integer
```

struct-like or **product** type

$$\{Heads\} \sqcup \{Tails\}$$

$$C^* \times \mathbb{Z}$$

(where C is the set of characters)

Types can be *recursive*...

```
data IntList = Cons Int IntList | Empty
```

… and generic!

```
data IntList = Cons Int IntList | Empty

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

Type aliases

```
type StudentId = Integer

type StudentName = String

data Student = Student StudentName StudentId
```

A deck of cards

What are some different ways to represent a Western deck of playing cards?

Optional data

```
data Maybe a = Just a | Nothing
```

(This is already in the Prelude.)

Continue with Grok