CS653

Mao: A Game of Rules

Milind Luthra (150363)

April 30, 2018

What is Mao?

- Card game with no set rules.

What is Mao?

- Card game with no set rules.
- Instead, rules are added at play time.

What is Mao?

- Card game with no set rules.
- Instead, rules are added at play time.
- Winner of the previous round adds one rule at the start of the next round.

What is Mao?

- Card game with no set rules.
- Instead, rules are added at play time.
- Winner of the previous round adds one rule at the start of the next round.
- Rules can be used to change objectives, win conditions, even make it into something entirely unrelated to card games.

Description of a Card Game

- The main unit of a game is a *Turn*. We keep having *Turns* until one of the player wins.

Description of a Card Game

- The main unit of a game is a *Turn*. We keep having *Turns* until one of the player wins.
- A *Turn* is a set of *Stages*, for instance, a "betting stage", followed by a "reveal stage" and so on.

Description of a Card Game

- The main unit of a game is a *Turn*. We keep having *Turns* until one of the player wins.
- A *Turn* is a set of *Stages*, for instance, a "betting stage", followed by a "reveal stage" and so on.
- Each *Stage* is a set of *Rules*, which are applied to each player.

Description of a Card Game

- The main unit of a game is a *Turn*. We keep having *Turns* until one of the player wins.
- A *Turn* is a set of *Stages*, for instance, a "betting stage", followed by a "reveal stage" and so on.
- Each *Stage* is a set of *Rules*, which are applied to each player.
- Some basic structures like *Cards*, *Tokens* and *Decks* are provided and additional ones can be defined.

Two Main Components

- A way to write rules easily: an **Embedded DSL**.

Two Main Components

- A way to write rules easily: an **Embedded DSL**.
  - The `do` syntax of Haskell lends itself very nicely to the creation of a 'language of rules'.

Two Main Components

- A way to write rules easily: an **Embedded DSL**.
  - The do syntax of Haskell lends itself very nicely to the creation of a 'language of rules'.
  - Predefined directives can provide syntactic sugar for commonly performed actions inside a rule.

Two Main Components

- A way to write rules easily: an **Embedded DSL**.
  - The `do` syntax of Haskell lends itself very nicely to the creation of a 'language of rules'.
  - Predefined directives can provide syntactic sugar for commonly performed actions inside a rule.
  - While the full power of the Haskell language is available to anyone who wants it.

Two Main Components

- A way to write rules easily: an **Embedded DSL**.
  - The `do` syntax of Haskell lends itself very nicely to the creation of a 'language of rules'.
  - Predefined directives can provide syntactic sugar for commonly performed actions inside a rule.
  - While the full power of the Haskell language is available to anyone who wants it.
- And a way to interpret and use these rules at runtime. For this, we use a library called `hint`.

How Does a Rule Look Like?

```
topMostCard <- pop from "mainDeck";
out topMostCard;
outS "Would you like to play this card?";
willPlay <- inp (t :: Bool);
if willPlay
  then push topMostCard to "sideDeck";
  else push topMostCard to "discardDeck";
```

Haskell Feature Usage

- **Dynamic and Typeable**

Haskell Feature Usage

- **Dynamic and Typeable**
  - Haskell usually deals with types at compile time.

Haskell Feature Usage

- **Dynamic and Typeable**
  - Haskell usually deals with types at compile time.
  - We can try providing common datatypes like `Deck/Token` to help users while they are coming up with rules.

Haskell Feature Usage

- **Dynamic and Typeable**
  - Haskell usually deals with types at compile time.
  - We can try providing common datatypes like `Deck/Token` to help users while they are coming up with rules.
  - But this is not enough as the user might want to save and operate over entirely different range of types.

Haskell Feature Usage

- **Dynamic and Typeable**
  - Haskell usually deals with types at compile time.
  - We can try providing common datatypes like `Deck/Token` to help users while they are coming up with rules.
  - But this is not enough as the user might want to save and operate over entirely different range of types.
  - Thus, we need dynamic types!

Haskell Feature Usage

- **Dynamic and Typeable**
  - Haskell usually deals with types at compile time.
  - We can try providing common datatypes like `Deck/Token` to help users while they are coming up with rules.
  - But this is not enough as the user might want to save and operate over entirely different range of types.
  - Thus, we need dynamic types!
  - Using the `Typeable` typeclass, we can make sure that objects are correctly typed when converted to and from the `Dynamic` type.

Haskell Feature Usage

- **Dynamic and Typeable**
  - Haskell usually deals with types at compile time.
  - We can try providing common datatypes like `Deck/Token` to help users while they are coming up with rules.
  - But this is not enough as the user might want to save and operate over entirely different range of types.
  - Thus, we need dynamic types!
  - Using the `Typeable` typeclass, we can make sure that objects are correctly typed when converted to and from the `Dynamic` type.
  - Need to use language extension `DeriveDataTypeable`.

Haskell Feature Usage

- **Monad Transformers**

Haskell Feature Usage

- **Monad Transformers**
  - Two monads, one a `State` monad to maintain the state of the game, and an `IO` monad.

Haskell Feature Usage

- **Monad Transformers**
  - Two monads, one a `State` monad to maintain the state of the game, and an `IO` monad.
  - Might need to use them together (a rule may contain IO as well).

Haskell Feature Usage

- **Monad Transformers**
  - Two monads, one a `State` monad to maintain the state of the game, and an `IO` monad.
  - Might need to use them together (a rule may contain IO as well).
  - Monad transformers provide a way to "wrap" a monad around other monads.

Haskell Feature Usage

- **Monad Transformers**
  - Two monads, one a `State` monad to maintain the state of the game, and an `IO` monad.
  - Might need to use them together (a rule may contain IO as well).
  - Monad transformers provide a way to "wrap" a monad around other monads.
  - For instance, `StateT GameState IO ()` wraps a state around an IO.

Haskell Feature Usage

- **Monad Transformers**
  - Two monads, one a `State` monad to maintain the state of the game, and an `IO` monad.
  - Might need to use them together (a rule may contain IO as well).
  - Monad transformers provide a way to "wrap" a monad around other monads.
  - For instance, `StateT GameState IO ()` wraps a state around an IO.
  - To use IO, we can use `lift` or `liftIO`.

Haskell Feature Usage

- Record syntax for data constructors.

Haskell Feature Usage

- Record syntax for data constructors.
  - Makes creation of new data with a few modification easy.

Haskell Feature Usage

- Record syntax for data constructors.
  - Makes creation of new data with a few modification easy.
  - For instance, `gameState' = gameState  winner = Just P1`

Haskell Feature Usage

- Record syntax for data constructors.
  - Makes creation of new data with a few modification easy.
  - For instance, `gameState' = gameState  winner = Just P1`
- `Data.Map` for map implementation which would be better/faster than using list of tuples.

Gameplay

Demo Time!

References

- Reference for modelling in terms of stages/turns (I only used the basic idea, this has a lot more)

References

- Reference for modelling in terms of stages/turns (I only used the basic idea, this has a lot more)
- Libraries used: mtl, random-shuffle, hint, containers (See stackage/hackage for all of these)

References

- Reference for modelling in terms of stages/turns (I only used the basic idea, this has a lot more)
- Libraries used: mtl, random-shuffle, hint, containers (See stackage/hackage for all of these)
- I generated some of the pointfree forms of functions using lambdabot.