Announcements

"Everything should be made as simple as possible, but not simpler."

— attributed to Albert Einstein

Haskell covered since midterm

- `type` defines a type name as an abbreviation for other types
- `data` defines new data structures (and a type) and constructors / deconstuctors
- `IO t` is the input/output monad
- `do` can be used to sequence input/output operations

Overview

Last class:

- Abstraction for games, so we can write interfaces and solvers for any games that fit the abstraction
- Representation of magic-sum game and count game
- A simple human interface for the abstraction
- A generic solver for the abstraction

Today:

- Make it more efficient
- Abstract data types
- Threading state

Games

- Players observe state and make actions
- Games take actions and update state of game, perhaps finishing.

```
type Game = Action -> State -> Result

type Player = State -> Action

data State = State InternalState [Action]
         deriving (Ord, Eq, Show)

data Result = EndOfGame Double State
            | ContinueGame State
         deriving (Eq, Show)
```

See MagicSum.hs Play.hs CountGame.hs

Magic Sum Game

- players take turns choosing different numbers in range [0..9]
- first player to have 3 numbers that sum to 15 wins
- tie if they run out of numbers to play

To use Play.hs we need to define:

- Action
- Internal State
- The Game Function

Counting Game

- Players can choose number in a fixed set, e.g., $\{1, 2, 3, 5, 7\}$
- Internal state is a number
- When a player plays an action i the state is incremented by i.
- Player loses if the internal state is greater than or equal to a break value (e.g., 20 or 21).

Minimax

- `type Game = Action -> State -> Result`
  `type Player = State -> Action`
  `mm_player:: Game -> Player`
- Minimax takes a game and a state and returns (action,value) for the best move (assuming there are moves available)
- `minimax:: Game -> State -> (Action, Double)`
  `valueact :: Game -> State -> Action -> Double`
  `value:: Game -> Result -> Double`
- The value is either:
  - the value for the end of the game, or
  - the negation of the value for the opponent (who now plays)
- `mm_player game state = fst ( minimax game state)`
- See `Minimax.hs` (run the test cases)

Clicker Question

```
argmax2 :: Ord v => (e -> v) -> [e] -> (e,v)
argmax2 f (h:t) =
   foldr (\ e (et,vt) -> let fe = f e in
                         if (fe > vt) then (e,fe)
                                      else (et,vt))
         (h, f h) t
```

What is not true about this:

- A `argmax2 f lst` returns a pair
- B It computes f of each element exactly once
- C It works for every list that type checks
- D It takes the first element from the list to start the foldr

Clicker Question

```
argmax2 :: Ord v => (e -> v) -> [e] -> (e,v)
argmax2 f (h:t) =
   foldr (\ e (et,vt) -> let fe = f e in
                          if (fe > vt) then (e,fe)
                                       else (et,vt))
         (h, f h) t
```

If there are multiple elements with the same maximal value for the function, what is returned?

A The first (e,v) pair that is maximal

B The last (e,v) pair that is maximal

C The second (e,v) pair that is maximal

D All of the (e,v) paris that are maximal

E The last (e,v) pair that is maximal, unless the first element is maximal

Improving Minimax

(a) Limit the depth of the tree, and have an *evaluation function* estimate value of a node when search stops.

(b) learn (approximate from self-play or human play)
  (i) `State -> value` function
  (ii) valueact (Q-value)
  (iii) `State -> Action` function (policy)

(c) Run it in parallel.

(d) Cache node values rather than recomputing.

(e) Exploit symmetry.

(f) Limit the width of the tree:
  (i) Prune dominated nodes (alpha-beta pruning)
  (ii) Sample random forward passes (Monte-Carlo tree search)

- Deep Blue (beat world chess champion 1997): a, c, d, f i
- AlphaGo (beat world Go champion 2016): bi, biii, c, f ii

Improving Minimax

Try minimax with count game

```
:set +s
minimax (countGame 20 [1,2,3,5,7]) (State 0 [1,2,3,5,7])
minimax (countGame 21 [1,2,3,5,7]) (State 0 [1,2,3,5,7])
minimax (countGame 25 [1,2,3,5,7]) (State 0 [1,2,3,5,7])
minimax (countGame 30 [1,2,3,5,7]) (State 0 [1,2,3,5,7])
```

Improving Minimax by caching results

- Minimax could cache the values of states it has evaluated
- A dictionary can be used to remember values
- A dictionary maps a key to a value

  ```
  Dict k v
  ```

  is a dictionary with key type `k` and value type `v`
- Dict Interface:

  ```
  emptyDict :: Dict k v
  getval :: (Ord k) => k -> Dict k v -> Maybe v
  insertval :: (Ord k) => k -> v -> Dict k v
                              -> Dict k v
  stats :: Dict t1 t2 -> [Char]
  ```

  "abstract data type"
- Minimax can use

  ```
  Dict state (action,value)
  ```

- A binary search tree can be used to implement a dictionary

```
data BSTree k v
     = BSEmpty
     | BSNode k v (BSTree k v) (BSTree k v)
    deriving (Show)
```

– a binary search tree where k is the type of key, and v is type of value

- It can be made to follow the Dict API.
- See `TreeDict.hs`

Clicker Question

```
data BSTree k v
      = BSEmpty
      | BSNode k v (BSTree k v) (BSTree k v)
```

What is not true:

- A `k` is a type variable
- B `BSNode "Fun" 7 BSEmpty BSEmpty`
  is of type `Num v => BSTree [Char] v`
- C `BSTree` is a function that takes 2 arguments
- D When using the data structure, `k` needs to be resolved into an actual type
- E `BSNode` is a function that takes 4 arguments