The Selection Monad Transformer

Backward Induction with Mixed Strategies

Bath Mathematical Foundations Seminar
30 March 2021

Paulo Oliva
(joint work with Martín Escardó)

Queen Mary University of London

Tic-Tac-Toe

If both players play optimally, the game ends in a **draw**

Hence **any first move is an optimal move!**

But what if the opponent is likely to make **mistakes...**

Surely some first moves are better than others

How do we calculate strategies that **maximises the chances of your opponent making a mistake?**

How to do this in a **modular** / **compositional** way?

Plan

- Strong Monads
- Continuation Monad (folding back procedure)
- Selection Monad (backward induction)
- Selection Monad Transformer
- An Example (Tic-Tac-Toe)

Strong Monads

Strong Monads

A function on types $T\colon \mathrm{Type} \to \mathrm{Type}$ is a **strong monad** if we have operations:

$$\eta_X : X \to TX$$
$$(\,\cdot\,)^\dagger : (X \to TY) \to TX \to TY$$

satisfying

$$(\eta_X)^\dagger = \mathrm{id}_{TX}$$
$$f^\dagger \circ \eta_X = f \qquad (f\colon X \to TY)$$
$$(g^\dagger \circ f)^\dagger = g^\dagger \circ f^\dagger \qquad (g\colon Y \to TZ)$$

Example 1 (Lists)

$$TX = [X]$$
$$\eta(x) = [x]$$
$$f^\dagger(xs) = [y : x \in xs, y \in fx]$$
$$\begin{aligned}(\eta_X)^\dagger(xs) &= [y : x \in xs, y \in [x]] \\ &= [x : x \in xs] \\ &= [x : x \in xs]\end{aligned}$$

Strong Monads

A function on types $T\colon \text{Type} \to \text{Type}$ is a **strong monad** if we have operations:

$$\eta_X : X \to TX$$
$$(\,\cdot\,)^\dagger : (X \to TY) \to TX \to TY$$

satisfying

$$(\eta_X)^\dagger = \mathrm{id}_{TX}$$
$$f^\dagger \circ \eta_X = f \qquad (f\colon X \to TY)$$
$$(g^\dagger \circ f)^\dagger = g^\dagger \circ f^\dagger \qquad (g\colon Y \to TZ)$$

Example 2 (Distributions)

$$\Delta X = [(\mathbb{Q}, X)]$$

$$\eta(x) = [(1, x)]$$

$$f^\dagger(d) = [(p_1 p_2, y) : \ (p_1, x) \in d, (p_2, y) \in fx]$$

$$(\eta_X)^\dagger(xs) = [(p, x) : \ (p, x) \in d]$$
$$= d$$

Strong Monad Product

For any strong monad $T\colon \mathrm{Type} \to \mathrm{Type}$ we can define an operation

$$\otimes : TX \times TY \to T(X \times Y)$$

which we can iterate to obtain

$$\bigotimes_i : \Pi_i(TX_i) \to T(\Pi_i X)$$

For some strong monads even the countable iteration is well defined

In game theory applications, TX_i captures the "local" strategy at round i and the product operation is used to compose simpler strategies into more complex ones

Continuation Monad

Continuation Monad

For any type R the following type function is a strong monad (**continuation monad**)

$$K_R X = (X \to R) \to R$$

with the mappings:

$$\eta : X \to K_R X \qquad (\,\cdot\,)^\dagger : (X \to K_R Y) \to K_R X \to K_R Y$$

$$\eta(x) = \lambda\kappa^{X \to R} . \kappa(x) \qquad f^\dagger(\phi) = \lambda\kappa^{Y \to R} . \phi(\lambda x^X . (fx)(\kappa))$$

A program $p : X$ in a context $E[p] : R$ views $\lambda x . E[x] : X \to R$ as its **continuation**

In proof theory the continuation monad is related to double negation translations

In game theory it corresponding to the “folding back procedure”

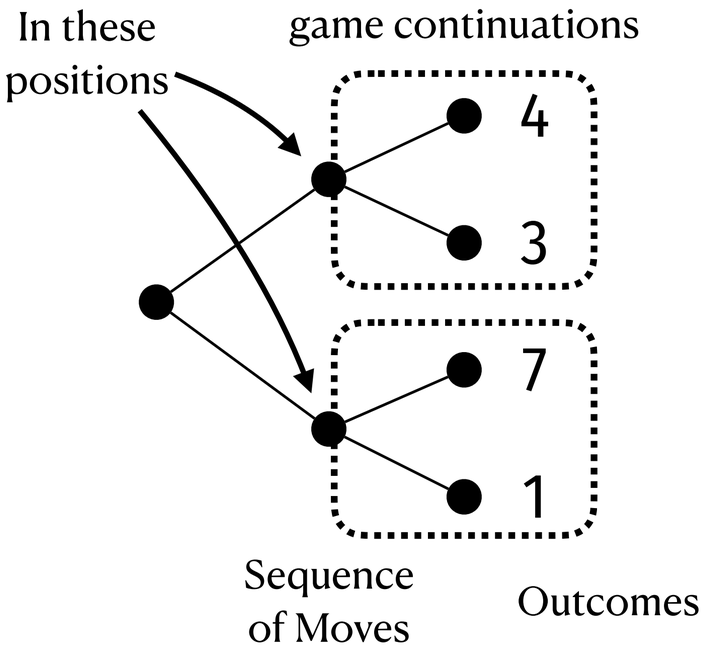
Folding Back Procedure

Given a game in extensive form (tree), and an aggregation function

$$\phi : (X \to R) \to R$$

we can “fold back” the game. For instance, suppose our aggregation function is

$$\max : (X \to \mathbb{R}) \to \mathbb{R}$$



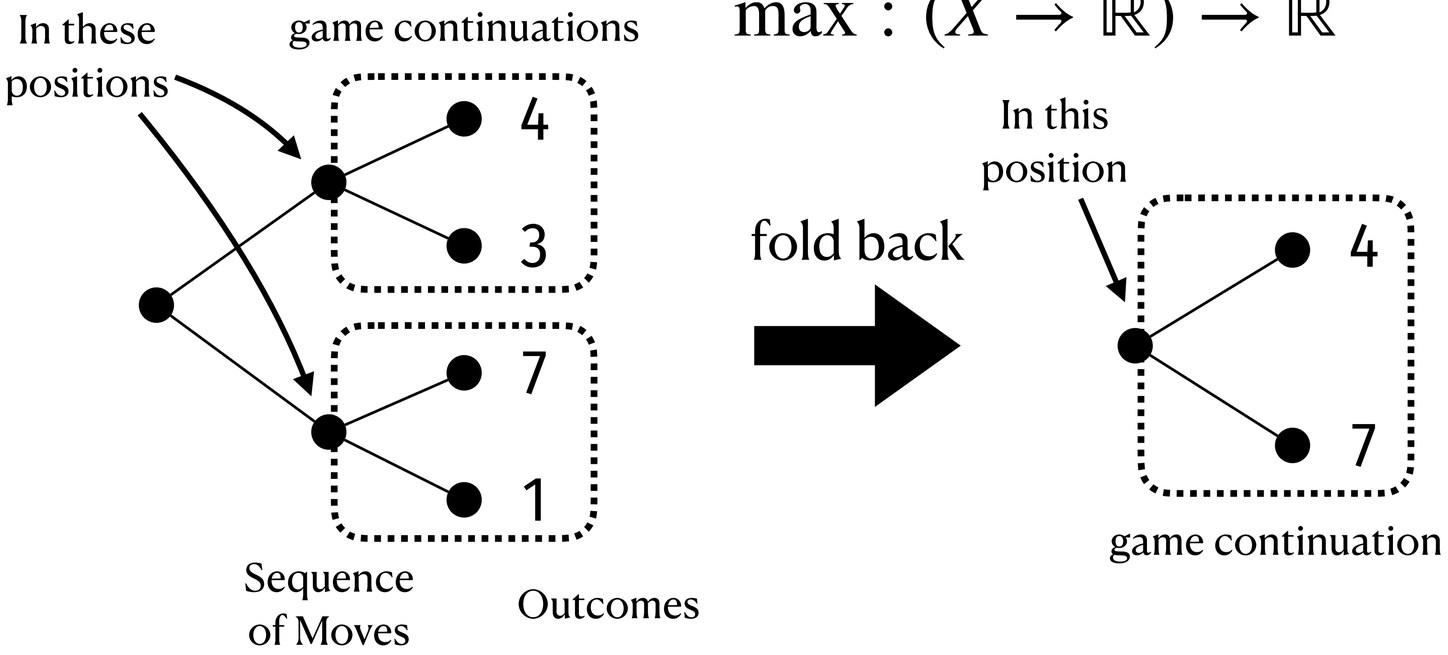
Folding Back Procedure

Given a game in extensive form (tree), and an aggregation function

$$\phi : (X \to R) \to R$$

we can “fold back” the game. For instance, suppose our aggregation function is

$$\max : (X \to \mathbb{R}) \to \mathbb{R}$$



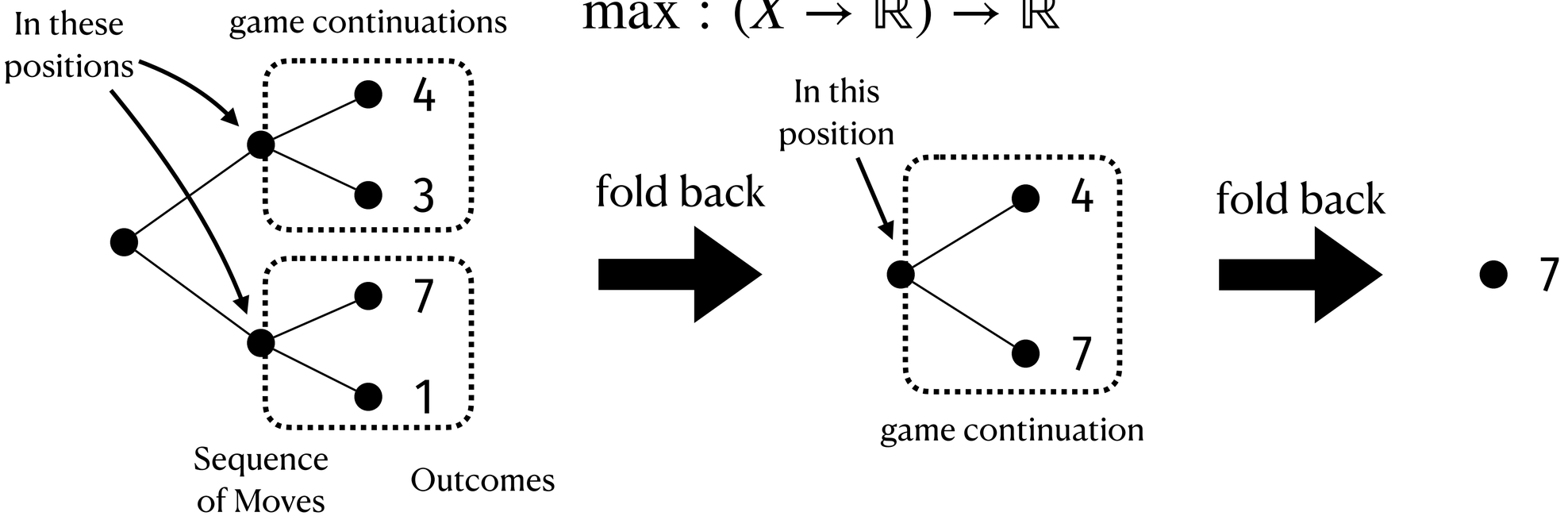
Folding Back Procedure

Given a game in extensive form (tree), and an aggregation function

$$\phi : (X \to R) \to R$$

we can “fold back” the game. For instance, suppose our aggregation function is

$$\max : (X \to \mathbb{R}) \to \mathbb{R}$$



Selection Monad

Selection Monad

For any type R the following type function is a strong monad (**selection monad**)

$$J_R X = (X \to R) \to X$$

with the mappings:

$$\eta : X \to J_R X \qquad\qquad (\,\cdot\,)^\dagger : J_R X \to (X \to J_R Y) \to J_R Y$$

$$\eta(x) = \lambda\kappa^{X\to R}\,.\,x \qquad\qquad f^\dagger(\varepsilon) = \lambda\kappa^{Y\to R}\,.\,f(a(\kappa))(\kappa)$$

where $a(\kappa) = \varepsilon(\lambda x\,.\,\kappa(f(x)(\kappa)))$

In proof theory the selection monad is related to the Peirce translation

In game theory it corresponding to “backward induction”
(computing optimal strategies in sequential games)

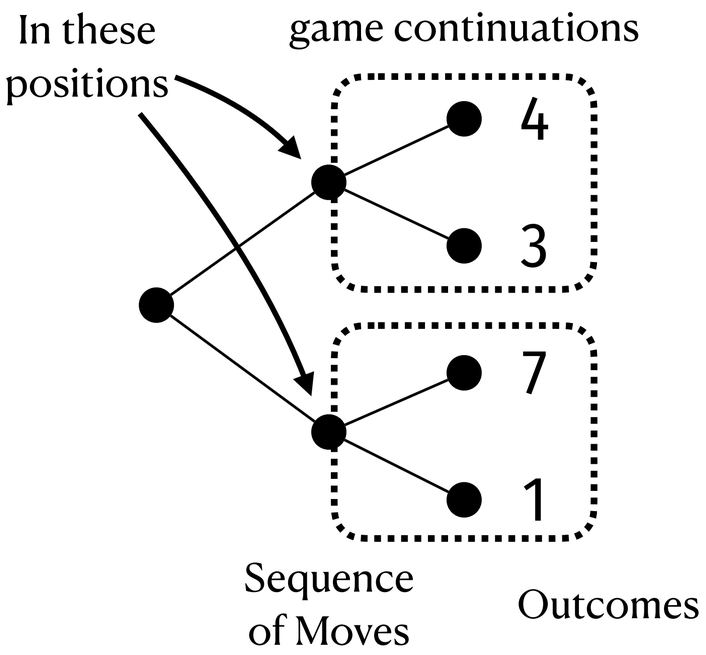
Backward Induction

Given a game in extensive form (tree), and a selection function

$$\varepsilon\colon (X \to R) \to X$$

we can calculate optimal plays. For instance, suppose our selection function is

$$\operatorname{argmax}\colon (X \to \mathbb{R}) \to X$$



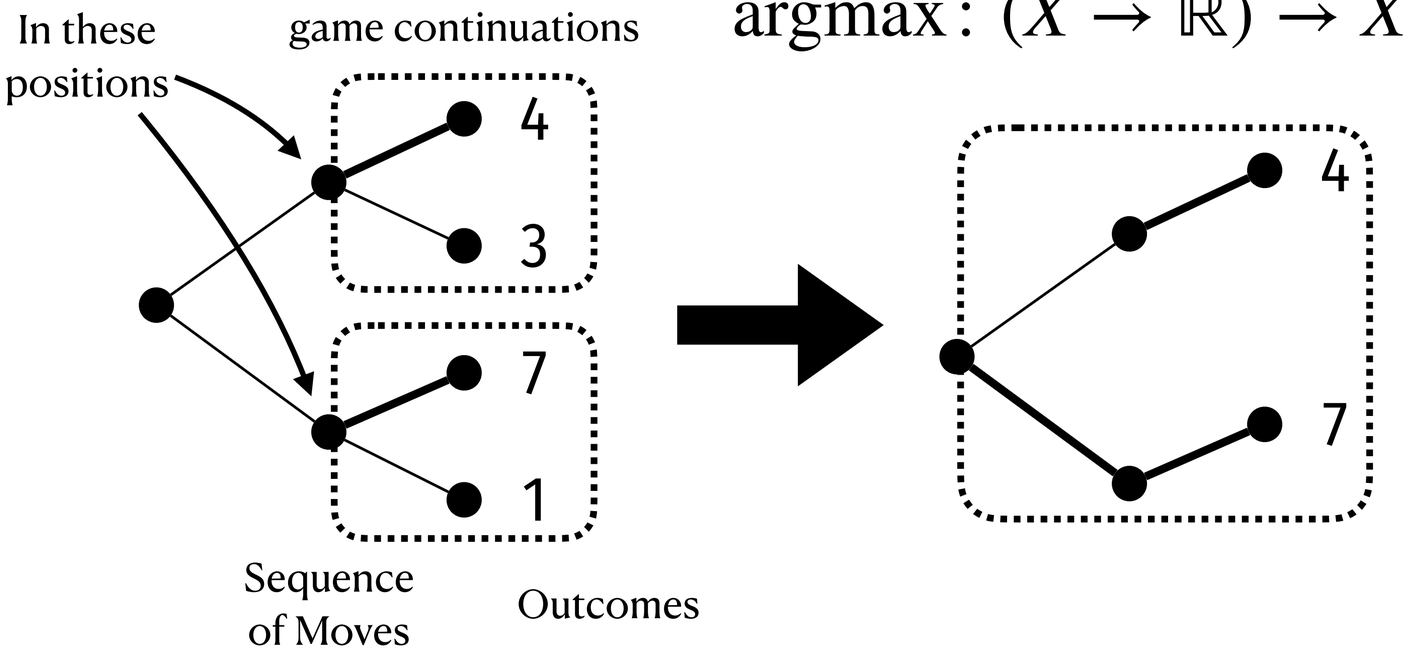
Backward Induction

Given a game in extensive form (tree), and a selection function

$$\varepsilon\colon (X \to R) \to X$$

we can calculate optimal plays. For instance, suppose our selection function is

$$\operatorname{argmax}\colon (X \to \mathbb{R}) \to X$$



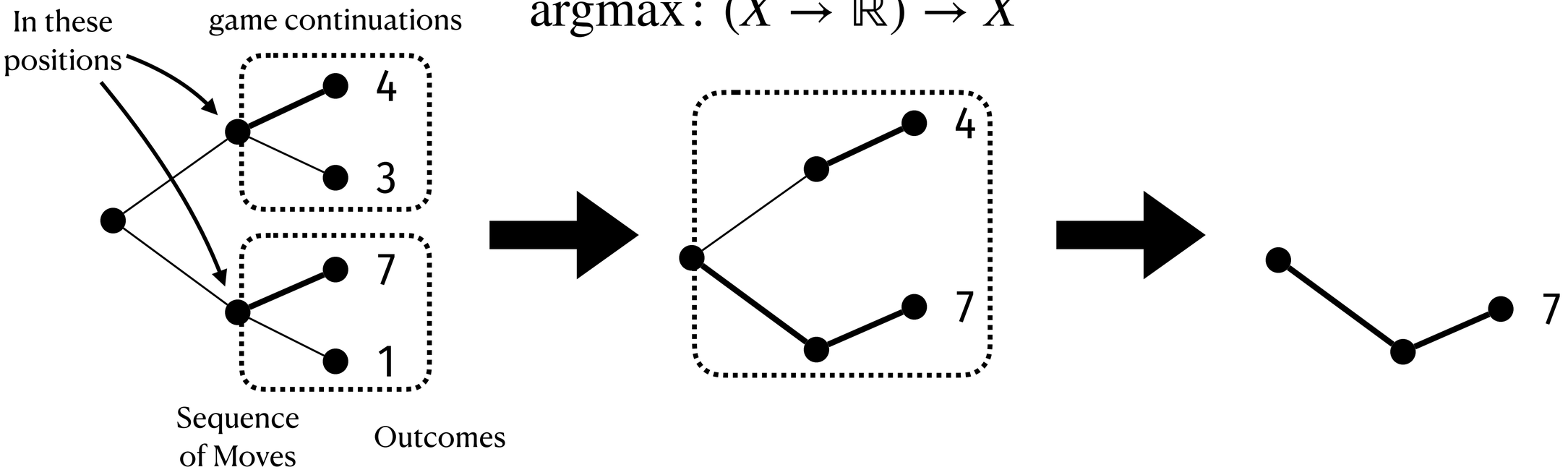
Backward Induction

Given a game in extensive form (tree), and a selection function

$$\varepsilon\colon (X \to R) \to X$$

we can calculate optimal plays. For instance, suppose our selection function is

$$\operatorname{argmax}\colon (X \to \mathbb{R}) \to X$$



Selection Monad Transformer

Selection Monad Transformer

For any strong monad M and type R the following is also a strong monad

$$J_R^M X = (X \to MR) \to MX$$

Non-deterministic players

$$J_R^{\mathscr{P}} X = (X \to \mathscr{P}R) \to \mathscr{P}X$$

For each move we have a set of possible outcomes

choose a set of move

Stochastic players

$$J_R^{\Delta} X = (X \to \Delta R) \to \Delta X$$

For each move we have a distribution over outcomes

choose a distribution of moves

Haskell

```haskell
data J r m x = J {selection :: (x -> m r) -> m x}
```

```haskell
instance (Monad m) => Monad (J r m) where
    return = pure
    e >>= f = muJ . (fmap f) $ e
```

```haskell
hsequence :: Monad m => [[x] -> m x] -> m [x]
hsequence [] = return []
hsequence (xm:xms) = do
    x <- xm []
    xs <- hsequence [ \ys -> ym(x:ys) | ym <- xms ]
    return (x : xs)
```

Tic-Tac-Toe

Tic-Tac-Toe

If both players play optimally, the game ends in a **draw**

Standard backward induction then says that **any first move is an optimal move**

But what if the opponent is likely to make **mistakes**, surely some first moves are better than others

You want to choose a first move that **maximises the chances of your opponent making a mistake**

This can be easily calculated with the **selection monad transformer**

$$J_R^{\Delta} X = (X \to \Delta R) \to \Delta X$$

Demo

Haskell

```
-- TIC-TAC-TOE GAME

-- The board and moves are
--     012        (-1, 1)  (0, 1)  (1, 1)
--     345   or   (-1, 0)  (0, 0)  (1, 0)
--     678        (-1,-1)  (0,-1)  (1,-1)
-- R is the set 3 = {-1,0,1}
```

```
-- Rational maximising player
pR :: [Move] -> J R D Move
pR xs = J (\p -> uniform $ argmax ([0..8] `minus` xs) (probs . p))

-- Rational minimising opponent
oR :: [Move] -> J R D Move
oR xs = J (\p -> uniform $ argmin ([0..8] `minus` xs) (probs . p))

-- Irrational opponent
oI :: [Move] -> J R D Move
oI xs = J (\p -> uniform ([0..8] `minus` xs))
```

```
players :: [[Move] -> J R D Move]
players = [pR, oI, pR, oI, pR, oI, pR, oI, pR]
```

Thank you for your attention!