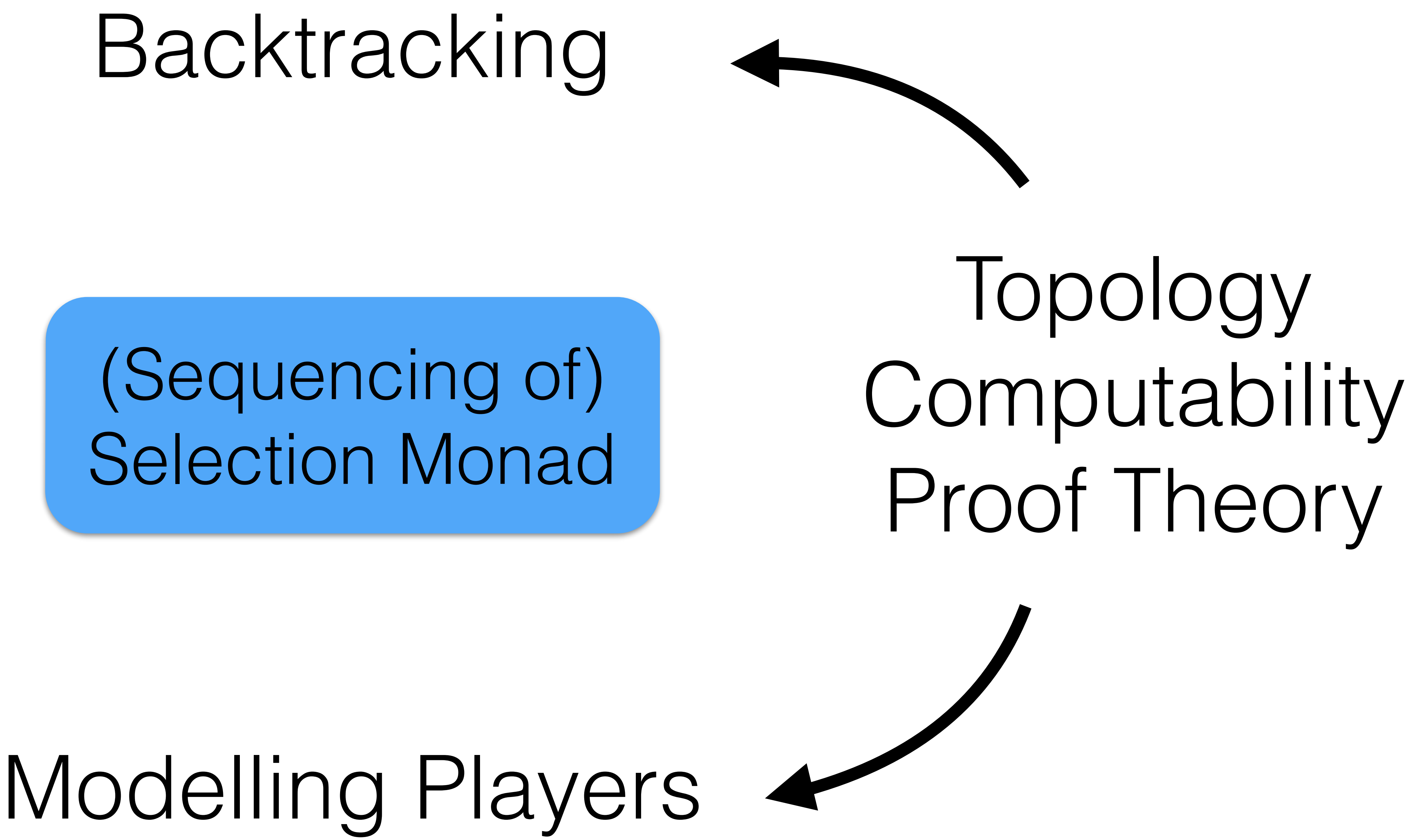
A Monad for Backtracking

(*Backward Induction and Unbounded Games*)

Paulo Oliva
Queen Mary University of London

BCTCS 2015
14 September, London


Backtracking
(Sequencing of)
Selection Monad
Topology
Computability
Proof Theory
Modelling Players

A Puzzle

A Puzzle

Using the numbers 1,2,…,10 fill in the empty cells below so that each row and column has the same sum

	X	X	X
	X	X	X

A Puzzle

Using the numbers 1,2,…,10 fill in the empty cells below so that each row and column has the same sum

1	X	X	X
2	**5**	**7**	**8**
9	**3**	**4**	**6**
10	X	X	X

Searching for a Solution…

Order the cells:

0	X	X	X
1	2	3	4
5	6	7	8
9	X	X	X

Generate all arrays $[x_0,\ldots,x_9]$, with x_i in $\{1,\ldots,10\}$

Until we find a “good” one

C Implementation

```c
int xs[10];

for (xs[0]=1; xs[0]<=10; xs[0]++)
  for (xs[1]=1; xs[1]<=10; xs[1]++)
    fo
                    if (good(xs))
                      { print(xs); return 0; }
```

```c
int good(int *xs) {
  int test1 = distinct(xs);
  int sum1 = xs[0] + xs[1] + xs[5] + xs[9];
  int sum2 = xs[1] + xs[2] + xs[3] + xs[4];
  int sum3 = xs[5] + xs[6] + xs[7] + xs[8];
  int test2 = (sum1 == sum2) && (sum2 == sum3);
  return test1 && test2;
}
```

Haskell Implementation

```haskell
good :: [Int] -> Bool
good
    w
```

```haskell
e :: (Int -> Bool) -> Int
e p = if sol == Nothing then 0 else fromJust sol
  where sol = find p [1..10]

es :: [J Bool Int]
es = map (\i -> J e) [1..10]

super :: J Bool [Int]
super = sequence es

play :: [Int]
play = selection super good
```

Haskell 20x faster than C

xterm

```
*Main> play
Chomsky{oliva}: gcc example1.c -o example1-c
Chomsky{oliva}: time ./example1-c
1
2 5 7 8
9 3 4 6
10

real    0m22.740s
user    0m22.676s
sys     0m0.059s
Chomsky{oliva}:
Chomsky{oliva}:
Chomsky{oliva}: ghc example1.hs -o example1-haskell
Chomsky{oliva}: time ./example1-haskell
1
2 5 7 8
9 3 4 6
10

real    0m1.222s
user    0m1.205s
sys     0m0.015s
Chomsky{oliva}:
```

(Magically Efficient) Backtracking
=
Sequencing Selection Monad

A Game

Purple player starts, Green players continues

0	X	X	X
1	2	3	4
5	6	7	8
9	X	X	X

Green wins if a solution is achieved

Purple wins otherwise

Selection Monad

Monads

DEFINITION 1.2 (Strong monad). *Let T be a meta-level unary operation on simple types, that we will call a* type operator. *A type operator T is called a* strong monad *if we have a family of closed terms*

$$\begin{aligned} \eta_X &: X \to TX \\ (\cdot)^\dagger &: (X \to TY) \to (TX \to TY) \end{aligned}$$

satisfying the laws

(*i*) $(\eta_X)^\dagger = \mathrm{id}_{TX}$

(*ii*) $g^\dagger \circ \eta_Y = g$

(*iii*) $(g^\dagger \circ f)^\dagger = g^\dagger \circ f^\dagger$

where $g\colon Y \to TR$ *and* $f\colon X \to TY$.

Selection Monad

- Fix R. The type mapping

$$J\,X = (X \longrightarrow R) \longrightarrow X$$

is a **strong monad**

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

monJ :: J r x -> (x -> J r y) -> J r y
monJ e f = J (\p -> b p (a p))
   where
      a p = selection e $ (\x -> p (b p x))
      b p x = selection (f x) p

instance Monad (J r) where
   return x = J(\p -> x)
   e >>= f = monJ e f
```

Product of Selection Functions

- Strong monads support two operations

$$(T\ X) \times (T\ Y) \longrightarrow T\ (X \times Y)$$

- So we have two “products” of type

$$(J\ X) \times (J\ Y) \longrightarrow J\ (X \times Y)$$

- **Game theoretic interpretation**:
  A way of combining players’ strategies!

Sequencing…

sequence :: Monad m => [m a] -> m [a]

base Prelude, base Control.Monad

Evaluate each action in the sequence from left to right, and collect the results.

- One product $(J\ X) \times (J\ Y) \longrightarrow J\ (X \times Y)$ can be iterated

sequence :: $\Pi_i\ J\ X_i \longrightarrow J\ \Pi_i\ X_i$

Interlude…

Topology

- **Theorem**[Tychonoff].
  Countable product of compact sets is compact

- **Searchable set** = set + selection function

$$(X \longrightarrow \text{Bool}) \longrightarrow X$$

- **Searchable sets** ~ compact sets

- **Theorem**[Escardo].
  Countable product of searchable sets is searchable

  Proof. *Sequencing of selection monad*

Logic

- T = Gödel's calculus of primitive recursive functionals
- Bar recursion BR: Spector (1962) computational interpretation of countable choice
- Interpretation of classical analysis into T + BR
- Theorem[Escardó/O.'2014] BR is T-equivalent to (bounded) sequencing of selection monad

Player

=

Local Strategy

=

Selection Monad

Beauty Contest

- Two contestants {A, B}
- Three judges $\{J_1, J_2, J_3\}$

- Judge J_1 prefers A > B
- Judge J_2 prefers B > A
- Judge J_3 wants to vote for the winner

Player Context

- If judges 1 and 2 fix their moves, say A and B, that defines a **context** for judge 3
- If judge 3 chooses A then A wins
- If judge 3 chooses B then B wins
- Context = a function from moves to outcomes

Player Context

- Assume a player is choosing moves in X having in mind an outcome in R
- This player's contexts are functions $f : X \longrightarrow R$
- When all other opponents have fixed their moves, this defines a context for the player
- **Note**: In a particular game, for particular opponents, some contexts might not arise

Player Context

J1 J2 \ J3	A	B
AA	A	A
AB	A	B
BA	A	B
BB	B	B

- In this game there are **three** possible contexts for judge 3 (which are they?)

Player

- Assume players are choosing moves in X having in mind an outcome in R

- Players will be modelled as mappings from **contexts** to **good moves**

$$(X \longrightarrow R) \longrightarrow P(X)$$

- Slogan: *To know a player is to know his optimal moves in any possible context*

Our Three Judges

- $X = R = \{A, B\}$. Let $A < B$
- Judge 1 is $\text{argmin} : (X \longrightarrow R) \longrightarrow P(X)$
- Judge 2 is $\text{argmax} : (X \longrightarrow R) \longrightarrow P(X)$
- Judge 3 is $\text{fix} : (X \longrightarrow R) \longrightarrow P(X)$

$$\text{fix}(p) = \{\ x : p(x) = x\ \}$$

```haskell
type Player r x = (x -> r) -> [x]
data Cand = A | B deriving (Eq,Ord,Enum,Show)
type Judge x = Player Cand x

cand = enumFrom A  -- List of candidates [A, B,..]

-- Judge that prefer A > B
argmax1 :: Judge Cand
argmax1 p = [ x | x <- cand, p x == minimum (map p cand) ]

-- Judge that prefer B > A
argmax2 :: Judge Cand
argmax2 p = [ x | x <- cand, p x == maximum (map p cand) ]

-- Judge that wants to vote for the winner
fix :: Judge Cand
fix p = [ x | x <- cand, p x == x ]
```

Implementing in Haskell

Summary

- Selection monad models “local backtracking” and modelling of players
- Sequencing of selection monad gives
  - Efficient backtracking
  - Implementation of backward induction
  - Computational interpretation of countable choice
  - Computational version of Tychonoff’s theorem

References

- Escardó and Oliva. *Selection functions, bar recursion and backward induction*. Mathematical Structures in Computer Science, 20(2):127-168, 2010

- Escardó and Oliva. *Sequential games and optimal strategies*. Proceedings of the Royal Society A, 467:1519-1545, 2011

- Hedges, Oliva, Sprits, Zahn, and Winschel. *A higher-order framework for decision problems and games*, ArXiv, http://arxiv.org/abs/1409.7411, 2014