A Monad for Backtracking

(Backward Induction and Unbounded Games)

Paulo Oliva
Queen Mary University of London

BCTCS 2015
14 September, London
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.

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
A Puzzle

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Searching for a Solution...

Order the cells:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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]++)
                                        if (good(xs))
                                            { print(xs); return 0; } 

int good(int *xs) {
    int test1 = distinct(xs);
    int test2 = (sum1 == sum2) && (sum2 == sum3);
    return test1 && test2;
}
```

Haskell Implementation

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

where
good xs = test1 && test2
  where
test1 = distinct [1..10] xs

sum1 = (xs !! 1) + (xs !! 2) + (xs !! 3) + (xs !! 4)

sum2 = (xs !! 5) + (xs !! 6) + (xs !! 7) + (xs !! 8)

sum3 = (xs !! 0) + (xs !! 1) + (xs !! 5) + (xs !! 9)

test2 = (sum1 == sum2) && (sum2 == sum3)
```

```haskell
Haskell Implementation

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

```
*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

<table>
<thead>
<tr>
<th>0</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ \eta_X : X \to TX \]

\[ (\cdot)^\dagger : (X \to TY) \to (TX \to TY) \]

satisfying the laws

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

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

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

where $g : Y \to TR$ and $f : X \to TY$. 
Selection Monad

- Fix $R$. The type mapping

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

is a **strong monad**

```haskell
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
  \[(T X) \times (T Y) \rightarrow T (X \times Y)\]

- So we have two “products” of type
  \[(J X) \times (J Y) \rightarrow J (X \times Y)\]

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

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

\[
\text{sequence} :: \quad \Pi_i J X_i \rightarrow J \Pi_i X_i
\]
Interlude...
Topology

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

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

  \[(X \rightarrow \text{Bool}) \rightarrow X\]

- **Searchable sets** \sim compact sets

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

  **Proof.** *Sequencing of selection monad*
Logic

- $T = \text{Gödel's calculus of primitive recursive functionals}$

- Bar recursion $\text{BR}$: Spector (1962) computational interpretation of countable choice

- Interpretation of classical analysis into $T + \text{BR}$

- Theorem[Escardó/O.’2014] $\text{BR}$ 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 $f : X \rightarrow R$

• 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

<table>
<thead>
<tr>
<th>J1 J2 \ J3</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>AB</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>BA</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>BB</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

- 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

\[(X \rightarrow R) \rightarrow P(X)\]

• 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 $\text{argmin} : (X \rightarrow R) \rightarrow P(X)$

- Judge 2 is $\text{argmax} : (X \rightarrow R) \rightarrow P(X)$

- Judge 3 is $\text{fix} : (X \rightarrow R) \rightarrow P(X)$

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

