What Sequential Games, the Tychonoff Theorem and the Double-Negation Shift have in Common

Martín Escardó
University of Birmingham, UK

Joint work with Paulo Oliva, Queen Mary, London, UK.

phd open, Warsaw University, 23-25 May 2013

Abstract

I will discuss a higher-type functional, written here in Haskell, which

(1) calculates optimal strategies for sequential games,

(2) implements a computational version of the Tychonoff Theorem from topology,

(3) realizes the Double-Negation Shift from logic and proof theory.

Abstract

This functional makes sense for both finite and infinite lists.

The binary case amounts to an operation that is available in any monad, specialized to a certain selection monad.

Once this monad is defined, this functional turns out to be already available in the Haskell Standard Prelude, called `sequence`

An amazingly versatile functional

```
bigotimes :: [(x -> r) -> x] -> ([x] -> r) -> [x]
bigotimes [] p = []
bigotimes (e : es) p = x0 : bigotimes es (p.(x0:))
  where x0 = e(\x -> p(x : bigotimes es (p.(x:))))
```

1. `r` is a type of generalized truth values.
2. $(\mathtt{x} \to \mathtt{r}) \to \mathtt{x}$ is a type of selection functions.
3. The input is a list of selection functions for `x`.
4. The output is a single, combined selection function for $[\mathtt{x}]$.

Products of selection functions

If the input list of selection functions is

$$\varepsilon = [\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_n, \ldots],$$

then the mathematical notation for the output of the algorithm is

$$\bigotimes_i \varepsilon_i.$$

The definition of the algorithm amounts to a recursive definition,

$$\bigotimes_i \varepsilon_i = \varepsilon_0 \otimes \bigotimes_i \varepsilon_{i+1},$$

for a suitable binary operation $\otimes$ that combines two selection functions.

What I intend to do in this lecture

1. Explain the main ideas behind selection functions and the mathematical operation $\bigotimes$.

2. Implement some sample applications in functional programming:

   1. playing Tic-Tac-Toe,

   2. solving the n-Queens puzzle, and

   3. deciding equality of functions (!) on certain infinite types.

Organization

2. Selection functions.
3. Products of selection functions.
4. Playing games.
5. The Tychonoff Theorem.
6. Monads.
7. The Double-Negation Shift.

Selection functions

If we define

```
type J r x = (x -> r) -> x
```

then the type specification of the function `bigotimes` can be rewritten as

```
bigotimes :: [J r x] -> J r [x]
```

The type `J r` turns out to have the structure of a (strong) monad.

We'll explore this towards the end of the lecture.

Quantifiers

To understand the type constructor `J`, we also consider

```
type K r x = (x -> r) -> r
```

Continuation monad, related to `call/cc` and CPS translation, classical logic.

We regard it as a type of generalized quantifiers with truth values `r`.

With `r` = `Bool`, two elements of the type `K r x` are the existential and universal quantifiers.

Monad morphism from `J r` to `K r`

We'll use this morphism before we give `J r` the structure of a monad:

```
overline :: J r x -> K r x
overline e = \p -> p(e p)
```

The Haskell notation

`overline e`

corresponds to the mathematical notation

$$\overline{\varepsilon}.$$

This operation transforms selection functions into quantifiers.

Terminology

Terminology	Mathematics	Haskell	Type
predicate	p, q	`p, q`	`x -> r`
selection function	ε, δ	`e, d`	`J r x = ((x -> r) -> x)`
quantifier	ϕ, γ	`phi, gamma`	`K r x = ((x -> r) -> r)`

Selection functions for sets

A selection function for a set finds an element for which a given predicate holds.

$$\begin{cases} S \subseteq X \\ p\colon X \to \texttt{Bool} \\ \varepsilon(p) \in S \end{cases}$$

1. We require our selection functions to be total.
2. We select an arbitrary element of S if none satisfies p.
3. This forces S to be non-empty.

Haskell code

```
find :: [x] -> J Bool x
find []     p = undefined
find [x]    p = x
find (x:xs) p = if p x then x else find xs p

forsome, forevery :: [x] -> K Bool x
forsome  = overline.find
forevery xs p = not(forsome xs (not.p))
```

Or, expanding the definitions,

```
forsome xs p = p(find xs p)
```

Hilbert's ε-calculus

Our definition of the existential quantifier is the same as in Hilbert's ε-calculus:

$$\exists x\ p(x) \iff p(\varepsilon(p)).$$

The definition of `forevery` uses the De Morgan Law for quantifiers,

$$\forall x\ p(x) \iff \neg\exists x\ \neg p(x).$$

E.g.

```
find    [1..100] (\x -> odd x && x > 17) = 19
forsome [1..100] (\x -> odd x && x > 17) = True

find    [1..100] (\x -> odd x && even x) = 100
forsome [1..100] (\x -> odd x && even x) = False
```

As already discussed, we are interested in finite non-empty lists only, to make sure the produced selection function is total.

Summary so far

A selection function ε for a set S has to satisfy:

1. $\varepsilon(p) \in S$, whether or not there actually is some $x \in S$ such that $p(x)$ holds.
2. If $p(x)$ holds for some $x \in S$, then it holds for $x = \varepsilon(p)$.

The first condition forces the set S to be non-empty.

Haskell code

We shall need this later:

```
findBool :: J Bool Bool
findBool p = p True
```

This is equivalent to

```
findBool p = if p True then True else False
```

Selection functions for quantifiers

If $\phi(p)$ stands for $\exists x \in S\ p(x)$, then Hilbert's condition can be written as

$$\phi(p) = p(\varepsilon(p)),$$

or equivalently,

$$\phi = \overline{\varepsilon}.$$

If this equation holds, we say that ε is a selection function for the quantifier ϕ.

Thus, a selection function for a set S is the same thing as a selection function for the existential quantifier of S.

Example: selection function for a universal quantifier

When $\phi(p)$ is the universal quantifier $\forall x \in S \;\; p(x)$ of the set S, this amounts to

1. $\varepsilon(p) \in S$.

2. If $p(x)$ holds for $x = \varepsilon(p)$, then it holds for all $x \in S$.

Example: selection function for a universal quantifier

When $\phi(p)$ is the universal quantifier $\forall x \in S\ \ p(x)$ of the set S, this amounts to

1. $\varepsilon(p) \in S$.

2. If $p(x)$ holds for $x = \varepsilon(p)$, then it holds for all $x \in S$.

This is known as the Drinker Paradox.

In every pub there is a person a such that if a drinks then everybody drinks.

Here S is the set of people in the pub, and $p(x)$ means that x drinks, and we calculate a with the selection function as $a = \varepsilon(p)$.

Haskell code

A selection function for the universal quantifier of a finite set:

```
findnot :: [x] -> J Bool x
findnot []     p = undefined
findnot [x]    p = x
findnot (x:xs) p = if p x then findnot xs p else x
```

This satisfies

```
findnot xs p = find xs (not.p)
```

and the function `forevery` defined earlier satisfies

```
forevery = overline.findnot
```

Another useful instance of `r`

Consider “predicates” that give numbers rather than boolean truth values.

E.g., the predicate may assign prices to goods.

1. Find the price of the most expensive good.

   This is done by a quantifier, called $\sup$.

   $(\mathtt{Good} \to \mathtt{Price}) \to \mathtt{Price}$.

2. Find the most expensive good.

   This is done by a selection function, called argsup.

   $(\mathtt{Good} \to \mathtt{Price}) \to \mathtt{Good}$.

Maximum-Value Theorem

Any continuous function $p\colon [0,1] \to \mathbb{R}$ attains its maximum value.

This means that there is $a \in [0,1]$ such that

$$\sup p = p(a).$$

However, the proof is non-constructive.

A maximizing argument a cannot be algorithmically calculated from p.

Mean-Value Theorem

There is $a \in [0, 1]$ such that

$$\int p = p(a).$$

Again this a cannot be found from p using an algorithm.

Drinker Paradox

Can be written as

$$\forall(p) = p(a).$$

Finding elements in sets corresponds to

$$\exists(p) = p(a).$$

We can find a as $a = \varepsilon(p)$ if the ranges of the universal and existential quantifiers are finite.

And we'll go beyond the finite case below.

General situation

With ϕ among $\exists, \forall, \sup, \inf, \int, \ldots$,

$$\phi(p) = p(a).$$

In favourable circumstances a can be calculated as

$$a = \varepsilon(p).$$

Haskell code

```
argsup, arginf :: [x] -> J Int x
argsup [] p = undefined
argsup [x] p = x
argsup (x:y:zs) p = if p x < p y
                    then argsup (y:zs) p
                    else argsup (x:zs) p
```

These will be useful for two-player games in which there can be a draw.

Binary product of quantifiers and selection functions

In every pub there are a man a_0 and a woman a_1 such that if a_0 buys a drink for a_1 then every man buys a drink for some woman.

Binary product of quantifiers and selection functions

In every pub there are a man a_0 and a woman a_1 such that if a_0 buys a drink for a_1 then every man buys a drink for some woman.

If $X_0 =$ set of men and $X_1 =$ set of women, and if we define $\phi = \forall \otimes \exists$ by

$$\phi(p) = (\forall x_0 \in X_0\ \exists x_1 \in X_1\ p(x_0, x_1)),$$

then this amounts to saying that

$$\phi(p) = p(a)$$

for a suitable pair $a = (a_0, a_1) \in X_0 \times X_1$,

Our objective is to calculate such a pair.

Binary product of quantifiers and selection functions

We have selection functions for $\forall$ and $\exists$, say ε_0 and ε_1.

We need a selection function $\varepsilon = \varepsilon_0 \otimes \varepsilon_1$ for the quantifier $\phi = \forall \otimes \exists$.

Binary product of quantifiers

It is easy to define the product $\otimes$ of quantifiers, generalizing from the previous example, where $\phi = \phi_0 \otimes \phi_1$ for $\phi_0 = \forall$ and $\phi_1 = \exists$:

$$(\phi_0 \otimes \phi_1)(p) = \phi_0(\lambda x_0.\phi_1(\lambda x_1.p(x_0, x_1))).$$

Binary products of selection functions

The definition of the product of selection functions is a bit subtler:

$$\begin{aligned}(\varepsilon_0 \otimes \varepsilon_1)(p) \quad &= \quad (a_0, a_1) \\ \text{where} \quad & a_0 = \varepsilon_0(\lambda x_0.\overline{\varepsilon_1}(\lambda x_1.p(x_0.x_1))) \\ & a_1 = \varepsilon_1(\lambda x_1.p(a_0, x_1)).\end{aligned}$$

We need to check that

$$\overline{\varepsilon_0} \otimes \overline{\varepsilon_1} = \overline{\varepsilon_0 \otimes \varepsilon_1},$$

which amounts to

if $\phi_0 = \overline{\varepsilon_1}$ and $\phi_1 = \overline{\varepsilon_1}$, then $\phi_0 \otimes \phi_1 = \overline{\varepsilon_0 \otimes \varepsilon_1}$.

Back to our motivating example

The required man and woman can be calculated with the formula

$$(a_0, a_1) = (\varepsilon_0 \otimes \varepsilon_1)(p),$$

where ε_0 and ε_1 are selection functions for the quantifiers $\forall$ and $\exists$ respectively.

Iterated product

Given a sequence of sets $X_0, X_1, \ldots, X_n, \ldots$, write

$$\prod_{i<n} X_i = X_0 \times \cdots \times X_{n-1}.$$

We define $\bigotimes\colon \prod_{i<n} JRX_i \to JR \prod_{i<n} X_i$, by induction on n:

$$\bigotimes_{i<0} \varepsilon_i = \lambda p.(), \qquad \bigotimes_{i<n+1} \varepsilon_i = \varepsilon_0 \otimes \bigotimes_{i<n} \varepsilon_{i+1}.$$

Informally, and assuming right associativity of $\otimes$,

$$\bigotimes_{i<n} \varepsilon_i = \varepsilon_0 \otimes \varepsilon_1 \otimes \cdots \otimes \varepsilon_{n-1}$$

Main property of products of selection functions

Products of quantifiers can be defined in the same way, and

$$\bigotimes_{i<n} \overline{\varepsilon_i} = \overline{\bigotimes_{i<n} \varepsilon_i}.$$

The products are of quantifiers in the left-hand side and of selection functions in the right-hand side.

Haskell code

```
otimes :: J r x -> J r [x] -> J r [x]
otimes e0 e1 p = a0:a1
 where a0 = e0(\x0 -> overline e1(\x1 -> p(x0:x1)))
       a1 = e1(\x1 -> p(a0:x1))

bigotimes :: [J r x] -> J r [x]
bigotimes [] = \p -> []
bigotimes (e:es) = e ‘otimes‘ bigotimes es
```

Although we have motivated this algorithm by considering finite lists, it does make sense for infinite lists too, as we shall see in due course.

Playing games

1. Products of selection functions compute optimal plays and strategies.

2. Generalize products in order to account for history dependent games.

3. Give concise implementations of Tic-Tac-Toe and n-Queens as illustrations.

First example

Alternating, two-person game that finishes after exactly n moves.

1. Eloise plays first, against Abelard. One of them wins (no draw).

2. The i-th move is an element of the set X_i.

3. The game is defined by a predicate $p\colon \prod_{i<n} X_i \to \texttt{Bool}$
   that tells whether Eloise wins wins a given play $x = (x_0, \ldots, x_{n-1})$.

4. Eloise has a winning strategy for the game p if and only if

$$\exists x_0 \in X_0 \forall x_1 \in X_1 \exists x_2 \in X_2 \forall x_3 \in X_3 \cdots p(x_0, \ldots, x_{n-1}).$$

First example

4. Eloise has a winning strategy for the game p if and only if

$$\exists x_0 \in X_0 \forall x_1 \in X_1 \exists x_2 \in X_2 \forall x_3 \in X_3 \cdots p(x_0, \ldots, x_{n-1}).$$

If we define

$$\phi_i = \begin{cases} \exists_{X_i} & \text{if } i \text{ is even,} \\ \forall_{X_i} & \text{if } i \text{ is odd,} \end{cases}$$

then this condition for Eloise having a winning strategy can be equivalently expressed as

$$\left(\bigotimes_{i<n} \phi_i \right) (p).$$

Calculating the optimal outcome of a game

More generally, the value

$$\left(\bigotimes_{i<n} \phi_i\right)(p)$$

gives the optimal outcome of the game.

This takes place when all players play as best as they can.

In the first example, the optimal outcome is `True` if Eloise has a winning strategy, and `False` if Abelard has a winning strategy.

Calculating an optimal play

Suppose each quantifier ϕ_i has a selection function ε_i (thought of as a policy function for the i-th move).

Theorem. The sequence

$$a = (a_0, \ldots, a_{n-1}) = \left(\bigotimes_{i<n} \varepsilon_i\right)(p)$$

is an optimal play.

This means that for every stage $i < n$ of the game, the move a_i is optimal given that the moves $a_0, \ldots, a_{i-1}$ have been played.

Finding an optimal strategy

Theorem. The function $f_k : \prod_{i<k} X_i \to X_k$ defined by

$$f_k(a) = \left(\left(\bigotimes_{i=k}^{n-1} \varepsilon_i \right) (\lambda x . p(a \mathbin{++} x)) \right)_0$$

is an optimal strategy for playing the game.

This means that given that the sequence of moves $a_0, \ldots, a_{k-1}$ have been played, the move $a_k = f_k(a_0, \ldots, a_{k-1})$ is optimal.

Second example

Choose $R = \{-1, 0, 1\}$ instead, with the convention that

$$\begin{cases} -1 = \text{Abelard wins}, \\ 0 = \text{draw}, \\ 1 = \text{Eloise wins}. \end{cases}$$

The existential and universal quantifiers get replaced by $\sup$ and $\inf$:

$$\phi_i = \begin{cases} \sup_{X_i} & \text{if } i \text{ is even}, \\ \inf_{X_i} & \text{if } i \text{ is odd}. \end{cases}$$

The optimal outcome is still calculated as $\bigotimes_{i<n} \phi_i$, which amounts to

$$\sup_{x_0 \in X_0} \inf_{x_1 \in X_1} \sup_{x_2 \in X_2} \inf_{x_3 \in X_3} \cdots p(x_0, \ldots, x_{n-1}).$$

Second example

The optimal outcome is

$$\begin{cases} -1 = \text{Abelard has a winning strategy,} \\ 0 = \text{the game is a draw,} \\ 1 = \text{Eloise has a winning strategy.} \end{cases}$$

Can compute optimal outcomes, plays and strategies with the same formulas.

History-dependent games

Allowed moves in the next round depend on the moves played so far.

The idea is that the nth selection function depends on the first n played moves.

It selects moves among the allowed ones, which depend on the played ones.

Haskell code

```
otimes :: J r x -> (x -> J r [x]) -> J r [x]
otimes e0 e1 p = a0 : a1
 where a0 = e0(\x0->overline(e1 x0)(\x1->p(x0:x1)))
       a1 = e1 a0 (\x1 -> p(a0:x1))
```

```
bigotimes :: [[x] -> J r x] -> J r [x]
bigotimes [] = \p -> []
bigotimes (e:es) =
 e[] `otimes` (\x->bigotimes[\xs->d(x:xs) | d<-es])
```

Implementation of games

We need to define a type `R` of outcomes, a type `Move` of moves, a predicate

```
p :: [Move] -> R
```

that gives the outcome of a play, and (history-dependent) selection functions for each stage of the game:

```
epsilons :: [[Move] -> J R Move]
```

Implementation of games

Each different game needs a different definition

```
(R, Move, p, epsilons)
```

Implementation of optimal plays

But all games are played in the same way, using the previous theorems:

```
optimalPlay :: [Move]
optimalPlay = bigotimes epsilons p

optimalOutcome :: R
optimalOutcome = p optimalPlay

optimalStrategy :: [Move] -> Move
optimalStrategy as = head(bigotimes epsilons' p')
  where epsilons' = drop (length as) epsilons
        p' xs = p(as ++ xs)
```

Tic-Tac-Toe

```
data Player = X | O
```

The outcomes are $R = \{-1, 0, 1\}$.

```
type R = Int
```

The possible moves are

0	1	2
3	4	5
6	7	8

```
type Move = Int

type Board = ([Move], [Move])
```

Tic-Tac-Toe

```
wins :: [Move] -> Bool
wins =
 someContained [[0,1,2],[3,4,5],[6,7,8],
                [0,3,6],[1,4,7],[2,5,8],
                [0,4,8],[2,4,6]]

value :: Board -> R
value (x,o) | wins x     =  1
            | wins o     = -1
            | otherwise  =  0
```

Tic-Tac-Toe

```
outcome :: Player -> [Move] -> Board -> Board

outcome whoever [] board = board

outcome X (m : ms) (x, o) =
 if wins o then (x, o)
 else outcome O ms (insert m x, o)

outcome O (m : ms) (x, o) =
 if wins x then (x, o)
 else outcome X ms (x, insert m o)
```

We assume that player `X` starts, as usual:

```
p :: [Move] -> R
p ms = value(outcome X ms ([],[]))
```

Tic-Tac-Toe

```
epsilons :: [[Move] -> J R Move]
epsilons = take 9 epsilons
 where epsilons = epsilonX : epsilonO : epsilons
       epsilonX h = argsup ([0..8] `setMinus` h)
       epsilonO h = arginf ([0..8] `setMinus` h)
```

Let’s run this

```
main :: IO ()
main =  putStr
("An optimal play for Tic-Tac-Toe is " ++ show optimalPlay ++
 "\nand the optimal outcome is " ++ show optimalOutcome ++ "\n")
```

Compile and run:

```
$ ghc --make -O2 TicTacToe.hs
$ time ./TicTacToe
An optimal play for Tic-Tac-Toe is [0,4,1,2,6,3,5,7,8]
and the optimal outcome is 0

real 0m1.721s    user 0m1.716s   sys 0m0.004s
```

(Under the operating system Ubuntu/Debian 9.10 in a 2.13GHz machine.)

In pictures

X		

X		
	O	

X	X	
	O	

X	X	O
	O	

X	X	O
	O	
X		

X	X	O
O	O	
X		

X	X	O
O	O	X
X		

X	X	O
O	O	X
X	O	

X	X	O
O	O	X
X	O	X

n-Queens

We adopt the following conventions:

1. A solution is a permutation of $[0..(n-1)]$, which tells where each queen should be placed in each row.

1. A move is an element of $[0..(n-1)]$, saying in which column of the given row (=stage of the game) the queen should be placed.

n-Queens

```
n = 8
type R = Bool
type Coordinate = Int
type Move = Coordinate
type Position = (Coordinate,Coordinate)

attacks :: Position -> Position -> Bool
attacks (x, y) (a, b) =
  x == a  ||  y == b  || abs(x - a) == abs(y - b)

valid :: [Position] -> Bool
valid [] = True
valid (u : vs) =
  not(any (\v -> attacks u v) vs) && valid vs
```

n-Queens

```
p :: [Move] -> R
p ms = valid(zip ms [0..(n-1)])

epsilons :: [[Move] -> J R Move]
epsilons = replicate n epsilon
 where epsilon h = find ([0..(n-1)] `setMinus` h)
```

n-Queens

```haskell
main :: IO ()
main = putStr
 ("An optimal play for " ++ show n ++  "-Queens is " ++
                                               show optimalPlay ++
  "\nand the optimal outcome is " ++ show optimalOutcome ++ "\n")
```

Compile and run:

```
$ ghci --make -O2 NQueens.hs
$ time ./NQueens
```

and we get:

```
An optimal play for 8-Queens is [0,4,7,5,2,6,1,3]
and the optimal outcome is True

real 0m0.011s   user 0m0.012s    sys 0m0.000s
```

12-Queens

Get $[0, 2, 4, 7, 9, 11, 5, 10, 1, 6, 8, 3]$ computed in five seconds.

The Tychonoff Theorem

1. Close connection of some computational and topological ideas.

2. With applications to computation.

3. No background on topology required.

The language spoken in Topology Land

Space.

Continuous function.

Compact space.

Countably based space.

Hausdorff space.

Cantor space.

Baire space.

Guided tour to Topology Land

Call a set

1. searchable if it has a computable selection function, and
2. exhaustible if it has a computable boolean-valued quantifier.

We showed that

> Exhaustible and searchable sets are compact.

Related to the well-known fact that

> Computable functionals are continuous.
>
> Finite parts of the output depend only on finite parts of the input.

A widely quoted topological slogan

> Infinite compact sets behave, in many interesting and useful ways, as if they were finite.

This matches computational intuition:

> The ability to exhaustively search an infinite set, algorithmically and in finite time, is indeed a computational sense in which the set behaves as if it were finite.

It may seem surprising at first sight that there are such sets, but this was known in the 1950's or before.

Topological properties

Compact sets of total elements form countably based Hausdorff spaces.

Importing results from topology to computation

We had previously explored what happens if one looks at theorems in topology and applies the previous dictionary.

Exhaustible and searchable sets are compact.

Computable functions are continuous.

Searchable sets of total elements form countably based Hausdorff spaces.

Some results in topology

1. Finite sets are compact, and hence for example the booleans are compact.

2. Arbitrary products of compact sets are compact (Tychonoff Theorem).

   Hence the space of infinite sequences of booleans is compact.

   This the *Cantor space*.

2. Continuous images of compact sets are compact.

3. Any non-empty, countably based, compact Hausdorff space is a continuous image of the Cantor space.

Applying the dictionary, we get

1. Finite sets are searchable, and hence for example the booleans are searchable.
2. Finite and countably infinite products of searchable sets are searchable.

   Hence the Cantor space is searchable.
3. Computable images of searchable/exhaustible sets are searchable/exhaustible.
4. Any non-empty exhaustible set is a computable image of the Cantor space.

A souvenir from our excursion

Non-empty exhaustible sets are searchable.

This is computationally interesting:

If we have a search procedure that answers yes/no (given by a quantifier),

we get a procedure that gives witnesses (given by a selection function).

Haskell code

The Tychonoff Theorem is implemented by the history-free version `bigotimes` of the product of selection functions.

Hence a selection function for the Cantor space is given by

```
findCantor :: J Bool [Bool]
findCantor = bigotimes (repeat findBool)
```

Haskell code

Computable images of searchable sets are searchable:

```
image :: (x -> y) -> J r x -> J r y
image f e = \q -> f(e(\x -> q(f x)))
```

Application: deciding equality of functionals

Perhaps contradicting common wisdom, we write a total (!) functional that decides whether or not two given total functionals equivalent:

```
equal :: Eq z => ((Integer -> Bool) -> z)
              -> ((Integer -> Bool) -> z) -> Bool

equal f g = foreveryFunction(\u->f u == g u)
```

This doesn't contradict computability theory.

Haskell code

Code functions `Integers->Bool` as lazy lists by storing non-negative and negative arguments at even and odd indices:

```
code :: (Integer -> Bool) -> [Bool]
code f = [f(reindex i)| i<-[0..]]
  where reindex i | even i     =      i `div` 2
                  | otherwise = -((i+1)`div` 2)
```

Haskell code

But actually we are interested in the opposite direction:

```
decode :: [Bool] -> (Integer -> Bool)
decode xs i | i >= 0    =  xs `at`   (i * 2)
            | otherwise =  xs `at` ((-i * 2) - 1)

at :: [x] -> Integer -> x
at (x:xs) 0 = x
at (x:xs) (n+1) = at xs n
```

Haskell code

```
findFunction :: J Bool (Integer -> Bool)
findFunction = image decode findCantor

forsomeFunction :: K Bool (Integer -> Bool)
forsomeFunction = overline findFunction

foreveryFunction :: K Bool (Integer -> Bool)
foreveryFunction p = not(forsomeFunction(not.p))
```

This completes the definition of the function `equal`.

Experiment

Here the function `c` is a coercion:

```
c :: Bool -> Integer
c False = 0
c True  = 1

f, g, h :: (Integer -> Bool) -> Integer
f a = c(a(7 * c(a 4) +  4 * (c(a 7)) + 4))
g a = c(a(7 * c(a 5) +  4 * (c(a 7)) + 4))
h a = if not(a 7)
      then if not(a 4) then c(a  4) else c(a 11)
      else if a 4      then c(a 15) else c(a  8)
```

Are any two of these three functions equal?

Experiment

When we run this, using the interpreter this time, we get:

```
 $ ghci Tychonoff.hs
 ...
 Ok, modules loaded: Main.
*Main> :set +s
*Main> equal f g
False
(0.02 secs, 4274912 bytes)
*Main> equal g h
False
(0.01 secs, 0 bytes)
*Main> equal f h
True
(0.00 secs, 0 bytes)
```

Experiment

Where do `f` and `g` differ?

```
*Main> take 11 (code (findFunction(\u->g u /= h u)))
[True,True,True,True,True,True,True,True,True,True,False]
(0.05 secs, 3887756 bytes)
```

We believe that these ideas open up the possibility of new, useful tools for automatic program verification and bug finding.

Monads

It turns out that the function `image :: (x -> y) -> J r x -> J r y` defined above is the functor of a monad.

The unit is

```
singleton :: x -> J r x
singleton x = \p -> x
```

The multiplication is

```
bigunion :: J r (J r x) -> J r x
bigunion e = \p -> e(\d -> overline d p) p
```

Haskell code for the monads

In the links provided.

Consequences of having a monad

The history-free version of the functional `bigotimes` is simply the Haskell standard prelude function `sequence` instantiated to the selection monad:

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])
            where mcons p q =
                   p >>= \x->q >>= \y->return (x:y)
```

Thus, our computational manifestation of the Tychonoff Theorem is already in the Haskell standard prelude, and becomes immediately available once one defines the selection monad.

The history-dependent product

Can also be defined for any monad. See the paper.

The Double-Negation Shift

$\forall n\ \neg\neg(An) \implies \neg\neg\forall n\ (An)$.

Used by Spector'62 to realize the Classical Axiom of Countable Choice.

He used the dialectica interpretation.

We use Kreisel's modified realizability.

K-shift

Generalized double negation shift.

Monad $KA = (A \to R) \to R$.

$\forall x\, K(Ax) \implies K\forall x\ (Ax)$.

Algebra $KA \to A$: proposition A with double-negation elimination.

CPS-translation = Gödel–Gentzen negative translation.

J-shift

Gives the double negation shift via the monad morphism $J \to K$.

Monad $JA = (A \to R) \to A$.

$\forall x\, J(Ax) \implies J\forall x\, (Ax)$ directly realized by $\bigotimes$.

Algebra $JA \to A$: proposition A that satisfies Peirce's Law.

Get translation based on J rather than K.

Get witnesses from classical proofs that use countable choice using $\bigotimes$.

Concluding remarks

Diverse mathematical subjects coexist harmoniously and have a natural bed in functional programming:

1. game theory (optimal strategies),
2. topology (Tychonoff Theorem),
3. category theory (monads), and
4. proof theory (double-negation shift, classical axiom of choice).

Selection functions everywhere

It is the selection monad that unifies these mathematical subjects.

Its associated product functional $\bigotimes$

1. computes optimal strategies,
2. implements a computational manifestation of the Tychonoff Theorem,
3. realizes the double-negation shift and the classical axiom of choice.