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

CORRECTED 12 AUG 2010

Martín Escardó
University of Birmingham, UK
m.escardo@cs.bham.ac.uk

Paulo Oliva
Queen Mary University of London, UK
pbo@dcs.qmul.ac.uk

Abstract

This is a tutorial for mathematically inclined functional programmers, based on previously published, peered reviewed theoretical work. We discuss a higher-type functional, written here in the functional programming language Haskell, which (1) optimally plays sequential games, (2) implements a computational version of the Tychonoff Theorem from topology, and (3) realizes the Double-Negation Shift from logic and proof theory. The functional makes sense for finite and infinite (lazy) lists, and in the binary case it amounts to an operation that is available in any (strong) monad. In fact, once we define this monad in Haskell, it turns out that this amazingly versatile functional is already available in Haskell, in the standard prelude, called `sequence`, which iterates this binary operation. Therefore Haskell proves that this functional is even more versatile than anticipated, as the function `sequence` was introduced for other purposes by the language designers, in particular the iteration of a list of monadic effects (but effects are not what we discuss here).

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: functional programming

General Terms Algorithms, languages, theory.

Keywords Functional programming, Haskell, monad, search, game theory, optimal strategy, exhaustible set, axiom of choice, infinite data, dependent type, Agda, topology, logic, foundations.

1. An amazingly versatile functional

Perhaps the most concise and self-contained definition of the functional we discuss in this tutorial is this, using Haskell notation:

```
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:))))
```

This is an example of an algorithm that is patently not self-explanatory. Here is a preliminary indication of what is going on, to be elaborated below. Think of `r` as a type of *generalized truth values*. A particular case of interest will be `r = Bool`, but not the only relevant one. The input of the algorithm is a list of so-called *selection functions* `(x -> r) -> x` for a type `x`, and the output is a single, combined selection function `([x] -> r) -> [x]` for the type `[x]` of lists of elements of the type `x`. 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

$$\bigotimes_i \varepsilon_i.$$

As we shall see, the definition of the algorithm amounts to, and can in fact be literally written as 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.

This algorithm makes sense for both finite and infinite (lazy) lists, and we shall write it in several different but equivalent ways. It is difficult to give a general specification of what it does, because it turns out that it performs a variety of seemingly unrelated tasks:

1. It optimally plays sequential games.
2. It implements a computational manifestation of the Tychonoff Theorem from topology.
3. It realizes the Double-Negation Shift (DNS) from logic and proof theory (whose role is to computationally extract witnesses from classical proofs that use the axiom of countable choice).

And more, as discussed in [6–8, 10–12]. For example, the so-called Bekic's Lemma for fixed-points that arises in domain theory and programming language semantics is a particular case of the product of selection functions [8].

Originally, the algorithm was designed to achieve (2), but, by studying it in more detail, we discovered that it also achieves (3) and (1), in chronological order. We regard this as a rather surprising course of events, given that the Tychonoff Theorem hasn't appeared so far in the theoretical study of (1) and (2), at least not explicitly. In any case, it is certainly surprising that the same mathematical operation or algorithm should simultaneously solve these seemingly unrelated tasks, each of which is fundamental in its own right.

It is the purpose of this tutorial to explain the main ideas behind the mathematical operation $\bigotimes$, defined and studied in [6–8, 10–12]. This expository paper is an invitation for the reader to explore these references, which give full mathematical details and proofs, as well as more techniques, ideas and directions.

MSFP'10, September 26, 2010, Baltimore, Maryland, USA.

We also implement a number of sample applications in functional programming, such as playing Tic-Tac-Toe (aka Noughts-and-Crosses), solving the n-Queens puzzle, and deciding equality of functions (!) on certain infinite types, among others.

Organization. 2. Selection functions. 3. Products of selection functions. 4. Playing games. 5. The Tychonoff Theorem. 6. Monads. 7. The Double-Negation Shift. 8. Concluding remarks.

2. 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]
```

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

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

The type constructor `K r` is known as the continuation monad, and is related to control operators such as `call/cc`. But we shall instead regard it as a type of generalized *quantifiers* over the type `x`, with type of generalized *truth values* `r`. As we shall see later, `J r` is a monad too, called the *selection monad*, and there is a monad morphism to `K r`. We shall have occasion to use the 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

$$\overline{\varepsilon}.$$

We refer to the elements of `J r x` as *selection functions*. What `overline` does, then, is to transform selection functions into quantifiers. In this context, we refer to the elements of the function type `(x -> r)` as *predicates*.

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

With `r = Bool`, a particular element of the type `K r x` is the existential quantifier, or the *existential quantification functional* to be more precise, which arises when we consider selection functions for sets. After considering this particular case of a selection function, we consider the more general case of selection functions for arbitrary quantifiers, which include the universal quantifier, of course, but other generalized quantifiers as well.

In this discussion we adopt the traditional terminology from logic that refers to higher-type functions as *functionals*, and we stress that in logic one usually adopts the terminology *higher-order* to logics that quantify over propositions, and the terminology *higher-type* to formal systems that allow nested function types (such as Gödel's system T and generalizations, including PCF and Haskell).

2.1 Selection functions for sets

A selection function for a set S finds an element of S for which a given predicate holds. But we emphasize that we require our selection functions to be *total*. Hence if there is no element of S satisfying the predicate, we select an arbitrary element of S, and so S must be non-empty. Here is an example, where we consider finite sets given as finite lists:

```
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)
```

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

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

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

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

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

```
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
```

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 $x = \varepsilon(p)$.

Notice that the first condition forces the set S to be non-empty. Here is another example that we shall need later:

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

This is equivalent to

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

but it is silly to check the two possible cases. This is so both conceptually and for the sake of efficiency.

2.2 Selection functions for quantifiers

Our use of selection functions goes beyond the above idea. If $\phi(p)$ stands for $\exists x \in S\ p(x)$, then Hilbert's condition can be written as an equation,

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

or equivalently, using the monad morphism, as the equation

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

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.

When $\phi(p)$ is the universal quantifier $\forall x \in S\ p(x)$ of the set S, the above equivalent equations amount 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 $a = \varepsilon(p)$. A selection function for the universal quantifier of a finite set can be defined in Haskell as follows:

```
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
```

Notice that this satisfies

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

and that the function `forevery` defined above satisfies

```
forevery = overline.findnot
```

We now consider "predicates" that give numbers rather than boolean truth values (sometimes known as *objective functions*). For example, the predicate may assign prices to goods. Given such a predicate, we may wish to

1. find the price of the most expensive good — this is done by a quantifier, called $\sup$,
2. find the most expensive good — this is done by a selection function, called argsup.

Of course, the result of the selection function is ambiguously defined, because there may be more than one good with the highest price (just as there may be more than one element satisfying a given boolean-valued predicate, and we chose one arbitrarily with the above algorithms).

This should be compared with the Maximum-Value Theorem, which says that 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, and it is known that there is no computable selection function argsup that calculates $a = \mathrm{argsup}(p)$ (but the supremum quantifier $\sup\colon ([0,1] \to \mathbb{R}) \to \mathbb{R}$ is known to be computable). This can also be compared to the Mean-Value Theorem, which says that for some $a \in [0,1]$,

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

Again this a cannot be found from p using an algorithm. Similarly the Drinker Paradox can be written

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

and finding elements in sets corresponds to

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

We have already seen that 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). The general equation we are considering here is thus

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

where in favourable circumstances a can be calculated as

$$a = \varepsilon(p)$$

for a suitable selection function ε that works for all predicates p.

Continuing the discussion about prices of goods, in the next example we artificially assume that the only two goods are `True` and `False`, and that our numbers are integers:

```
argsup :: J Int Bool
argsup p = p True > p False
```

```
sup :: K Int Bool
sup = overline argsup
```

Of course the definition of `argsup` is equivalent to

```
argsup p = if p True > p False then True else False
```

which is perhaps more natural. If we want our goods to be given as a list, we can instead define

```
argsup :: [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
```

This can be made more efficient by avoiding re-evaluations of `p`. Moreover, if the range of `p` is not the set of all integers, but just a finite range, such as $-1, 0, 1$, then the following is a more efficient algorithm, which stops when the maximum value 1 is reached, and switches to a specialized algorithm when the value 0 is reached:

```
argsup :: [x] -> J Int x
argsup     [] p = undefined
argsup (x:xs) p = f xs x (p x)
 where f    xs  a   1  = a
       f     [] a   r  = a
       f (x:xs) a (-1) = f xs x (p x)
       f    xs  a   0  = g xs
        where g [] = a
              g (x:xs) | p x == 1  = x
                       | otherwise = g xs
```

Infima can he handled by mirroring this algorithm, or alternatively by reduction to suprema, where the involution `\x -> -x` plays the role of negation in this context:

```
arginf :: [x] -> J Int x
arginf xs p = argsup xs (\x -> - p x)
```

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

3. Products of selection functions

We first consider binary products of selection functions, and then iterate them (in)finitely often, obtaining the functional discussed in Section 1.

3.1 Binary product

In order to motivate our construction, consider the following variation of the Drinker Paradox discussed above: in every pub there are a man a_0 and a woman a_1 such that if a_0 buys a drink to a_1 then every man buys a drink to some woman. If the sets of men and women are X_0 and X_1 respectively, and if we define the combined quantifier $\phi = \forall \otimes \exists$ by

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

then this can be formalized by saying that, for a suitable pair $a = (a_0, a_1) \in X_0 \times X_1$,

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

Our objective is to calculate such a pair a.

The key observation is that we have selection functions for the quantifiers $\forall$ and $\exists$, say ε_0 and ε_1, and that what we need is a suitable combined selection function $\varepsilon = \varepsilon_0 \otimes \varepsilon_1$ for the combined quantifier $\phi = \forall \otimes \exists$. We define combinations of quantifiers and of selection functions in such a way that they commute with the

above monad morphism, where the combinator is written $\otimes$ for both quantifiers and selection functions. In other words:

> If ε_i is a selection function for the quantifier ϕ_i for $i = 0, 1$, then $\varepsilon = \varepsilon_0 \otimes \varepsilon_1$ is a selection function for the quantifier $\phi = \phi_0 \otimes \phi_1$.

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

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

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

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

What we need to check is that

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

which amounts to

> if $\phi_i = \overline{\varepsilon_i}$ for $i = 0, 1$, then $\phi_0 \otimes \phi_1 = \overline{\varepsilon_0 \otimes \varepsilon_1}$.

This is a routine verification that the reader is encouraged to perform for the sake of understanding (and that is performed in the given references). Coming 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.

The binary product of selection functions can be implemented as

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

We leave the implementation of the binary product of quantifiers as an exercise (but later we shall implement the binary product for all monads, which will apply to the selection monad and the continuation monad in particular).

3.2 Iterated product

Given a sequence of sets $X_0, X_1, \ldots, X_n, \ldots$, define the product $\prod_{i<n} X_i$ by induction on n as

$$\prod_{i<0} X_i = \{()\}, \qquad \prod_{i<n+1} X_i = X_0 \times \prod_{i<n} X_{i+1}.$$

Informally,

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

For each n we define a function

$$\bigotimes : \prod_{i<n} JRX_i \to JR \prod_{i<n} X_i,$$

by induction as

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

Finite products of quantifiers can be defined in the same way, and the following equation holds, by induction:

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

Of course, the products are of quantifiers in the left-hand side and of selection functions in the right-hand side.

In dependently typed languages such as Agda, one can write a single product program that works for every n, but in Haskell a different program for each n is required. This limitation can be overcome if we restrict ourselves to the particular case in which $X_i = X$ for every i and some fixed X. Then $\prod_{i<n} X_i$ amounts to the type of finite lists of length n, and we write an algorithm that works for lists of any length.

We first need to consider the particular case of the binary product with types `x0 = x` and `x1 = [x]` for a given type `x`. Then a pair $(\mathtt{a}_0, \mathtt{a}_1) :: (\mathtt{x}_0, \mathtt{x}_1)$ can be coded as the list $(\mathtt{a}_0 : \mathtt{a}_1) :: [\mathtt{x}]$. With these choices and coding, the above binary-product program can be rewritten as

```
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))
```

This is now in a suitable form for iteration:

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

Expanding the definitions, this is the same algorithm given in Section 1. 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.

4. Playing games

We first show that products of selection functions compute optimal plays and strategies. We then generalize the product of selection functions in order to account for history dependent games, and give concise and efficient implementations of Tic-Tac-Toe and n-Queens as illustrations of the techniques.

4.1 Optimal outcomes, plays and strategies

As a first example, consider an alternating, two-person game that finishes after exactly n moves, with one of the players winning. The i-th move is an element of the set X_i and the game is defined by a predicate $p\colon \prod_{i<n} X_i \to R$ with

$$R = \mathtt{Bool},$$

that tells whether the first player, Eloise playing against Abelard, wins a given play $x = (x_0, \ldots, x_{n-1}) \in \prod_{i<n} X_i$. Then 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

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

More generally, this value gives the *optimal outcome* of the game, which takes place when all players play as best as they can. In this example, the optimal outcome is `True` if Eloise has a winning strategy, and `False` if Abelard has a winning strategy.

The following is proved in the paper [8], which also gives formal definitions of the game theoretic terminology. Suppose each quantifier ϕ_i has a selection function ε_i (thought of as a policy function for the i-th move).

1. 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.

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

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

is an optimal strategy for playing the game.

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

Here $k < n-1$, and $x \in \prod_{i=k}^{n-1} X_i$, and $a +\!\!+ x$ is the concatenation of the sequences $a = (a_0, \ldots, a_{k-1})$ and $x = (x_k, \ldots, x_{n-1})$, and the subscript 0 picks the first element of the list calculated inside the brackets.

As a second example, we choose

$$R = \{-1, 0, 1\}$$

instead, with the convention that -1 means that Abelard wins, 0 means that the game is a draw, and 1 that Eloise wins. Because Eloise and Abelard want to maximize and minimize the outcome of the game respectively, we replace the existential and universal quantifiers sup and inf respectively,

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

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

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

and is 1 if Eloise has a winning strategy, -1 if Abelard has a winning strategy, and 0 otherwise. Moreover, the above formulas for computing optimal outcomes, plays and strategies apply.

4.2 History dependent games

In most sequential games of interest and that occur in practice, the set of allowed moves at a given stage depends on the moves played at the previous stages. The simplest case is that of a two-move game. We assume that we have sets of moves X_0 and X_1. Once a move $x_0 \in X_0$ has been played, the allowed moves form a set $S_{x_0} \subseteq X_1$ that depends on x_0. We want to account for situations such as

$$\forall x_0 \in X_0 \exists x_1 \in S_{x_0}\ p(x_0, x_1).$$

For example, in every pub there are a man a_0 and a woman a_1 older than a_0 such that if a_0 buys a drink to a_1, then every man buys a drink to an older woman. Here S_{x_0} is the set of women older than x_0. This can be formalized by considering two quantifiers, the second of which has a parameter:

$$\phi_0 \in KRX_0, \qquad \phi_1 \colon X_0 \to KRX_1.$$

In our running example,

$$\begin{aligned} \phi_0(q) &= \forall x_0 \in X_0\, q(x_0), \\ \phi_1(x_0)(q) &= \exists x_1 \in S_{x_0}\ q(x_1). \end{aligned}$$

Their history-dependent product is defined as the history-free product considered before, taking care of instantiating the parameter appropriately:

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

Similarly, given a selection function and a family of selection functions,

$$\varepsilon_0 \in JRX_0, \qquad \varepsilon_1 \colon X_0 \to JRX_1,$$

we define their history-dependent product

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

where it is understood that $\overline{\varepsilon_1}(x_0) = \overline{\varepsilon_1(x_0)}$. And then again we have

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

This amounts to saying that if ε_0 is a selection function for the quantifier ϕ_0, and if $\varepsilon_1(x_0)$ is a selection function for the quantifier $\phi_1(x_0)$ for every $x_0 \in X_0$, then $\varepsilon_0 \otimes \varepsilon_1$ is a selection function for the quantifier $\phi_0 \otimes \phi_1$.

We can iterate this if we are given a sequence of history dependent selection functions

$$\varepsilon_n \colon \prod_{i<n} X_i \to JRX_n.$$

We do this by induction, using the binary history dependent product in the induction step:

$$\bigotimes_{i<0} \varepsilon_i = \lambda p.(),$$

$$\bigotimes_{i<n+1} \varepsilon_i = \varepsilon_0() \otimes \lambda x_0. \bigotimes_{i<n} (\lambda(x_1, \ldots, x_i).\varepsilon_{i+1}(x_0, \ldots, x_i)).$$

This can be written in Haskell as follows, if we trivialize the dependent products as above:

```
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])
```

4.3 Implementation of games and optimal strategies

To define a sequential game, 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]
```

Once this is done, we can compute optimal plays, optimal outcomes, and optimal strategies with the mathematical formulas given above:

```
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)
```

Notice that all players use the same strategy function. To be precise, because the notion of player is not part of our definition of sequential game, the strategy says what move should be played at stage k given the moves at stages $i < k$. These three definitions apply to both history-free and history-dependent games and products of selection functions.

4.4 Finite games of unbounded length

So far we have discussed finite products, and hence games, of fixed length. In order to account for finite games of unbounded length, we use infinite lazy lists, and we assume that the predicate `p` scans the list of moves until the game ends and produces an outcome. Of course, for this we need that every sequence of moves does eventually lead to the end of the game in a finite number of moves. The only explicitly given infinite lazy list in the specification of a game is the list `epsilons` of policy functions for each stage of the game.

4.5 Tic-Tac-Toe

We now consider Tic-Tac-Toe as an illustration of the above techniques, but we emphasize that the ideas are very general and apply to a wide variety of sequential games. The type of players is

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

The outcomes are 1 (first player wins), -1 (second player wins) and 0 (draw). We represent them by integers:

```
type R = Int
```

The possible moves are

0	1	2
3	4	5
6	7	8

So again we represent moves by integers:

```
type Move = Int
```

Although the concept of a game board doesn't feature explicitly in our specification of a sequential game, it is convenient to introduce it in our implementation:

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

The components of the pair are the sets of moves played by `X` and `O` respectively. We represent these sets as sorted lists without repetitions for efficiency. A set of moves wins if it contains a row, a column or a diagonal, and we use this to define the value of a board:

```
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
```

Here the function

```
someContained :: Ord x => [[x]] -> [x] -> Bool
```

satisfies `someContained xss ys = True` if and only if some list `xs` in the list `xss` has all elements occurring in the list `ys`. This is a standard programming exercise, but we include the definition later for the sake of completeness.

Next we define a function `outcome` such that, given a player, a play (that is, a list of moves), and a board, produces the board that results after playing the moves in an alternating fashion. If at some point one player wins, then the remaining moves are ignored and the board is unchanged:

```
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)
```

Then the predicate that defines the game is obtained by computing the outcome of the play starting from the empty board. We assume that player `X` starts, as usual:

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

Finally, we need to the define the selection functions, or policy functions, for each stage of the game. As discussed above, because the player `X` wants to maximize the outcome of the game, and `O` wants to minimize it, we use selection functions for the supremum and infimum quantifiers, defined above. These selection functions are history-dependent, where the history `h` is the sequence of moves played so far, and they choose moves among those that haven't been played yet:

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

The function

```
setMinus :: Ord x => [x] -> [x] -> [x]
```

returns a list with all elements that are in the first list but not in the second one. And that's it, apart from the standard definition of set-theoretical operations, which are routine, but are given in Section 4.7 below for the sake of completeness.

To test the program, we add

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

Compiling this with the Glasgow Haskell compiler as

```
$ ghc --make -O2 TicTacToe.hs
```

and running

```
$ time ./TicTacToe
```

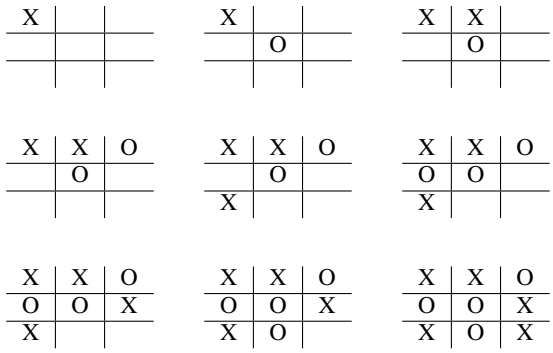
under the operating system Ubuntu/Debian 9.10 in a 2.13GHz machine, we get:

```
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
```

This means that, as is well known, if the two players play as best as they can, the game is a draw. There are many optimal plays.

The computed one depends on which selection functions we have chosen for our supremum and infimum quantifiers. In pictures, the computed optimal play is:



4.6 n-Queens

We can solve the n-Queens puzzle using the same ideas. This time this is a 1-player game, but again the number of players is not explicit in the formalization of the problem and its solution. 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.
2. 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.

Let's say we consider the standard chess board:

```
n = 8
```

The following should be self-explanatory given the above development and assuming familiarity with the formulation of the n-Queens problem:

```
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

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)
```

And that's it. We test this as above:

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

We then 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
```

The fact that the optimal outcome is `True` means that the optimal play is indeed a solution of the 8-Queens problem. With $n = 12$ the solution $[0, 2, 4, 7, 9, 11, 5, 10, 1, 6, 8, 3]$ is computed in five seconds. It should be apparent that many other standard games and search problems can be concisely expressed and solved in this formalism using the same pattern.

4.7 Appendix

For the sake of completeness, so that the reported experiments can be reproduced by the readers, with similar run-time results in similar machines, we include the routine code for finite sets represented as sorted lists without repetitions used above:

```
contained :: Ord x => [x] -> [x] -> Bool
contained [] ys = True
contained xs [] = False
contained (us@(x : xs)) (y : ys)
    | x == y    = contained xs ys
    | x >= y    = contained us ys
    | otherwise = False

someContained :: Ord x => [[x]] -> [x] -> Bool
someContained [] ys = False
someContained xss [] = False
someContained (xs : xss) ys
  = contained xs ys || someContained xss ys

insert :: Ord x => x -> [x] -> [x]
insert x [] = [x]
insert x (vs@(y : ys))
    | x == y       = vs
    | x <  y       = x : vs
    | otherwise    = y : insert x ys

delete :: Ord x => x -> [x] -> [x]
delete x [] = []
delete x (vs@(y : ys))
    | x == y    = ys
    | x <  y    = vs
    | otherwise = y : delete x ys

setMinus :: Ord x => [x] -> [x] -> [x]
setMinus xs [] = xs
setMinus xs (y : ys) = setMinus (delete y xs) ys
```

5. The Tychonoff Theorem

This section is about the close connection of some computational and topological ideas, with applications to computation. For the purposes of this exposition, it is not required to know what the topological terms *space*, *compact*, *continuous*, *countably based*, and *Hausdorff* mean. Rather, the readers should regard topology as a *foreign language*, and use the facts stated below as a *dictionary* between computational and topological notions. Computer-Science Land readers interested in learning the language can start from the introductory text [17], but this is not necessary in order to get around in our guided tour to Topology Land.

5.1 An excursion 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.

It was shown in [7] that

Exhaustible sets are topologically compact.

Here we are implicitly assuming that we are dealing only with sets of *total* elements. The situation for sets that include partial or undefined elements is subtler and we refer the reader to [7]. By the development of Section 2, using the monad morphism, we know that searchable sets are exhaustible, and hence searchable sets are compact too. The above quoted fact is related to the well-known fact that

Computable functionals are topologically continuous.

This is so even from a purely operational point of view [4], without considering any denotational semantics such as Scott domains. A widely quoted topological slogan is that

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

This matches computational intuition, because 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 [13, 14], and we shall see some examples shortly. We also need to mention that in our computational setting:

Compact sets of total elements form countably based Hausdorff spaces.

Here we assume that any two equivalent total elements are identified (for example, the strict and non-strict constant function `Int -> Int` are considered to be the same total element).

The papers [5, 7] explore what happens if one looks at theorems in topology and applies this dictionary. Even theorems that are at the core of topology and analysis turn out to be computationally relevant:

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 is known as the *Cantor space*, as it is topologically isomorphic (or homeomorphic) to the famous Cantor Third-Middle set in the real line.
3. Continuous images of compact sets are compact.
4. 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.

This is proved in [7], which develops more examples. Notice that we have replaced *compact* sometimes by *searchable* and sometimes by *exhaustible* (and sometimes by both). We did this deliberately, in order to get the following souvenir from our topological excursion:

Non-empty exhaustible sets are searchable.

This works as follows: if the set K is non-empty and exhaustible, then it is a computable image of the Cantor space by (4), and because the Cantor space is searchable by (1) and (2), the set K is searchable by (3). This is computationally interesting, because it says that if we have a search procedure that answers yes/no (given by a quantifier), then we can automatically get a procedure that gives witnesses (given by a selection function). Moreover, the selection function can be obtained by a higher-type functional, which can be written in Haskell with the following type:

```
selectionFromQuantifier :: K Bool x -> J Bool x
```

The definition and the argument that it works are rather complicated and use non-trivial recursion-theoretic and topological technology, and interested readers are referred to [7].

We now give some indications of how the above works, and develop some examples. The finite and countably infinite 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)
```

We shall run an example in a moment. We now implement the fact that 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)))
```

This works as follows. Suppose that `f :: x -> y` is given and that `e :: J r x` is a selection function for a set $S \subseteq$ `x`, and let $f(S) = \{f(s) \mid s \in S\} \subseteq$ `y` be the image of S. Given a predicate `q :: y -> r`, we wish to find `s` $\in S$ such that `q(s) = True`. The given algorithm first finds `x` such that `q(f x) = True`, using the selection function `e`, and then applies `f` to it, giving the desired result.

5.2 Deciding equality of functionals

As an application, perhaps contradicting common wisdom, we write a total (!) functional that decides whether or not two given total functionals `(Integer -> Bool) -> z` are equivalent, where `z` is any type with given decidable equality:

```
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, which correctly asserts that, for example, equality of functions `Integer -> Bool` is *not* decidable. A grammar that singles out infinitely many types that have decidable equality is provided in [7].

To complete this definition, we first faithfully code functions `Integers->Bool` as lazy lists by storing non-negative and negative arguments at even and odd indices respectively:

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

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
```

Because the space of total infinite sequences `[Bool]` is searchable by the computational Tychonoff Theorem, and because its image under the computable function `decode` is the set of total functions `Integer -> Bool`, this set of total functions is searchable and hence exhaustible:

```
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 all ingredients needed to define the function `equal`. Here is an experiment, where 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? 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)
```

(We have no clue why the last computation is reported to take zero bytes!) These examples are somewhat trivial, although probably puzzling. A non-trivial application of quantification over the Cantor space is given in [16]. This develops an algorithm for numerical integration, where real numbers are represented as infinite sequences of digits. Hence there are no round-off errors in the produced results, which can be computed to arbitrary precision with guaranteed accuracy. Coming back to our trivial example, 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.

5.3 Uniform continuity and the fan functional

Readers unfamiliar with topology and its relation to computation are likely to be puzzled regarding the above decision procedure for equality of functions, and more generally universal quantification over infinite spaces. What is going on? Here is an indication.

Functions `f :: [Bool] -> z` are continuous. If `z` is topologically discrete (has equality), this means that for every input `a`, there is an index n depending on `a`, such that the answer doesn't depend on positions of `a` with indices $i \geq n$. This is intuitively clear from a computational point of view, and can be rigorously proved (see e.g. [4]). Now, the Cantor space is compact, and there are theorems in topology that say that, in many cases, continuous functions defined on compact spaces are *uniformly continuous*. In our case, this means that there is a *single* index n, independent of the input, such that the function `f :: [Bool] -> z` doesn't look at indices $i \geq n$ in order to produce the output. Another way of saying this is that if two inputs `a` and `b` of the function `f` agree in the first n positions, then they produce the same output. This criterion can be coded in Haskell, and hence such an n can be computed. The functional that computes it is called the *fan functional*, and is known since the 1950's or even earlier [14]:

```
fan :: Eq z => ([Bool] -> z) -> Int
fan f = least(\n -> forevery(\a -> forevery(\b ->
           agree n a b --> (f a == f b))))

least :: (Int -> Bool) -> Int
least p = if p 0 then 0 else 1+least(\n -> p(n+1))

forsome, forevery :: K Bool [Bool]
forsome = overline findCantor
forevery p = not(forsome(not.p))

agree :: Int -> [Bool] -> [Bool] -> Bool
agree n a b = take n a == take n b

(-->) :: Bool -> Bool -> Bool
p --> q = not p || q
```

Here is a trivial example:

```
*Main> fan(\a -> a !! 5  &&  a !! 6)
7
```

However, we are not interested in computing the fan functional. What is relevant is that once we know the modulus of uniform continuity n, it is enough to inspect 2^n cases to figure out the complete behaviour of the function.

If there is any magic in the above algorithm for quantifying over the Cantor space and hence deciding equality of functions defined on the Cantor space, it amounts to the facts that (1) this n is not explicitly calculated by the quantification procedure (although it can be calculated with the fan functional that *uses* this procedure), *and* (2) very often, a small portion of the 2^n cases actually need to be checked in the quantification procedure (this has to do with lazy evaluation). If we move from `[Bool]` to `[Integer]`, then compactness and uniform continuity fail, and moreover equality of functions `(Integer -> Integer) -> z` is no longer decidable.

6. Monads

It turns out that the function `image` defined above is the functor of a monad [8]. The unit is

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

It implements the fact that singletons are searchable. The multiplication is

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

It implements the fact that the union of a searchable set of searchable sets is searchable (but it actually satisfies a more general specification involving generalized quantifiers [8]). To see that the multiplication behaves as claimed, assume that the selection function `e` selects over a collection of selection functions `d` for sets $S_{\mathtt{d}}$. Using the selection function `e`, we find a selection function `d` such that `p(d p)` holds, i.e. `p s` holds for some $\mathtt{s} \in S_{\mathtt{d}}$. Then we apply this selection function to `p`, to actually find such `s`.

6.1 The continuation and selection monads

We now write the monads using Haskell's conventions, where the above gets a bit cumbersome and probably harder to understand, as Haskell requires tags for monads, and hence we need functions for extracting tags (namely `quantifier` and `selection`). We first define the continuation monad:

```
module K (K(K), quantifier, unitK, functorK, muK) where

newtype K r x = K {quantifier :: (x -> r) -> r}

unitK :: x -> K r x
unitK x = K(\p -> p x)

functorK :: (x -> y) -> K r x -> K r y
functorK f phi = K(\q -> quantifier phi(\x -> q(f x)))

muK :: K r (K r x) -> K r x
muK phi = K(\p -> quantifier phi
           (\gamma -> quantifier gamma p))

instance Monad (K r) where
  return = unitK
  phi >>= f = muK(functorK f phi)
```

Using this, we define the selection monad:

```
module J (J(J), selection, unitJ, functorJ, muJ,
         module K, morphismJK) where

import K
newtype J r x = J {selection :: (x -> r) -> x}

morphismJK :: J r x -> K r x
morphismJK e = K(\p -> p(selection e p))

unitJ :: x -> J r x
unitJ x = J(\p -> x)

functorJ :: (x -> y) -> J r x -> J r y
functorJ f e = J(\q->f(selection e(\x->q(f x))))

muJ :: J r(J r x) -> J r x
muJ e = J(\p -> selection(selection e
         (\d -> quantifier(morphismJK d) p)) p)

instance Monad (J r) where
  return = unitJ
  e >>= f = muJ(functorJ f e)
```

A proof that the monad laws hold is given in [8]. There is also a short proof written in the dependently typed programming language/proof checker Agda [3] available at [9].

6.2 Consequences of having a monad

First of all, now 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.

Here the function `mcons` is simply one of the versions of our function `otimes` generalized from the continuation and selection monads to any monad, and can be equivalently written as

```
mcons :: Monad m => m x -> m[x] -> m[x]
xm `mcons` xsm =
    do x <- xm
       xs <- xsm
       return (x:xs)
```

Moreover, `sequence` itself can be equivalently written as

```
sequence :: Monad m => [m x] -> m[x]
sequence [] = return []
sequence (xm : xms) =
    do x <- xm
       xs <- sequence xms
       return(x : xs)
```

The history-dependent version of `bigotimes` also generalizes to any monad:

```
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)
```

but this time this is not in the standard prelude.

Another consequence of having a monad is the topic of Section 7, for which the remarks of Section 6.3 are relevant.

6.3 The function (h)sequence in monads other than J r

We observe that `sequence` and `hsequence`, although defined for all monads, usually don't produce convergent computations in monads other than `J r` when supplied with infinite lists. Moreover, the type `r` has to have decidable equality for infinite products of selection functions to be total (topologically, it has to be discrete) [8]. The termination proof is non-trivial and relies on the so-called *bar induction* principle. In particular, infinite products of quantifiers *cannot* be computed with `sequence` or $\bigotimes$, as also shown in [8], with a continuity argument.

One can easily understand this kind of phenomenon with the list monad. In the finite case, `sequence` computes cartesian products in the lexicographic order:

```
Prelude> sequence [[0,1],[0,1],[0,1]]
[[0,0,0],[0,0,1],[0,1,0],[0,1,1],
 [1,0,0],[1,0,1],[1,1,0],[1,1,1]]
```

But now the elements of the countable cartesian product $\{0,1\}^\omega$ cannot be arranged in an infinite list, by Cantor's diagonal argument, and when we attempt to list them using `sequence`, we get a divergent computation, which in practice aborts for lack of memory:

```
Prelude> sequence (repeat [0,1])
*** Exception: stack overflow
```

It is reassuring to see Haskell refusing to give an answer to an impossible question.

Apart from the selection monad, a monad for which we know that `sequence` and `hsequence` converge for infinite lists is the identity monad:

```
newtype Id x = Id { di :: x } deriving (Show)

instance Monad Id where
    return a     = Id a
    (Id x) >>= f = f x
```

Here `di` removes the tag `Id`. Bearing in mind that semantically `Id` is the identity, `sequence` essentially does nothing:

```
*Main> sequence [Id 1,Id 2,Id 3]
Id [1,2,3]
```

However, `hsequence` is much more interesting, as it amounts to course-of-values recursion. The type of `hsequence` in this case reduces to `[[x] -> Id x] -> Id[x]`, and, according to the development of the previous sections, the nth function in the input list $[f_0, f_1, f_2, \dots]$ is intended to have n arguments. What `hsequence` computes, then, is the sequence $\mathtt{Id}[x_0, x_1, x_2, \dots]$ defined by course-of-values induction as

$$\mathtt{Id}x_n = f_n[x_0, \dots, x_{n-1}].$$

As an example, we enrich the literature with yet another way of computing the Fibonacci sequence:

```
fibonacci :: [Integer]
fibonacci = di(hsequence (repeat f))
  where f [ ] = Id 1
        f [_] = Id 1
        f xs  = Id((xs !! (i - 1)) + (xs !! i))
          where i = length xs - 1
```

Notice that the definition of f is *not* recursive. As discussed above, the recursion is performed by `hsequence`. Here is what we get when we run it:

```
*Main> take 10 fibonacci
[1,1,2,3,5,8,13,21,34,55]
```

Now, the identity functional and primitive recursion functional are realizers of the intuitionistic axioms of choice and of dependent choice respectively. It turns out that when the monad is `J r` rather than `Id`, the functionals `sequence` and `hsequence` are instead realizers of the *classical* axioms of choice and of (slightly generalized) dependent choice, which brings us to the subject of the next section.

7. The Double-Negation Shift

This last section of the tutorial is a rather brief excursion to Proof-Theory Land with many gaps. We look at the J and K monads from the propositions-as-types and proofs-as-programs point of view, for which the dependently-typed functional language Agda [3] is more appropriate than Haskell. We revert to the notation defined before the previous section, in order to avoid the distracting tags that arise in the definitions of the monads given in Section 6.1.

7.1 The Gödel–Gentzen negative translation and the continuation monad

It is well known that $K_R A = ((A \to R) \to R)$ can be seen as a generalized double-negation operator, reducing to standard double negation when $R = \bot$ (the proposition *absurdity*, or the empty type). In order to avoid notational clutter, we write $KA = K_R A$, relying on the reader's ability to infer the subscript R from the context. The Gödel–Gentzen's *negative-translation* prefixes double negations in front of atomic propositions, disjunctions and existential quantifiers, leaving implications, conjunctions and universal quantifiers unchanged. A more general translation prefixes K instead. Given a formula A, we denote its translation by A^K. The reason for considering this translation is that given any *classical* proof of A one can algorithmically find an *intuitionistic* proof of its translation A^K.

From the point of view of proofs-as-programs, this algorithm amounts to one of the possible forms of the well-known *continuation passing style* translation. If one works through the technical details, one sees that what makes everything work is that fact that K is a monad. In particular, algebras of the monad, which are propositions A satisfying

$$KA \to A,$$

arise: every translated formula can be shown to be an algebra. In the particular case $R = \bot$, this amounts to saying that A satisfies the double-negation elimination rule $\neg\neg A \to A$, which is a classical principle. Thus, the translation forces this principle to hold intuitionistically.

7.2 The Peirce translation, the selection monad and call/cc

One can routinely repeat the above development with any monad. When this is done for $JA = J_R A = ((A \to R) \to A)$, algebras

$$JA \to A$$

are propositions that satisfy Peirce's Law

$$((A \to R) \to A) \to A.$$

Usually the continuation monad is invoked to explain call/cc, but a more natural explanation is obtained in terms of the selection monad.

7.3 Extracting programs from proofs

Let $A = \forall x \exists y\, p(x, y)$ be a formula in Heyting arithmetic with finite types (HA^ω), where p is decidable. Given an intuitionistic proof of A one can find a program t (in Gödel system T, which can be considered as a downgraded version of Haskell) such that $p(x, tx)$ holds for every x. One can think of this as follows: (1) p is the specification of the input-output relation of a program to be written down, (2) the given proof of A, because it is intuitionistic, implicitly carries such a program, (3) there is a procedure that exhibits the program given the proof, (4) hence rather than writing a program, one can show in intuitionistic logic that for every input x there is an output y satisfying the specification, and get a program automatically, which by construction satisfies the specification.

Of course, this would be impractical for the average programmer. But there are many mathematicians and logicians in computer science departments who are doing just that, using various kinds of mechanical proof assistants. An added bonus is that there is a large body of literature with intuitionistic proofs available, and hence potentially a large body of programs that don't need to be explicitly written down. The downside is that *formalizing* rigorous proofs is known to be no easy task. But nevertheless, if this activity is not practical at the moment, it is certainly very exciting, scientifically deep, enlightening, and mathematically pleasing.

But what if the proof of A is classical, rather than intuitionistic, that is in Peano arithmetic with finite types (PA^ω)? Never mind. Using the negative translation, one can still extract a program. However, when one climbs up the logical systems, one gets into difficulties. For example, there is no problem in using the axioms of choice or dependent choice in HA^ω, because they are realizable (one can write programs in system T that implement them). But in PA^ω, the situation is subtler. What one needs is to realize the negative translations of the axioms, which is problematic because they involve existential quantification, which is altered by the translation.

7.4 The axiom of choice and the double-negation shift

Let T be any of the monads J or K. The T-translation of the axiom of choice is

$$\forall x\ T\exists y\ A(x,y) \implies T\exists f\forall x\ A(x,f(x)).$$

The axiom of choice is the case in which T is the identity monad. To extract programs from classical proofs in PA^ω using the axiom of choice, one needs to realize the K-translation of the axiom of choice, which is often referred to as the *classical axiom of choice*. Spector's idea [18] was to instead realize the T-shift (with $T = K$ and $R = \bot$),

$$\forall x\ TB(x) \implies T\forall x\ B(x).$$

and prove that the intuitionistic axiom of choice together with this gives the classical axiom of choice. This is enough to be able to extract programs from proofs in PA^ω extended with choice. Spector did this for the case where x ranges over natural numbers, realizing the *axiom of countable choice*. Although Spector worked with the dialectica interpretation and an extension of system T with so-called bar recursion, his ideas remain relevant and crisp. See also [1, 2].

It turns out that the history-free product of selection functions directly realizes the T-shift for $T = J$, and that the J-shift implies the K-shift when B is in the image of the K-translation [11]. In fact, the iterated product of selection functions provides a alternative and intuitive formulation of bar recursion. Formal proofs/programs written in Agda are available at [9]. It is worth mentioning that the J-shift is not system-T definable, and that to define it in Agda one has to disable the termination checker (and then users have to trust us).

7.5 The axiom of dependent choice

The previous subsection states that the history-free product of selection functions realizes the J-shift, which in turn gives the double-negation shift, and the J- and K-translations of the axiom of choice. It turns out that the history-dependent product of selection functions realizes a version of the axiom of dependent choice. We don't have an implementation of this in Agda yet, but we are working on this and related things.

7.6 Where are the (proofs and) programs for this section?

Proofs/programs, written in Agda [3], can be found at [9]. No previous knowledge of Agda is necessary: the types of the programs read like usual logical expressions in ordinary mathematical notation, and their proofs look like usual functional programs. There is also no need to run them, because their types tell us what their behaviour will be. However, we plan to apply them to give an alternative implementation of the program extractions from classical proofs using dependent choice developed by Monika Seisenberger in the system Minlog [15].

8. Concluding remarks

We have shown that diverse mathematical subjects coexist harmoniously and have a natural bed in functional programming: game theory (optimal strategies), topology (Tychonoff Theorem), category theory (monads), and proof theory (double-negation shift, classical axiom of (dependent) choice).

An alternative title to this paper could have been *selection functions everywhere*. It is the selection monad that unifies these mathematical subjects, where its associated product functional $\otimes$ computes optimal strategies, implements a computational manifestation of the Tychonoff Theorem, and realizes the double-negation shift and the classical axiom of (dependent) choice.

Full mathematical proofs of the claims made in this tutorial can be found in our joint papers given in the references. We wish to point out that several of these proofs use mathematical machinery outside the scope of this tutorial, but, as exemplified here, the claims can be understood and applied with minimal mathematical background if we accept them without proof.

Certainly more references are needed in a tutorial such as this: please consult the references given in our self-references. We have been unable to include them for lack of space.

References

[1] S. Berardi, M. Bezem, and T. Coquand. On the computational content of the axiom of choice. *The Journal of Symbolic Logic*, 63(2):600–622, 1998.

[2] U. Berger and P. Oliva. Modified bar recursion and classical dependent choice. *Lecture Notes in Logic*, 20:89–107, 2005.

[3] A. Bove and P. Dybjer. Dependent types at work. *Proceedings of Language Engineering and Rigorous Software Development, LNCS*, 5520:57–99, 2009.

[4] M. Escardó and W. Ho. Operational domain theory and topology of sequential programming languages. *Information and Computation*, 207:411437, 2009.

[5] M. H. Escardó. Synthetic topology of data types and classical spaces. *Electron. Notes Theor. Comput. Sci.*, 87:21–156, 2004.

[6] M. H. Escardó. Infinite sets that admit fast exhaustive search. In *Proceedings of LICS*, pages 443–452, 2007.

[7] M. H. Escardó. Exhaustible sets in higher-type computation. *Logical Methods in Computer Science*, 4(3), 2008.

[8] M. H. Escardó and P. Oliva. Selection functions, bar recursion, and backward induction. *Mathematical Structures in Computer Science*, 20(2):127–168, 2010.

[9] M. H. Escardó and P. Oliva. Companion programs in Haskell and Agda for the present publication. `http://www.cs.bham.ac.uk/~mhe/papers/msfp2010/`, July 2010.

[10] M. H. Escardó and P. Oliva. Computational interpretations of analysis via products of selection functions. In F. Ferreira, B. Lowe, E. Mayordomo, and L. M. Gomes, editors, *Computability in Europe 2010, LNCS*, pages 141–150. Springer, 2010.

[11] M. H. Escardó and P. Oliva. The Peirce translation and the double negation shift. In F. Ferreira, B. Löwe, E. Mayordomo, and L. M. Gomes, editors, *Programs, Proofs, Processes - CiE 2010, LNCS 6158*, pages 151–161. Springer, 2010.

[12] M. H. Escardó and P. Oliva. Searchable sets, Dubuc–Penon compactness, omniscience principles, and the Drinker Paradox. In F. Ferreira, H. Guerra, E. Mayordomo, and J. Rasga, editors, *Computability in Europe 2010, Abstract and Handout Booklet*, pages 168–177. Centre fo Applied Mathematics and Information Technology, Department of Mathematics, University of Azores, 2010.

[13] D. Normann. *Recursion on the Countable Functionals*, volume 811 of *Lecture Notes in Mathematics*. Springer, Berlin, 1980.

[14] D. Normann. Computing with functionals—computability theory or computer science? *Bull. Symbolic Logic*, 12(1):43–59, 2006.

[15] M. Seisenberger. Programs from proofs using classical dependent choice. *Annals of Pure and Applied Logic*, 153(1–3):97–110, 2008.

[16] A. Simpson. Lazy functional algorithms for exact real functionals. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 456–464, 1998.

[17] M. Smyth. Topology. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1 of *Oxford science publications*, pages 641–761. 1992.

[18] C. Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In F. D. E. Dekker, editor, *Recursive Function Theory: Proc. Symposia in Pure Mathematics*, volume 5, pages 1–27. American Mathematical Society, Providence, Rhode Island, 1962.