Monad Transformers for Backtracking Search

Jules Hedges
Queen Mary University of London
j.hedges@qmul.ac.uk

This paper extends Escardó and Oliva's selection monad to the selection monad transformer, a general monadic framework for expressing backtracking search algorithms in Haskell. The use of the closely related continuation monad transformer for similar purposes is also discussed, including an implementation of a DPLL-like SAT solver with no explicit recursion. Continuing a line of work exploring connections between selection functions and game theory, we use the selection monad transformer with the nondeterminism monad to obtain an intuitive notion of backward induction for a certain class of nondeterministic games.

1 Introduction

Selection functions are higher-order functions related to continuations, introduced by Martin Escardó and Paulo Oliva in (Escardó and Oliva 2010a), with many remarkable properties. There are many intuitions that can be used to understand selection functions, including viewing them as

- a refinement of continuations that carry additional information
- a generalised form of search algorithm
- a generalised notion of rationality

All of these intuitions will be used in this paper. Selection functions form a monad, called the *selection monad*, and the monoidal product of this monad, called the *product of selection functions*, can be used for several seemingly unrelated purposes:

1. In proof theory, it gives a computational meaning to the negative translation of the axiom of countable choice (Escardó and Oliva 2010a)
2. In synthetic topology, it is a computational form of Tychonoff's theorem (Escardó 2008)
3. In functional programming, it provides a monadic framework for backtracking search problems (Escardó and Oliva 2010b)
4. In game theory, it is a generalisation of the backward induction algorithm (Escardó and Oliva 2012)

In this paper we are concerned with points (3) and (4).

The intuition behind (3) is that a selection function is a generalised search algorithm, inputting a 'generalised predicate' characterising objects to be found, and outputting an object that satisfies the predicate if one exists. The purpose of the product of selection functions is to combine search algorithms for simple search spaces into search algorithms for more complex search spaces. This has been used to derive so-called *seemingly impossible functional programs* which search certain infinite (but topologically compact) data types in finite time (Escardó 2007).

For (4) we have a powerful intuition that a selection function is a generalised form of preference for rational agents in game theory (Hedges 2013). Indeed the $\arg\max$ operator, which characterises the

To appear in EPTCS.

behaviour of classical economic agents, is one of the canonical examples of a selection function. The product of selection functions applied to copies of arg max is precisely the backward induction algorithm, used to compute subgame-perfect Nash equilibria of sequential games (Escardó and Oliva 2012), and we can transfer this game-theoretic intuition to other instances of the product of selection functions. An introduction to selection functions written from this point of view is (Escardó and Oliva 2011).

Part 2 of this paper is an introduction to selection functions and the Haskell implementation of the continuation and selection monads. In part 3 we show the use of the continuation monad, rather than the selection monad, for writing search algorithms in Haskell, and the use of the continuation monad transformer to express more advanced search algorithms such as DPLL. In part 4 we define the selection monad transformer and show its relationship to the ordinary selection monad and the continuation monad transformer. Part 5 defines nondeterministic sequential games, and in part 6 we use the monoidal product of the selection monad transformer, applied to the nondeterminism monad, to give an intuitive notion of backward induction for these games.

This paper uses two different notations, namely Haskell and 'naive type theory', which can be read either as ordinary set theory or as functional pseudocode. To make it easy to distinguish the notations, Haskell code is set in a $\boxed{\text{box}}$. There are three appendices listing Haskell code verbatim: appendix A contains the DPLL implementation (part 3), B contains the selection monad transformer library (part 4) and C contains an example nondeterministic game (parts 5 and 6). All of this code can be downloaded from the author's homepage[1].

2 Continuations and selection functions

A selection function is defined to be any function with a type of the form

$$\varepsilon : \mathscr{J}_R X$$

where the selection monad $\mathscr{J}_R$ is defined as

$$\mathscr{J}_R X = (X \to R) \to X$$

or, in Haskell notation,

```
data Sel r x = Sel {runSel :: (x → r) → x}
```

There is a close relationship between the selection monad and the well-known continuation monad

$$\mathscr{K}_R X = (X \to R) \to R$$

which in Haskell is

```
data Cont r x = Cont {runCont :: (x → r) → r}
```

[1] http://www.eecs.qmul.ac.uk/~julesh/

(Note that in Haskell the constructor of *Cont* is written *cont* as a consequence of the continuation monad being, in reality, defined in terms of the continuation monad transformer.) Every selection function $\varepsilon : \mathscr{J}_R X$ induces a continuation $\overline{\varepsilon} : \mathscr{K}_R X$ by

$$\overline{\varepsilon} p = p(\varepsilon p)$$

This operation is a monad morphism from $\mathscr{J}_R$ to $\mathscr{K}_R$. When φ is a continuation satisfying $\varphi = \overline{\varepsilon}$ we say that ε *attains* φ. Not every continuation is attainable, for example a constant continuation $\varphi p = r_0$ is not attainable because we can find some $p : X \to R$ such that r_0 is not in the image of p. However many important continuations are attainable, and in those cases selection functions offer several advantages.

The canonical examples of continuations are the maximum operator $\max : \mathscr{K}_{\mathbb{R}} X$ for a finite set X defined by

$$\max p = \max_{x \in X} px$$

and the existential quantifier $\exists : \mathscr{K}_{\mathbb{B}} X$, where $\mathbb{B} = \{\bot, \top\}$, defined by

$$\exists p = \begin{cases} \top & \text{if } px = \top \text{ for some } x \in X \\ \bot & \text{otherwise} \end{cases}$$

Both of these continuations are attained: max is attained by the operator $\arg\max$ which produces a point at which p attains its maximum, and $\exists$ is attained by Hilbert's ε operator. Another interesting example is the integral operator

$$\int : \mathscr{K}_{\mathbb{R}} [0,1]$$

defined by

$$\int p = \int_0^1 px \, \mathrm{d}x$$

A combination of the mean value theorem and the axiom of choice proves that this continuation is attained. This example is interesting because a computable continuation is attained by a noncomputable selection function (the selection function $\arg\max$ also has this property if we replace the finite set X with a compact topological space with infinitely many points, such as the unit interval).

All (strong) monads M have a monoidal product

$$\otimes : MX \times MY \to M(X \times Y)$$

which can be expressed in terms of unit and bind. (In fact there are always two monoidal products, but we are interested only in one of them.) A monomorphic iterated form of this product is present in the Haskell prelude as

```
sequence :: (Monad m) => [m a] -> m [a]
```

It is important to note that due to the restrictions of Haskell's type system we can only take the iterated product of selection functions and continuations which have the same type. In a dependently typed language we could express the general product, which for selection functions has type

$$\bigotimes : \prod_i \mathscr{J}_R X_i \to \mathscr{J}_R \prod_i X_i$$

For continuations the binary product is given by

$$(\varphi \otimes \psi)q = \varphi(\lambda x^X.\psi(\lambda y^Y.q(x,y)))$$

As special cases this includes min-maxes in game theory, composition of quantifiers such as $\exists x^X \forall y^Y.q(x,y)$, and multiple integration over products of $[0,1]$, as well as combinations of these such as

$$\max_{x \in X} \int_0^1 q(x,y)\,\mathrm{d}y$$

The product of selection functions is a more complex operation, given by

$$(\varepsilon \otimes \delta)q = (a, b_a)$$

where

$$\begin{aligned} a &= \varepsilon(\lambda x^X.q(x,b_x)) \\ b_x &= \delta(\lambda y^Y.q(x,y)) \end{aligned}$$

Because the overline operation is a monad morphism it commutes with the monoidal products:

$$\overline{\varepsilon \otimes \delta} = \overline{\varepsilon} \otimes \overline{\delta}$$

In other words, if ε attains φ and δ attains ψ then $\varepsilon \otimes \delta$ attains $\varphi \otimes \psi$.

One of the most important and remarkable properties of selection functions is that the product of selection functions is well-defined when iterated infinitely, so long as R is discrete and q is continuous. Moreover the Haskell function *sequence* specialised to the selection monad will terminate on an infinite list, provided the Haskell datatype r is discrete in the sense of (Escardó 2008), since computable functions are continuous. Other well-known monads, including the continuation monad, do not have this property.

3 SAT solving with the continuation monad transformer

The selection function

```
ε :: Sel Bool Bool
ε = Sel $ λp → p True
```

will solve a simple optimisation problem: given a function $\boxed{p :: \mathit{Bool} \to \mathit{Bool}}$ the selection function ε will find x making $p\ x$ true, if one exists. Then

```
sequence $ repeat ε
```

will, given a function $\boxed{q :: [Bool] \to Bool}$, find xs making $q\,xs$ true, if one exists. In other words, this function will find satisfying assignments of propositional formulas.

However, the classical SAT problem is only to decide whether a satisfying assignment exists, rather than to actually compute one. By construction, if a formula q has a satisfying assignment then $(\otimes\varepsilon)q$ is a satisfying assignment. Therefore q is satisfiable iff $q((\otimes\varepsilon)q)$ is true, that is, if $(\overline{\otimes\varepsilon})q$ is true. If the product is finite this is equal to $(\otimes\exists)q$, and $\exists$ can be written directly in Haskell as

```
∃ :: Cont Bool Bool
∃ = cont $ λp → p $ p True
```

Using *sequence* for the monoidal product yields an extremely small, self-contained Haskell SAT solver:

```
import Control.Monad.Cont
sat n = runCont $ sequence $ replicate n $ cont $ λp → p $ p True
```

The type is $\boxed{sat :: Int \to ([Bool] \to Bool) \to Bool}$, so it is not a true SAT solver in the sense that it takes its input as a function $\boxed{[Bool] \to Bool}$ rather than in a discrete form such as a clause-set. The other input is the number of variables to search, which is necessary because *sequence* specialised to continuations will diverge on infinite lists. It is also important to stress that this algorithm is not a SAT solver written in continuation-passing style, rather it uses the continuation monad to directly represent the recursion.

Using the continuation monad transformer we can begin to refine this algorithm, for example we can write a DPLL-like algorithm using a state monad to store clause-sets. The DPLL algorithm, introduced in (Davis et al. 1960), decides the satisfiability of CNF-formulas by successively extending the formula with either a literal or its negation, and at each stage applying two simplifying transformations, namely unit clause propagation and pure literal elimination. Most modern SAT solvers are based on DPLL combined with various heuristics to improve average-case complexity, see for example (Marques-Silva et al. 2008).

For simplicity we implement only unit clause propagation. The algorithm we will implement is represented in imperative pseudocode in algorithm 1.

We begin with a datatype representing literals:

```
data Literal = Positive Int | Negative Int
```

so the type of a clause-set is $[[Literal]]$. The top-level function will be

```
dpll   :: Int → [[Literal]] → Bool
dpll n = evalState s . initialState
```

Algorithm 1 Imperative DPLL algorithm

```
function DPLL(φ)
    if φ is an empty clause-set then
      return True
    end if
    if φ contains the empty clause then
      return False
    end if
    for each unit clause l in φ do
        for each clause c in φ do
            if c contains l then
                φ ← remove c from φ
            end if
            if c contains l̄ then
                c ← remove l̄ from c
            end if
        end for
    end for
    l ← next literal
      return DPLL(φ ∧ l) ∨ DPLL(φ ∧ l̄)
end function
```

```
where
  s :: State DPLL Bool
  s = runContT (sequence $ replicate n φ) q

  φ :: ContT Bool (State DPLL) Bool
  φ = ContT $ λp → p True >>= p
```

Notice that in φ the expression `p $ p True` is replaced by its monad transformer equivalent `p True >>= p`.

The type *DPLL* must represent the state used by the DPLL algorithm, and we need a function `initialState :: [[Literal]] → DPLL`. Most of the actual algorithm is contained in the query function

```
q :: [Bool] → State DPLL Bool
```

The implementation of this function is given in appendix A. It is important to note that the monad transformer stack `ContT Bool (State DPLL)` will thread a single state through an entire search, however for the DPLL algorithm we want to create a new copy of the state for every recursive call. We achieve this by representing the recursion tree explicitly, making the state type *DPLL* a type of binary trees with leaves labelled by the necessary data. The idea is that the function q will move up the tree according to its input, interpreting *True* as ‘go left’ and *False* as ‘go right’. If it finds a leaf labelled by a clause set for which satisfiability is trivial, the function returns. If not, it extends the tree according to the remaining input. A full implementation is presented in appendix A.

As written, this program is not particularly efficient. Potentially we could use the *IO* monad rather than *State*, and produce an optimised SAT solving algorithm for example by storing the clause sets as arrays rather than lists. However optimised SAT solvers and other search algorithms require explicit control over the backtracking, which is precisely what the continuation and selection monads do not allow. In (Bauer and Pretnar 2012) is presented an alternative implementation of the product of selection functions that allows explicit control over backtracking. See the next section for a discussion of how *sequence* explores a search space.

4 The selection monad transformer

In creating a selection monad transformer, our guiding example is the generalisation in the Haskell monad transformer library from the continuation monad

$$\mathscr{K}_R X = (X \to R) \to R$$

to the continuation monad transformer

$$\mathscr{K}_R^M X = \mathscr{K}_{MR} X = (X \to MR) \to MR$$

Paralleling this, the selection monad

$$\mathscr{J}_R X = (X \to R) \to X$$

is generalised to the selection monad transformer

$$\mathscr{J}_R^M X = (X \to MR) \to MX$$

The Haskell code for the monad instance is

```
data SelT r m x = SelT {runSelT :: (x → m r) → m x}

instance (Monad m) ⟹ Monad (SelT r m)
  where
    return = SelT . const . return
    ε >>= f = SelT $ λp → let
                              g x = runSelT (f x) p
                              h x = g x >>= p
                            in
                            runSelT ε h >>= g
```

This comes from taking the instance declaration for the ordinary selection monad and replacing certain function applications with the monadic bind of *m*. Similarly we obtain a monad morphism from selections to continuations by replacing a function application with monadic bind:

$$\begin{aligned}&\mathit{toCont} :: (\mathit{Monad}\ m) \implies \mathit{SelT}\ r\ m\ x \to \mathit{ContT}\ r\ m\ x\\ &\mathit{toCont}\ \varepsilon = \mathit{ContT}\ \$\ \lambda p \to \mathit{runSelT}\ \varepsilon\ p \gg= p\end{aligned}$$

In type-theoretic notation we continue to write this operation as an overline $\overline{\varepsilon}$. A full code listing is given in appendix B.

A simple but extremely useful example is to use $\mathscr{J}_R^{\mathrm{IO}}$ to perform a verbose backtracking search, laying bare the subtle behaviour of the product of selection functions which could previously only be investigated using *unsafePerformIO*. For example we can write a function

$$\mathit{verboseQuery} :: ([\mathit{Bool}] \to \mathit{Bool}) \to [\mathit{Bool}] \to \mathit{IO}\ \mathit{Bool}$$

which will print information about the query and its result before returning. Then a verbose SAT solver is given by

$$\mathit{verboseSAT}\ n = (\mathit{runSelT}\ \$\ \mathit{sequence}\ \$\ \mathit{replicate}\ n\ \$\ \mathit{SelT}\ (\$\ \mathit{True}))\ .\ \mathit{verboseQuery}$$

Experiments with this function confirm that *sequence* will prune large parts of a search space when it is able to, but it also duplicates a certain amount of work, potentially calling q on the same input several times. It also sometimes continues to call q even after it has found a satisfying assignment (that is, after *verboseQuery* has printed the result *True*). The problem of explaining exactly how *sequence* explores a search space is still open, although the experimental data provided by *verboseSAT* and related functions may make progress possible.

For the application to nondeterministic games we need to find an explicit expression for the monoidal product

$$\otimes : \mathscr{J}_R^M X \times \mathscr{J}_R^M Y \to \mathscr{J}_R^M (X \times Y)$$

Starting with the usual expression for the product of selection functions

$$(\varepsilon \otimes \delta) q = (a, b_a)$$

where

$$\begin{aligned} a &= \varepsilon(\lambda x^X . q(x, b_x))\\ b_x &= \delta(\lambda y^Y . q(x, y))\end{aligned}$$

we replace a function application with monadic bind, and the cartesian pairing with the monoidal product $\otimes$ of M, yielding

$$(\varepsilon \otimes \delta) q = a \otimes b_a$$

where

$$\begin{aligned} a &= \varepsilon(\lambda x^X . (b_x \gg= \lambda y^Y . q(x, y)))\\ b_x &= \delta(\lambda y^Y . q(x, y))\end{aligned}$$

Writing q in a curried form, this is

$$a = \varepsilon(\lambda x^X.(b_x >\!\!>\!\!= qx))$$
$$b_x = \delta(qx)$$

5 Nondeterministic sequential games

In the remaining two sections we write $X \rightrightarrows Y$ for the type of nondeterministic functions from X to Y. This is equal to the Kleisli arrow $X \to \mathscr{P}Y$ where $\mathscr{P}$ is the nondeterminism monad (whose underlying functor is the covariant powerset functor). In Haskell we replace $\mathscr{P}$ with the list monad for simplicity. In order to distinguish between the use of the list monad as 'poor man's nondeterminism' and its ordinary use as an ordered data structure (such as for an ordered list of moves in a game) we use a type synonym

```
type 𝒫 a = [a]
```

Definition 1 (Finite nondeterministic sequential game)**.** *An n-player sequential game consists of types $X_1, \ldots, X_n$ of moves, a type R of outcomes and an outcome function*

$$q : \prod_{i=1}^{n} X_i \to R$$

A play *of the game is a tuple $(x_1, \ldots, x_n) \in \prod_{i=1}^{n} X_i$, and $q(x_1, \ldots, x_n)$ is called the* outcome *of the play.*

In a nondeterministic sequential game, we instead take the outcome function to be a nondeterministic function

$$q : \prod_{i=1}^{n} X_i \rightrightarrows R$$

and we consider $q(x_1, \ldots, x_n)$ to be the set of all possible outcomes of the play $(x_1, \ldots, x_n)$.

In classical game theory the outcome type will be $\mathbb{R}^n$, and then we interpret the ith element of the tuple $q(x_1, \ldots, x_n)$ to be the profit of the ith player.

As a running example, we will consider a simple 2-player nondeterministic game implemented in Haskell, the full code of which is presented in appendix C. Each player has a choice of moves given by

```
data Move = Cautious | Risky
```

The game will be zero-sum, so its outcome will be a single integer with the first player maximising and the second minimising. The nondeterministic outcome function will be given by

```
type Outcome = Int
```

$$\begin{array}{ll} q & :: [\mathit{Move}] \rightarrow \mathscr{P}\ \mathit{Outcome} \\ q\ [\mathit{Cautious}, \mathit{Cautious}] & = [0] \\ q\ [\mathit{Cautious}, \mathit{Risky}] & = [-1, 0, 1] \\ q\ [\mathit{Risky}, \mathit{Cautious}] & = [-1, 0, 1] \\ q\ [\mathit{Risky}, \mathit{Risky}] & = [-2, -1, 0, 1, 2] \end{array}$$

Consider the ith player of a game deciding which moves to play, having observed the previous moves $x_1, \ldots, x_{i-1}$. Assuming that the players to follow are playing according to certain constraints (such as rationality) which are common knowledge, the player can associate to each possible move x_i a set of outcomes which may result after the other players' moves. Thus the player has a nondeterministic function $X_i \rightrightarrows R$, which we call the *context* of player i's move.

Definition 2 (Policies of players)**.** *An* outcome policy *for player i is a rule that associates each context $p : X_i \rightrightarrows R$ with a set of outcomes which the player considers to be* good *following that context. Thus an outcome policy for player i is a continuation with type*

$$\varphi_i : \mathscr{K}_R^{\mathscr{P}} X_i$$

A move policy *for player i is a rule which, given a context, nondeterministically chooses a move. Thus a move policy for player i is a selection function with type*

$$\varepsilon_i : \mathscr{J}_R^{\mathscr{P}} X_i$$

A move policy ε_i is called rational *for an outcome policy φ_i if for every context p and every move x that may be selected by the move policy, every outcome that could result from the context given that move is a good outcome. Symbolically,*

$$\{r \in px \mid x \in \varepsilon_i p\} \subseteq \varphi_i p$$

The left hand side of this is precisely $\overline{\varepsilon_i} p$, where $\overline{\cdot}$ is the monad morphism $\mathscr{J}_R^{\mathscr{P}} \to \mathscr{K}_R^{\mathscr{P}}$. (However the subset relation prevents us from obtaining a game-theoretic interpretation of more general instances of the selection monad transformer.)

An outcome policy is called realistic *iff, for every context, there is a move such that every possible outcome of that move in that context is a good outcome. In other words, an outcome policy is realistic iff it has a rational move policy that never selects the empty set of moves.*

For deterministic games there are canonical move policies in the case when the outcome type is $\mathbb{R}^n$, namely maximisation with respect to the ordering of $\mathbb{R}$:

$$\varepsilon_i p = \underset{x \in X_i}{\arg\max} (px)_i$$

where $(px)_i$ is the ith coordinate projection of the vector $px : \mathbb{R}^n$. However for nondeterministic games there is no canonical policy, even when the outcome type is $\mathbb{R}$: a player might always choose x such that px contains the largest possible element, or they might have a more complicated policy to mitigate risk, for example preferring possible outcomes $\{5\}$ to $\{10, -100\}$. We leave this undetermined and allow the policies to be arbitrary.

We will consider two pairs of move policies for the example game, also called *cautious* and *risky*. For the implementation of these we begin with the type

$$\textbf{type}\ Choice = \mathscr{P}\ (Move,\ \mathscr{P}\ Outcome) \to (Move,\ \mathscr{P}\ Outcome)$$

representing 'choice functions' which, given a set of move-outcome pairs, will choose one. Next we write a function called *argopt*, which is a generic way to convert a choice function into a selection function.

$$\begin{aligned} &argopt\ \ :: Choice \to SelT\ Outcome\ \mathscr{P}\ Move \\ &argopt\, f = SelT\ \$\ \lambda p \to [fst\ \$\ f\ [(Cautious,\ p\ Cautious),\ (Risky,\ p\ Risky)]] \end{aligned}$$

It should be noted that Haskell does not allow a type synonym with a free type variable to be used as a parameter, so $\boxed{SelT\ Outcome\ \mathscr{P}\ Move}$ must be written $\boxed{SelT\ Outcome\ []\ Move}$ in real code.

Now we implement our two pairs of choice functions:

$$\begin{aligned} &riskymax &&= maximumBy\ \$\ comparing\ \$\ maximum\ .\ snd \\ &riskymin &&= minimumBy\ \$\ comparing\ \$\ minimum\ .\ snd \\ &cautiousmax &&= riskymax\ .\ filter\ (all\ (\geq\ (-1))\ .\ snd) \\ &cautiousmin &&= riskymin\ .\ filter\ (all\ (\leq\ 1)\ .\ snd) \end{aligned}$$

In words, the risky move policies will select moves to *maximise* the player's *maximum* possible gain (which for the second player is *minimising* the *minimum* outcome), ignoring the risk of bad outcomes. The cautious move policies do likewise, but avoid moves that lead to the possibility of the worst possible outcome, which is ± 2. The risky move policies are deterministic in the sense that they always return a list of length 1. The cautious move policies also have this property for the particular game we are considering, but not in general (indeed, as written the functions *cautiousmax* and *cautiousmin* are partial functions: if the *filter* returns an empty list then an exception will be thrown).

Definition 3 (Strategy)**.** *A strategy for player i in a game is a function*

$$\sigma_i : \prod_{j=1}^{i-1} X_j \to X_i$$

which chooses a move given the observed previous moves. For a nondeterministic sequential game we take the strategies to be nondeterministic functions

$$\sigma_i : \prod_{j=1}^{i-1} X_j \rightrightarrows X_i$$

Thus in particular a strategy for player 1 is $\sigma_1 : \mathscr{P}X_1$. A tuple σ containing a strategy for each player is called a strategy profile.

We can understand this definition intuitively by imagining that nondeterminism is resolved at several points during the course of playing a game. First, player 1 nondeterministically chooses a move. Then the nondeterminism resolves, and some concrete move is played. Player 2 observes the actual move that was played, and nondeterministically chooses a move, and so on. Eventually, after all players have moved, we obtain a concrete play $(x_1, \dots, x_n)$. Then the rules of the game nondeterministically determine an outcome, and finally this nondeterminism resolves to produce the actual outcome.

6 Backward induction for nondeterministic games

Now we need to define what it means for a strategy profile to be optimal. In game-theoretical language an optimal strategy profile is called a *subgame-perfect Nash equilibrium*.

Intuitively, a strategy profile is optimal if for all players i and all partial plays up to player $i-1$, every outcome that could result from playing the strategy profile is a good outcome in the context which, following player i's move, the remaining players play according to the strategy profile.

We spell out this definition explicitly. Given a partial play $\vec{x} = (x_1, \dots, x_{i-1})$, the set all moves that might be made by player $i \leq j \leq n$ by playing the strategy profile σ is given inductively by

$$b_j^{\vec{x}} = \{x \in \sigma_j(\vec{x}, x_i, \dots, x_{j-1}) \mid x_k \in b_k^{\vec{x}},\ i \leq k < j\}$$

which in monad notation is

$$b_j^{\vec{x}} = \left(\bigotimes_{k=i}^{j-1} b_k^{\vec{x}} \right) \ggg\!= \sigma_j \vec{x}$$

where $\otimes$ is the monoidal product of M, which for $M = \mathscr{P}$ is the cartesian product. The set of all outcomes that might result from playing the strategies σ on the partial play $\vec{x}$ is

$$\{r \in q(\vec{x}, x_i, \dots, x_n) \mid x_k \in b_k^{\vec{x}},\ i \leq k \leq n\}$$

Now we need to define the context in which player i chooses a move. Suppose after the partial play $\vec{x}$, player i chooses the move $x \in X_i$. The set of outcomes that could result from this is simply the result of the previous calculation applied to the partial play $(\vec{x}, x)$, namely

$$\{r \in q(\vec{x}, x, x_{i+1}, \dots, x_n) \mid x_k \in b_k^{\vec{x},x},\ i < k \leq n\}$$

Definition 4 (Optimal strategy profile). *A strategy profile σ is called optimal for given outcome policies φ_i if for all partial plays $\vec{x} = (x_1, \dots, x_{i-1})$ for $1 \leq i \leq n$ we have*

$$\{r \in q(\vec{x}, x_i, \dots, x_n) \mid x_k \in b_k^{\vec{x}},\ i \leq k \leq n\} \subseteq \varphi_i p$$

where the context $p : X_i \rightrightarrows R$ is defined by

$$px = \{r \in q(\vec{x}, x, x_{i+1}, \dots, x_n) \mid x_k \in b_k^{\vec{x},x},\ i < k \leq n\}$$

Notice that an optimal strategy profile guarantees good outcomes for all players, despite the nondeterminism. In order for this to hold there must be enough good outcomes, for which it is sufficient that the outcome policies are realistic.

Theorem 1. *A nondeterministic sequential game in which all players' outcome policies are realistic has an optimal strategy profile.*

If the players' move policies are ε_i and the outcome function is q then an optimal strategy profile is given by

$$\sigma_i \vec{x} = \pi_i \left(\left(\bigotimes_{j=i}^{n} \varepsilon_j \right) (q\vec{x}) \right)$$

where $\otimes$ is the monoidal product in the monad $\mathscr{J}_R^{\mathscr{P}}$ and π_i is the projection onto X_i. These can be computed in Haskell by

```
strategy es q xs = head $ runSelT ε (q . (xs ++))
  where
    ε = sequence $ drop (length xs) es
```

Moreover the set of all plays which may occur from playing this strategy profile is given by

$$\left(\bigotimes_{i=1}^{n} \varepsilon_i \right) q$$

which is computed in list form by the program

```
plays = runSelT . sequence
```

The corresponding expression using the monoidal product in $\mathscr{J}_R$ was shown in (Escardó and Oliva 2012) to be a generalisation of backward induction, a standard and intuitive algorithm in classical game theory to find equilibria of sequential games. By taking the monoidal product in $\mathscr{J}_R^{\mathscr{P}}$, we also gain a generalisation of backward induction to nondeterministic games.

It was remarked above that the appearance of the subset relation in the definition of rational move policies and optimal strategy profiles prevents us from giving these definitions for an arbitrary monad M. The definition given applies to submonads of the nondeterminism monad, which include finite nondeterminism (finite lists), the exception (*Maybe*) monad and the identity monad (in which case we get Escardó and Oliva's definitions). It might be possible to interpret the subset relation for some other monads in *ad hoc* ways, for example as a refinement relation for the *IO* monad, which should have a game-theoretic meaning as games which interact with an environment outside of the game. However, the given expressions and Haskell functions to compute optimal strategy profiles works for any M, so we can say that this strategy profile is always optimal, despite having no general definition. It remains to be seen what game-theoretic intuition exists for the many different monads that are available.

We prove the theorem in the simple case of a 2-move sequential game. Let the move types be X and Y, so the outcome function has type $q : X \times Y \rightrightarrows R$. Let the outcome policies be $\varphi : \mathscr{K}_R^{\mathscr{P}} X$ and $\psi : \mathscr{K}_R^{\mathscr{P}} Y$, and the rational move policies be $\varepsilon : \mathscr{J}_R^{\mathscr{P}} X$ and $\delta : \mathscr{J}_R^{\mathscr{P}} Y$. We need to prove that the strategy profile

$$\begin{aligned} \sigma_X &= \{x \in X \mid (x,y) \in (\varepsilon \otimes \delta) q\} \\ \sigma_Y x &= \delta(qx) \end{aligned}$$

is optimal. Using the expression for the monoidal product in the previous section we have

$$(\varepsilon \otimes \delta)q = \{(x,y) \in X \times Y \mid x \in a, y \in b_x\}$$

where

$$\begin{aligned} a &= \varepsilon\,(\lambda x.\{r \in q(x,y) \mid y \in b_x\}) \\ b_x &= \delta(qx) \end{aligned}$$

therefore the first player's strategy is given directly by

$$\sigma_X = a$$

We can directly calculate

$$\begin{aligned} b_X^{()} &= a \\ b_Y^{()} &= \{y \in \delta(qx) \mid x \in a\} \\ b_Y^{(x)} &= \delta(qx) \end{aligned}$$

The conditions we need to verify are

1. $\{r \in q(x,y) \mid x \in a, y \in \delta(qx)\} \subseteq \varphi(\lambda x.\{r \in q(x,y) \mid y \in \delta(qx)\})$
2. For all $x : X$, $\{r \in q(x,y) \mid y \in \delta(qx)\} \subseteq \psi(qx)$

Since δ is a rational move policy for ψ we have $\{r \in py \mid y \in \delta p\} \subseteq \psi p$ for all contexts $p : Y \rightrightarrows R$. Using the context $py = q(x,y)$ gives the second condition. For the first condition we take the context

$$px = \{r \in q(x,y) \mid y \in \delta(qx)\}$$

so that $a = \varepsilon p$. Then we have

$$\begin{aligned} &\{r \in q(x,y) \mid x \in a, y \in \delta(qx)\} \\ &= \{r \in px \mid x \in \varepsilon p\} \\ &\subseteq \varphi p \\ &= \varphi(\lambda x.\{r \in q(x,y) \mid y \in \delta(qx)\}) \end{aligned}$$

This completes the proof.

For the example game the optimal plays can be computed interactively:

$$\begin{aligned} &\mathit{plays}\ [\mathit{argopt\ cautiousmax}, \mathit{argopt\ cautiousmin}]\ q \\ \Longrightarrow\ &[[\mathit{Risky}, \mathit{Cautious}]] \\ &\mathit{plays}\ [\mathit{argopt\ cautiousmax}, \mathit{argopt\ riskymin}]\ q \\ \Longrightarrow\ &[[\mathit{Cautious}, \mathit{Risky}]] \\ &\mathit{plays}\ [\mathit{argopt\ riskymax}, \mathit{argopt\ cautiousmin}]\ q \\ \Longrightarrow\ &[[\mathit{Risky}, \mathit{Cautious}]] \\ &\mathit{plays}\ [\mathit{argopt\ riskymax}, \mathit{argopt\ riskymin}]\ q \\ \Longrightarrow\ &[[\mathit{Risky}, \mathit{Risky}]] \end{aligned}$$

Thus, each combination of cautious and risky move policies results in a different deterministic play. The intuition behind these results is that the ‘personality’ defined by a move policy is common knowledge of the players. The only possibly unexpected result is the first, when both players are cautious. In this case the first player plays risky because she knows that the second player will avoid the maximum risk, meaning the risky move is safe.

A DPLL implementation

```
import Data.List
import Control.Monad.State
import Control.Monad.Trans.Cont

-- Literals

data Literal = Positive Int | Negative Int deriving (Show, Eq)

negateLiteral :: Literal -> Literal
negateLiteral (Positive n) = Negative n
negateLiteral (Negative n) = Positive n

-- Internal state of algorithm

data DPLL = Leaf {simplified :: Bool, clauses :: [[Literal]]}
          | Node {trueBranch :: DPLL, falseBranch :: DPLL}

-- Top level of algorithm

dpll :: Int -> [[Literal]] -> Bool
dpll n = evalState s . initialState where
     s = runContT (sequence $ replicate n e) q

e :: ContT Bool (State DPLL) Bool
e = ContT $ \p -> p True >>= p

q :: [Bool] -> State DPLL Bool
q bs = do s <- get
          let (s', b) = queryState s bs 0
          put s'
          return b

-- Interactions with internal state

initialState :: [[Literal]] -> DPLL
initialState cs = Leaf {simplified = False, clauses = cs}
```

```
queryState :: DPLL -> [Bool] -> Int -> (DPLL, Bool)
queryState (Leaf False cs) bs n = queryState (Leaf True $ simplify cs) bs n
queryState s@(Leaf True []) _ _ = (s, True)
queryState s@(Leaf True [[]]) _ _ = (s, False)
queryState (Leaf True cs) bs n = queryState (Node l r) bs n where
           l = Leaf False ([Negative n] : cs)
           r = Leaf False ([Positive n] : cs)
queryState (Node l r) (True : bs) n = (Node l' r, b) where
           (l', b) = queryState l bs (n + 1)
queryState (Node l r) (False : bs) n = (Node l r', b) where
           (r', b) = queryState r bs (n + 1)

-- Operations on clause sets

simplify :: [[Literal]] -> [[Literal]]
simplify cs = if null units
              then cs
              else simplify (foldl (flip propagateUnit) cs' units) where
         (units, cs') = partition isUnitClause cs

isUnitClause :: [Literal] -> Bool
isUnitClause [_] = True
isUnitClause _ = False

propagateUnit :: [Literal] -> [[Literal]] -> [[Literal]]
propagateUnit [l] = (map $ filter (/= negateLiteral l))
                  . (filter $ notElem l)
```

B The selection monad transformer library

```
-- Selection monad transformer library
-- A generic backtracking search and auto-pruning library

module SelT (SelT(..), Sel, toCont,
             boundedBinarySearch, unboundedBinarySearch) where

import Control.Monad.Cont
import Data.Functor.Identity

-- Selection monad transformer

newtype SelT r m x = SelT {runSelT :: (x -> m r) -> m x}

instance (Monad m) => Monad (SelT r m) where
         return = SelT . const . return
```

```
        e >>= f = SelT $ \p -> let g x = runSelT (f x) p
                                   h x = g x >>= p
                                in runSelT e h >>= g

instance (MonadIO m) => MonadIO (SelT r m) where
        liftIO = lift . liftIO

instance MonadTrans (SelT r) where
        lift = SelT . const

-- Monad morphism from selections to continuations

toCont :: (Monad m) => SelT r m x -> ContT r m x
toCont e = ContT $ \p -> runSelT e p >>= p

-- Vanilla selection monad

type Sel r = SelT r Identity

-- Generic search functions

unboundedBinarySearch :: (Monad m) => SelT Bool m [Bool]
unboundedBinarySearch = sequence $ repeat $ SelT ($ True)

boundedBinarySearch :: (Monad m) => Int -> SelT Bool m [Bool]
boundedBinarySearch n = sequence $ replicate n $ SelT ($ True)
```

C Example game

```
import SelT
import Data.List (maximumBy, minimumBy)
import Data.Ord (comparing)

data Move    = Cautious | Risky deriving (Show)
type Outcome = Int
type P a = [a]

q :: [Move] -> P Outcome
q [Cautious, Cautious] = [0]
q [Cautious, Risky]    = [-1, 0, 1]
q [Risky, Cautious]    = [-1, 0, 1]
q [Risky, Risky]       = [-2, -1, 0, 1, 2]

type Choice = P (Move, P Outcome) -> (Move, P Outcome)
```

```
argopt :: Choice -> SelT Outcome [] Move
argopt f = SelT $ \p -> [fst $ f [(Cautious, p Cautious), (Risky, p Risky)]]

riskymax, riskymin, cautiousmax, cautiousmin :: Choice
riskymax = maximumBy $ comparing $ maximum . snd
riskymin = minimumBy $ comparing $ minimum . snd
cautiousmax = riskymax . filter (all (>= (-1)) . snd)
cautiousmin = riskymin . filter (all (<= 1) . snd)

es1, es2, es3, es4 :: [SelT Outcome [] Move]
es1 = [argopt riskymax, argopt riskymin]
es2 = [argopt riskymax, argopt cautiousmin]
es3 = [argopt cautiousmax, argopt riskymin]
es4 = [argopt cautiousmax, argopt cautiousmin]

solve :: [SelT Outcome [] Move] -> P [Move]
solve es = runSelT (sequence es) q
```

References

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers, 2012. URL `http://math.andrej.com/2012/03/08/programming-with-algebraic-effects-and-handlers/`.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1960.

Martin Escardó. Seemingly impossible functional programs, 2007. URL `http://math.andrej.com/2007/09/28/seemingly-impossible-functional-programs/`.

Martin Escardó. Exhaustible sets in higher-type computation. *Logical methods in computer science*, 4 (3:3):1–37, 2008.

Martin Escardó and Paulo Oliva. Selection functions, bar recursion and backward induction. *Mathematical structures in computer science*, 20(2):127–168, 2010a.

Martin Escardó and Paulo Oliva. What sequential games, the Tychonoff theorem and the double-negation shift have in common. *Proc. of 3rd ACM SIGPLAN Wksh. on Mathematically Structured Functional Programming*, 2010b.

Martin Escardó and Paulo Oliva. Sequential games and optimal strategies. *Proc R Soc A*, 467:1519–1545, 2011.

Martin Escardó and Paulo Oliva. Computing Nash equilibria of unbounded games. *Proceedings of the Turing centenary conference*, 2012.

Jules Hedges. A generalisation of Nash's theorem with higher-order functionals. *Proc R Soc A*, 469 (2154), 2013.

Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsch, editors, *Handbook of satisfiability*, chapter 4. IOS Press, 2008.