JFP **19** (6): 645–661, 2009. © Cambridge University Press 2009
doi:10.1017/S0956796809990220 First published online 7 September 2009

FUNCTIONAL PEARL

A domain-specific language for experimental game theory

ERIC WALKINGSHAW and MARTIN ERWIG

School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331, USA
(*e-mail:* {walkiner,erwig}@eecs.oregonstate.edu)

Abstract

Experimental game theory is increasingly important for research in many fields. Unfortunately, it is poorly supported by computer tools. We have created Hagl, a domain-specific language embedded in Haskell, to reduce the development time of game-theoretic experiments and make the definition and exploration of games and strategies simple and fun.

1 Introduction

Experimental game theory is the use of game-theoretic models in simulations and experiments to understand strategic behavior. It is an increasingly important research tool in many fields, including economics, biology, and many social sciences (Camerer 2003), but computer support for such projects is primarily found only in custom programs written in general-purpose languages.

Here we present Hagl,[1] a domain-specific language embedded in Haskell, intended to drastically reduce the development time of such experiments and make the definition and exploration of games and strategies simple and fun.

In game theory, a game is a situation in which agents interact by playing *moves*, with the goal of maximizing their own *payoff*. In Hagl, a game is a tree:

```
data GameTree mv = Decision PlayerIx [(mv, GameTree mv)]
                 | Chance [(Int, GameTree mv)]
                 | Payoff [Float]
```

The `Payoff` nodes form the leaves of a game tree. Payoffs represent outcomes, in the form of scores, for each player at the end of a game. The tree `Payoff [1,2]` indicates that the first player receives one point and the second player two points. The type of awarded payoffs could be generalized to any instance of the standard `Num` type class, but this would inflate type signatures throughout Hagl while providing little benefit.

[1] Short for "Haskell game language" and loosely intended to evoke the homophone "haggle." Available for download at http://web.engr.oregonstate.edu/~walkiner/hagl/.

Internal nodes are either `Decision` nodes or `Chance` nodes and can be sequenced arbitrarily with the tree. The `Decision` nodes represent locations in the game tree at which a player must choose to make one of several moves. `PlayerIx` is a type synonym for `Int` and indicates which player must make the decision. An association list maps available moves to their resulting subtrees. In the following very simple game, the first (and only) player is presented with a decision; if they choose move `A`, they will receive zero points, but if they choose move `B`, they will receive five points:

```
easyChoice = Decision 1 [(A, Payoff [0]), (B, Payoff [5])]
```

Finally, the `Chance` nodes represent points at which an external random force pushes the players down some path or another based on a distribution. Distributions are given here by a list of subtrees prefixed with their relative likelihood; given `[(1,a),(3,b)]`, the subtree `b` is three times as likely to be chosen as `a`. A random die roll, while not technically a game (since there are no decisions and thus no players), is illustrative and potentially useful as a component in a larger game. It can be represented as a single `Chance` node, where each outcome is equally likely and the payoff of each is the number showing on the die:

```
die = Chance [(1, Payoff [n]) | n <- [1..6]]
```

This tree-oriented representation is known as *extensive form* in game theory and provides a flexible and powerful representation upon which many different types of games can be built. However, it also has sometimes undesirable side effects. Many games require moves by each player to be played simultaneously, without knowledge of the other players' moves. This is not possible with the simple tree representation, which forces decisions to be made in sequence as we traverse a path through the tree. The reachable payoffs from a later player's decision node are constrained by the moves made by earlier players, essentially revealing the earlier players' moves.

As a solution, game theory introduces *information groups*. An information group is a set of decision nodes for the same player, from which a player knows only the group she is in, not the specific node. A game with information groups of size one is said to have *perfect information*, while a game with potentially larger groups has *imperfect information*. In Hagl, information groups are represented straightforwardly:

```
data InfoGroup mv = Perfect (GameTree mv)
                  | Imperfect [GameTree mv]
```

Whenever players attempt to view their current position in the tree, Hagl returns a value of type `InfoGroup` rather than a `GameTree` value directly, obfuscating the exact position if necessary. Game definitions must therefore provide a way to get the information group associated with any decision node in the game tree.

A complete game definition in Hagl is thus a game tree, a function to provide the information group of nodes in the tree, and the number of players to play the game:

```
data Game mv = Game { numPlayers :: Int,
                      info       :: GameTree mv -> InfoGroup mv,
                      tree       :: GameTree mv }
```

In the next section we will look at a specific game in more detail, the *iterated prisoner's dilemma*, which will provide both motivation for Hagl and a vehicle for introducing increasingly advanced functionality.

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

		Player 2	
		C	**D**
Player 1	**C**	2, 2	0, 3
	D	3, 0	1, 1

Fig. 1. Normal-form representation of the prisoner's dilemma.

2 The iterated prisoner's dilemma

The prisoner's dilemma is a game in which each player must choose to either "cooperate" or "defect." Defection yields the higher payoff regardless of the other player's choice, but if both players cooperate they will do better than if they both defect. The game is typically represented as a matrix of payoffs indexed by each player's move, as shown in Figure 1, a notation known as *normal form*. In the iterated form, the game is played repeatedly, with the payoffs of each iteration accumulating.

Robert Axelrod's seminal book *The Evolution of Cooperation* (Axelrod 1984) provides one of the best documented successes of experimental game theory. In 1980 Axelrod held an iterated prisoner's dilemma tournament. Game theorists from around the world submitted strategies to the competition. The winning strategy was "Tit for Tat," a much simpler strategy than many of its opponents; it cooperates in the first game and thereafter plays the move last played by its opponent. Thus, if an opponent always cooperates, Tit for Tat will always cooperate, but if an opponent defects, Tit for Tat will retaliate by defecting on the next turn. The surprising success of such a simple strategy turned out to be a breakthrough in the study of cooperative behavior.

A 2004 attempt to recreate and extend Axelrod's experiments highlights the need for domain-specific language support for experimental game theory. Experimenters wrote a custom Java library (IPDLX) to run the tournament (Kendall *et al.* 2005). Excluding user interface and example packages, this library is thousands of lines of code. This represents a huge amount of effort that, since the library is heavily tied to the prisoner's dilemma, cannot be easily reused in different experiments.

Hagl provides a general platform for creating and running experiments, enabling the concise definition of the Axelrod tournament below. First we define the prisoner's dilemma and then a tournament in which each player faces every player (including themselves) for a 1000-game match, printing the final score of each:

```
data Dilemma = Cooperate | Defect

pd :: Game Dilemma
pd = matrix [Cooperate, Defect] [[2,2],[0,3],
                                 [3,0],[1,1]]

axelrod :: [Player Dilemma] -> IO ()
axelrod players = roundRobin pd players (times 1000 >> printScore)
```

A player who plays the Tit-for-Tat strategy can also be defined concisely:

```
tft :: Player Dilemma
tft = "Tit for Tat" ::: play Cooperate `atFirstThen` his (last game's move)
```

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

The `:::` operator takes a name and a strategy and produces a `Player` value. The combinators used to define strategies are named from a first-person perspective; that is, `his (last game's move)` (or `her (last game's move)`) refers to the move played by the opponent's strategy in the previous game, whereas `my (last game's move)` refers to the move played by this strategy in the previous game. These combinators will be explained in more detail in Section 5, but first let's turn our attention to the definition of normal-form games.

3 Normal-form games

The `matrix` function used in the definition of `pd` in the previous section is just one of Hagl's many smart constructors for defining different types of normal-form games. These functions build on terminology and notation familiar to game theorists. The type of the most general normal-form game constructor is given below:

```
normal :: Int -> [[mv]] -> [[Float]] -> Game mv
```

This function takes the following as arguments:

- the number of players, n, that play the game;
- a list of lists of available moves for each player, $m_1, m_2, \ldots, m_n$; and
- a list of $|m_1| \times |m_2| \times \ldots \times |m_n|$ payoffs, where each individual payoff has length n (one value for each player).

It produces a game tree with depth $n + 1$, one level for each player's decision, plus one for the payoff nodes. The generated game tree for the prisoner's dilemma is shown below. Hagl includes pretty-printing functions for viewing game trees more concisely, but the verbose rendering is used here for explicitness:

```
Decision 1 [(Cooperate, Decision 2 [(Cooperate, Payoff [2.0,2.0]),
                                    (Defect,    Payoff [0.0,3.0])]),
            (Defect,    Decision 2 [(Cooperate, Payoff [3.0,0.0]),
                                    (Defect,    Payoff [1.0,1.0])])]
```

Recall from Section 1 that in addition to a game tree, a `Game` value contains a function `info` for returning the information group associated with each decision node in the tree. Normal-form games have imperfect information, since all moves are assumed to be made simultaneously. For any decision node, the associated information group contains all other decision nodes at the same depth in the game tree.

The `matrix` function used to define `pd` is a specialization of `normal`, which makes the common assumptions of two players and a square matrix in which the same moves are available to each player:

```
matrix :: [mv] -> [[Float]] -> Game mv
matrix ms = normal 2 [ms,ms]
```

An additional assumption that is made in a large subset of the so-called matrix games is that for any outcome, the scores of both players sum to zero. These are known as *zero-sum* games and can be defined in Hagl with the following smart constructor:

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

```
zerosum :: [mv] -> [Float] -> Game mv
zerosum ms vs = matrix ms [[v, -v] | v <- vs]
```

These games represent situations in which one player wins and another loses. A simple example is the traditional game rock–paper–scissors, defined below:

```
data RPS = Rock | Paper | Scissors
rps = zerosum [Rock, Paper, Scissors] [0,-1, 1,
                                       1, 0,-1,
                                      -1, 1, 0]
```

Here, only one value is given for each outcome and the payoff awarded to the first player. The second player's payoff can be automatically derived from this to produce a zero-sum game.

Hagl strives to be concise and familiar to game theorists by utilizing existing terminology and notations, allowing users to define the large class of normal-form games easily. In the next section we introduce operators and constructors for easing the definition of extensive-form games as well.

4 Extensive-form games

Recall from Section 1 that games in Hagl are internally represented as trees, a representation known as *extensive form*. While the internal representation is intentionally austere, a few conveniences are afforded for defining larger games which are best represented in extensive form. To illustrate these, we will work through the definition of an extensive-form representation of the Cuban Missile Crisis, based on an example in Straffin (1993).

In this simplified representation of the crisis there are two players, the USSR and the USA, represented as follows:

```
ussr = player 1
usa  = player 2
```

The `player` function has type `PlayerIx -> (mv, GameTree mv) -> GameTree mv`; it takes a player index and a single decision branch and creates a decision node. This allows for flexible creation of decision trees when combined with the `<|>` operator introduced below.

One reason the USSR sent nuclear weapons to Cuba was as a response to US nuclear weapons in Turkey. Thus we have at least two factors which could contribute to the payoff of each country. American missiles in Turkey are a strategic advantage for the US and a disadvantage for the USSR, while Soviet missiles in Cuba are an advantage for the USSR and a disadvantage for the US. We can represent these outcomes with simple `Payoff` nodes:

```
nukesInTurkey = Payoff [-2, 1]
nukesInCuba   = Payoff [ 1,-2]
```

Note that the penalty of having nuclear weapons near you outweighs the benefit of having nuclear weapons near your opponent. Essentially, this means that both countries would prefer to have nuclear weapons in neither country than in both

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

countries. Additionally, if nuclear war breaks out, it would be devastating to both countries, so we'll assign big negative payoffs to that outcome:

```
nuclearWar = Payoff [-100,-100]
```

Alternatively, if we suspect that some nuclear wars are worse than others, we might model this possibility with a `Chance` node, giving one-to-four odds of a truly devastating war:

```
nuclearWar = Chance [(1, Payoff [-100,-100]), (4, Payoff [-20,-20])]
```

Finally, with a happy resolution, there won't be nuclear weapons in either country:

```
noNukes = Payoff [0,0]
```

Let's assume at the start of the game that the American missiles are already in Turkey. The USSR has the first decision to make: they can choose to send nuclear weapons to Cuba, or they can choose to do nothing. We represent this choice as follows:

```
start = ussr ("Send Missiles to Cuba", usaResponse)
        <|> ("Do Nothing", nukesInTurkey)
```

If the USSR chooses to send missiles to Cuba, the US must respond; if the USSR chooses to do nothing, the game is over and the payoff is simply the result of having nuclear weapons in Turkey. Here we introduce the decision-branching operator `<|>`, which has type `GameTree mv -> (mv, GameTree mv) -> GameTree mv` but is only defined for `Decision` nodes. Each option is represented by a tuple containing a move and the corresponding branch to follow if the move is played. The `player` function creates a decision node from a single branch (normally a decision node requires a list of such tuples), and the branching operator appends additional branches to this node.

The USA's potential response to missiles in Cuba is modeled as follows:

```
usaResponse = usa ("Do Nothing", nukesInTurkey <+> nukesInCuba)
              <|> ("Blockade", ussrBlockadeCounter)
              <|> ("Air Strike", ussrStrikeCounter)
```

The US could choose to do nothing, in which case there are missiles in both Turkey and Cuba. This introduces a second operator `<+>`, which combines two game trees. For the `Decision` and `Chance` nodes this means concatenating all branches; for the `Payoff` nodes, the payoffs are simply added together. If the US chooses action, by either a naval blockade or an air strike, the USSR must counterrespond:

```
ussrBlockadeCounter = ussr ("Agree to Terms", noNukes)
                      <|> ("Escalate", nuclearWar)
ussrStrikeCounter = ussr ("Pull Out", nukesInTurkey)
                    <|> ("Escalate", nuclearWar)
```

Since every path through the decision tree terminates with a payoff node, the game tree is complete. However, we still need to turn this Hagl `GameTree` value into a `Game` value. For this, the smart constructor `extensive` is provided, which has type `GameTree mv -> Game mv`. This function traverses the game tree (Hagl provides

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

helper functions for accessing game tree nodes in both breadth-first and depth-first orders) and extracts the number of players from finite-sized trees. All nodes in an extensive-form game defined in this way are assumed to have perfect information.

As an interesting aside that shows Hagl in action, we can use this game to demonstrate a possible weakness of the traditional minimax strategy in non-zero-sum games. Hagl provides a built-in implementation of this strategy named `minimax`, which determines the "optimal" move from the current location in the game tree (assuming perfect information and a finite depth). Let's assume that the Americans use this strategy:

```
kennedy = "Kennedy" ::: minimax
```

For the Soviets, we will define an approximation of what their strategy may have been – send missiles to Cuba, but avoid nuclear war at all costs:

```
khrushchev = "Khrushchev" :::
    play "Send Missiles to Cuba" `atFirstThen`
    do m <- his move `inThe` last turn
       play $ case m of "Blockade" -> "Agree to Terms"
                        "Air Strike" -> "Pull Out"
```

Don't worry about completely understanding this strategy now; the next section covers the strategy combinator library in depth.

Running the game once and printing its transcript produces the following output:

```
> runGame crisis [khrushchev, kennedy] (once >> printTranscript)
Game 1:
  Khrushchev's move: "Send Missiles to Cuba"
  Kennedy's move: "Do Nothing"
  Payoff: [-1.0,-1.0]
```

At first glance, Kennedy's decision, as determined by the minimax strategy, seems reasonable. The risk of nuclear war is so terrible that the best move is to play passively, ensuring that it won't occur. However, if we assume that Khrushchev is rational, then even without knowing his strategy beforehand we can be sure that he won't choose to pursue nuclear war, since it would result in a huge negative payoff for the USSR as well.

The minimax algorithm assumes that the opposing player is trying to minimize the current player's score. But this assumption doesn't hold in non-zero-sum games. In game theory a player's only goal is to maximize his or her *own* score – minimizing an opponent's score is only a side effect when payoffs sum to zero.

The best response from Kennedy would seem to be an air strike, since it would allow the US to keep their nuclear weapons in Turkey. That history did not follow this course could mean that Kennedy played this game suboptimally. Much more likely, our model could be incomplete. The payoffs do not account for international reputation, national pride, economic impact, and many other factors that certainly played a role in the actual crisis. Fortunately, the `<+>` and `<|>` operators make adding these additional payoff modifiers, and additional decisions that may result, very straightforward.

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

Now that we can define many types of games, we would like to play them. In the next section we introduce Hagl's suite of smart constructors for defining simple strategies and in the subsequent subsections a library of combinators for defining more complex ones.

5 Defining strategies

Strategies in Hagl are computations executed within the outermost of two layers of state monad transformers. Fortunately, Hagl is designed to make strategies easy to define without understanding the intricacies of the representation. Many of the types in this section will be left undefined until Sections 6 and 7, on game execution and player and strategy representation. In this section we focus on the concrete syntax of strategy definition.

The simplest strategies in game theory just return the same move every time. Game theorists call these *pure* strategies. Hagl provides a function `pure` which takes a move and produces a pure strategy.

Also common in game theory are *mixed* strategies, which play a move based on some distribution. The following strategy cooperates with a probability of 5/6 and defects with a probability of 1/6:

```
rr = "Russian Roulette" ::: mixed [(5,Cooperate), (1,Defect)]
```

Other strategy functions include `randomly`, which randomly selects one of the available moves (based on a linear distribution), and `periodic`, which cyclically plays a sequence of moves. There is also the built-in `minimax` strategy introduced in Section 4.

More complex strategies, like Tit for Tat defined in Section 2 and Khrushchev in Section 4, can be built from a suite of functions designed for the task. These functions primarily fall into two categories: *data accessors* and *list selectors*.

5.1 Data accessors

Strategies have access to a wealth of information about the current state of a game's execution. In Section 6 we define this state explicitly, while in this section we describe the interface presented to strategies for accessing it. The data accessor functions extract data from the execution state and transform it into some convenient form.

Before we get to the accessors themselves, we must introduce a set of types which are central to strategy definition in Hagl. These types simply wrap a list of data, indicating whether each element in the list corresponds to a particular game iteration, turn, or player:

```
newtype ByGame a = ByGame [a]
newtype ByTurn a = ByTurn [a]
newtype ByPlayer a = ByPlayer [a]
```

These act as type-level annotations of the contents of a list and provide many advantages. Firstly, they help programmers understand the structure of values

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

returned by accessors functions. More importantly, they enforce through the Haskell type system a proper ordering of the list selectors described in the next subsection. For the rare case in which one wishes to handle all types of dimensioned lists generically, a type class `ByX` is provided, which unifies the the three types and provides generic `toList` and `fromList` functions which convert dimensioned lists into standard Haskell lists and vice versa.

Below we list a useful subset of the data accessors provided by Hagl, along with their types and a brief description. The `GameMonad` type class present in each of the types will be explained in Section 7:

- `location :: GameMonad m mv => m (InfoGroup mv)`
  The information group of the current node in the game tree.
- `numMoves :: GameMonad m mv => m (ByPlayer Int)`
  The total number of moves played by each player so far.
- `moves :: GameMonad m mv => m (ByGame (ByPlayer (ByTurn mv)))`
  A triply nested list of all moves played so far, indexed first by game, then by player, and then by the order in which they were performed.
- `payoff :: GameMonad m mv => m (ByGame (ByPlayer Float))`
  A doubly nested list of the payoff received by each player in each game.
- `score :: GameMonad m mv => m (ByPlayer Float)`
  The current cumulative scores, indexed by player.

Working directly with the values returned by these functions would be cumbersome. Different indexing conventions are used for each type of list, and although the types help to prevent mixing them up, there are still many details to keep track of. Although we know, for example, that the `score` accessor returns a list of scores indexed by player, if we are writing a strategy, how do we know which score corresponds to our own? The list selectors presented in the next subsection manage these minutiae for us.

5.2 List selectors

Two features distinguish Hagl selectors from generic list operators. First, they provide increased type safety by only operating on lists of the appropriate type. Second, they may use information from the execution state to make different selections depending on the context in which they are run. Below are the list selectors for `ByPlayer` lists:

- `my :: GameMonad m mv => m (ByPlayer a) -> m a`
  Select the element corresponding to the current player.
- `her :: GameMonad m mv => m (ByPlayer a) -> m a`
  `his :: GameMonad m mv => m (ByPlayer a) -> m a`
  Select the element corresponding to the other player in a two-player game.
- `our :: GameMonad m mv => m (ByPlayer a) -> m [a]`
  Select the elements corresponding to all players (i.e., all elements).
- `their :: GameMonad m mv => m (ByPlayer a) -> m [a]`
  Select the elements corresponding to every player except the current player.

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

These selectors have names corresponding to possessive pronouns from the first-person perspective of the current player. For example, the expression `my score` returns the current player's score. Because selectors are context sensitive, two different strategies can refer to `my score` and get the two different scores corresponding to their respective players.

The selectors for `ByGame` and `ByTurn` lists share a set of adjectivally named functions through the use of a type class named `ByGameOrTurn`. An example of one such selector is the function `last` which returns the element corresponding to either the last game iteration or the last turn in this iteration, depending on the types of its arguments:

```
last :: (ByGameOrTurn d, GameMonad m mv) => d a -> m (d a) -> m a
```

The first argument to `last` is used only for its type and to increase readability. For example, the function `last game's payoff`, where `game's` is declared as `undefined :: ByGame a`, would select the payoff corresponding to the previous game iteration. A similar undefined value `turn's` has type `ByTurn a` for specifying `ByTurn` selectors. The variations `games'`, `turns'`, and `turn` are all included to maximize readability in different situations.

Other `ByGame` and `ByTurn` selectors include the following:

- `first :: (ByGameOrTurn d, GameMonad m mv) => d a -> m (d a) -> m a`
  Select the element corresponding to the first game iteration or turn.
- `every :: (ByGameOrTurn d, GameMonad m mv) => d a -> m (d a) -> m [a]`
  Select the elements corresponding to all iterations or turns (i.e., all elements).
- `this :: GameMonad m mv => ByGame a -> m (ByGame a) -> m a`
  Select the elements corresponding to the current game iteration.

Note that `ByTurn` lists do not have an element corresponding to the current turn, so the `this` selector applies to `ByGame` lists only. This is enforced by the type of the function.

One might suspect that the first argument of a `ByGameOrTurn` selector is superfluous, since the type of a list can be determined from the list itself. However, there is one accessor function whose return type varies depending on the types of the selectors applied to it. The `move` data accessor was not introduced in Section 5.1 but has been used in the strategies of both Tit for Tat and Khrushchev; it is defined in the following type class:

```
class (ByX d, ByX e) => MoveList d e where
  move :: GameMonad m mv => m (d (e mv))
```

This is a multi-parameter type class, requiring an extension to Haskell 98 available in the Glasgow Haskell Compiler (GHC; GHC 2004).

There are two instances of the class `MoveList`. The first defines a `move` function which returns lists of type `ByGame (ByPlayer mv)`, a doubly nested list of the last move played by each player in each game. This is most useful for games in which each player makes only a single move, as in most normal-form games. The expression `his (last game's move)` in Tit for Tat's strategy relies on this implementation.

The second instance of `MoveList` defines a `move` function that returns a list of type `ByPlayer (ByTurn mv)`, a list of each move played by each player in this game.

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

Khrushchev uses this version of `move` in the expression `his move ‘inThe‘ last turn`. The `inThe` function is defined as `flip ($)` and is used to reorder selectors to improve readability.

Most of the selectors above can be easily composed. For example, `my (last game’s payoff)` applies a `ByGame` selector followed by a `ByPlayer` selector to an accessor of the appropriate type. Unfortunately, this composability breaks down after selectors, which return a *list* of elements. For example, we cannot write `my (every games’ payoff)` because `every` returns a value of type `m [ByPlayer Float]` rather than the `m (ByPlayer Float)` that `ByPlayer` selectors accept. That the list is the result of a monadic computation complicates things further, making it cumbersome to use standard list operations like `map`.

As a solution, Hagl introduces the `eachAnd` operator which composes two selectors by mapping the first over the list returned by the second. Using `eachAnd` we can get a list of our payoffs for every game by writing `my ‘eachAnd’ every game’s payoff`.

We can build some pretty sophisticated strategies using only the tools we have seen so far. For example, consider another important iterated prisoner’s dilemma strategy called the “Grim Trigger” that cooperates until the opponent defects once, after which it defects forever:

```
grim = "Grim Trigger" :::
    do ms <- her ‘eachAnd‘ every games’ move
       if Defect ‘elem‘ ms then play Defect else play Cooperate
```

The only unknown function in this strategy is `play` which is just a synonym for the monadic `return`. The expression `her ‘eachAnd’ every games’ move` returns a list of all previous moves played by the opponent. If any of these moves represent defection, Grim Trigger will defect; otherwise it will cooperate.

Notice that the implementation of Grim Trigger above is robust in that it will do the correct thing even when the move history is empty (i.e., in the first game). In general, this is not always possible. For these other cases, a small suite of strategy composition functions are provided for the initialization of strategies.

5.3 Initializing strategies

Many strategies rely on information from previous iterations and require one or more temporary initial strategies until that information is available. Since this is very common, it is essential to have concise idioms for expressing these strategies. The definition of Tit for Tat in Section 2 demonstrates the simplest such case, where a single move must be played before the primary strategy is applicable. This situation is captured by Hagl’s `atFirstThen` function which takes two strategies, a strategy to play for the first move and one to play thereafter, combining them to form a single strategy:

```
atFirstThen :: Strategy mv s -> Strategy mv s -> Strategy mv s
```

The strategy of the following player, named after Pavlov’s dogs, utilizes `atFirstThen` and the built-in `randomly` function to play a random move in the first game, before moving on to the main body of the strategy:

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

```
pavlov = "Pavlov" :::
    randomly `atFirstThen`
    do p <- my (last game's payoff)
       m <- my (last game's move)
       if p > 0 then return m else randomly
```

This player is intended for use in zero-sum games such as the rock–paper–scissors game defined in Section 3. The strategy plays randomly in the first game and then plays its own previous move if that move earned a positive payoff. Otherwise, it continues to play randomly.

The `atFirstThen` function is defined in terms of the more general function `thereafter`, which takes a list of initial strategies and a primary strategy to play when that list is exhausted:

```
thereafter :: [Strategy mv s] -> Strategy mv s -> Strategy mv s
thereafter ss s = my numMoves >>= \n -> if n < length ss then ss !! n else s
```

For example, the following strategy would play rock in the first game, paper in the second game, and scissors thereafter:

```
plan = [play Rock, play Paper] `thereafter` play Scissors
```

Throughout this section we have referred to the game execution state, but we have only defined it indirectly by the type of data that can be extracted from it. In the next section we are much more explicit, defining the monadic structure underlying Hagl and how it is used in game execution.

6 Game execution

In Hagl, game execution occurs within the game execution monad. Execution is performed in steps, where each step corresponds to processing one node in the game tree. When a payoff node is reached, information about the completed game is saved to a history of past iterations and state corresponding to the current execution is reset. The game execution monad is defined by the following type:

```
newtype GameExec mv a = GameExec (StateT (ExecState mv) IO a)
```

This type wraps a state monad transformer and is itself an instance of `Monad`, `MonadState`, and `MonadIO`, simply deferring to the `StateT` monad it wraps in all cases. The innermost monad is the `IO` monad, which is needed for printing output and obtaining random numbers.

The state maintained by the `GameExec` monad is a value of type `ExecState`, which contains all of the information needed for game execution and to write strategies for an iterated game:

```
data ExecState mv = ExecState (Game mv) [Player mv] (ByPlayer Int)
                              (GameTree mv) (Transcript mv) (History mv)
```

The arguments to the `ExecState` constructor represent, in order,

- the game being played,
- the players currently playing the game,

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

- the total number of moves made by each player,
- the current location in the traversal of the game tree,
- a transcript of events in this iteration, and
- a history of past iterations.

The definitions of `Game` and `GameTree` were given in Section 1. The definition of the `Player` type will be given in Section 7. Here, let's examine the `Transcript` and `History` types, the values of which are generated by game execution.

A `Transcript` is a list of `Event` values, where each event corresponds to an already-processed node in the game tree:

```
type Transcript mv = [Event mv]
data Event mv = DecisionEvent PlayerIx mv
              | ChanceEvent Int
              | PayoffEvent [Float]
```

As nodes are traversed, their corresponding events are generated and appended to the transcript. A `DecisionEvent` records a decision made at a `Decision` node, capturing the index of the player involved and the move they played; a `ChanceEvent` records the branch taken at a `Chance` node; and a `PayoffEvent` records the payoffs awarded at a `Payoff` node.

`History` values are essentially just lists of past transcripts, combined with a `Summary` value which redundantly stores only the moves made by each player and the final payoffs for a particular iteration, for performance reasons:

```
type History mv = ByGame (Transcript mv, Summary mv)
type Summary mv = (ByPlayer (ByTurn mv), ByPlayer Float)
```

Many of the data accessors in Section 5.1 extract their information from `History` value in the game execution state.

Games are executed by running computations within the `GameExec` monad. The most fundamental of these is the `step` function, which has type `GameExec m ()`; that is, it is a computation within the `GameExec` monad with no return value. Each time `step` is run, it processes one node in the game tree, updating the location, transcript, game summaries, and payoffs, all stored in the `ExecState` value, accordingly. For example, if the current location in the game tree is a `Decision` node, `step` will request a move from the player indicated by the `Decision` node, update the location based on that move, and add an event to the transcript.

Since `step` is a monadic computation, multiple invocations of `step` can be sequenced using the monad bind operation (`>>`). The function `step >> step >> step` could be used to run one iteration of the prisoner's dilemma; since the depth of the prisoner's dilemma is three, it takes three executions of `step` to fully evaluate a single game. Fortunately, Hagl provides a function `once` which recursively executes `step`, running a game to completion a single time, obviating the need to know the depth of a game to run it. Additionally, the function `times` takes an integer and runs the game that many times. For example, `times 100` runs a game 100 times consecutively.

The `runGame` function, whose type is given below, is used to execute computations like `once` on combinations of players and games:

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

```
runGame :: Game m -> [Player m] -> GameExec m a -> IO (ExecState m)
```

This function takes a game, a list of players, and a computation in the `GameExec` monad; it generates an initial `ExecState` and then runs the given computation on that state. The following, run from an interactive prompt (e.g., `GHCi`), would play 10 iterations of the prisoner's dilemma between players `a` and `b`:

```
> runGame pd [a,b] (times 10)
```

Unfortunately, running 10 iterations of the prisoner's dilemma isn't very useful if we don't know anything about the results of those games. In addition to `step`, `once`, and `times`, Hagl provides a suite of functions for printing information about the current state of game execution. The most generally useful of these are `printTranscript`, which prints the transcript of all completed games, and `printScore`, which prints the current score of each player. In order to see these in action, let us first define a couple of simple players, one that always cooperates and one that always defects:

```
mum = "Mum" ::: pure Cooperate
fink = "Fink" ::: pure Defect
```

From an interactive prompt we can now instruct Hagl to run one game of the prisoner's dilemma between Mum and Fink, print the transcript of that game, run 99 more games, and print the final score:

```
> runGame pd [mum, fink] (once >> printTranscript >> times 99 >> printScore)
Game 1:
  Mum's move: Cooperate
  Fink's move: Defect
  Payoff: [0.0,3.0]
Score:
  Mum: 0.0
  Fink: 300.0
```

Clearly, pure cooperation is not a good strategy against pure defection.

Often we want to run a game between not just one set of opponents but a series of opponents, as in the Axelrod tournament described in Section 2. The function `runGames` provides a general means of doing so:

```
runGames :: Game m -> [[Player m]] -> GameExec m a -> IO ()
```

The type of `runGames` is similar to that of `runGame`, except that in place of a list of players, there is a list of lists of players. In `runGames`, the provided computation is executed once for each list of players. The scores of all players are accumulated, and the final scores printed.

The definition of the `axelrod` function in Section 2 uses the function `roundRobin`, one of a few functions for producing standard types of tournaments. This function takes a single list of players and runs every unique combination (including each player against itself). Below we run `axelrod` with the two players defined above and Tit for Tat from Section 2. Only the final score is shown below; scores of individual matchups are omitted for space:

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

```
> axelrod [mum, fink, tft]
Final Scores:
  Tit for Tat: 6999.0
  Fink: 6002.0
  Mum: 6000.0
```

Although still finishing last, here Mum performs much better relative to Fink, since it has both Tit for Tat and itself to cooperate with. Tit for Tat outperforms both by being willing to cooperate and by being flexible enough to avoid being exploited by Fink.

In the next section we finally give a formal definition of player and strategy representations in Hagl. For strategies, we introduce another monadic layer that allow each strategy to maintain its own independent state.

7 Player and strategy representation

A player in Hagl is represented by the `Player` data type defined below. Values of this type contain the player's name (e.g., "Tit for Tat"), an arbitrary state value, and a strategy which may utilize that state:

```
data Player mv = forall s. Player Name s (Strategy mv s)
```

The `Name` type is just a synonym for `String`. A player's name is used both to distinguish among different players within Hagl and for labelling output like player scores. Also notice that `Player` is a locally quantified data constructor (Läufer & Odersky 1994), another extension to Haskell 98 available in the GHC. This allows players to maintain their own different state types while still being stored and manipulated generically.

The definition of the `Strategy` type requires introducing the outermost monadic layer in Hagl. The `StratExec` monad, defined below, wraps the `GameExec` monad introduced in Section 6, providing an additional layer of state management available to strategies through the standard `get` and `put` functions:

```
newtype StratExec mv s a = StratExec (StateT s (GameExec mv) a)
```

Like `GameExec`, `StratExec` instantiates `Monad`, `MonadState`, and `MonadIO`. Additionally, both `GameExec` and `StratExec` instantiate the `GameMonad` type class first mentioned in Section 5.1. This allows many combinators, including all of the accessor and selector functions, to be used identically (without lifting) from within either monad.

Finally, a `Strategy` is a computation within the `StratExec` monad which returns the next move to be played:

```
type Strategy mv s = StratExec mv s mv
```

Since many strategies in Hagl do not require the use of state, an additional data constructor is provided for defining players with stateless strategies:

```
(:::) :: Name -> Strategy mv () -> Player mv
```

This constructor has been used throughout the paper to define players with stateless strategies. Since we have made it this far without state in strategies, one may wonder whether this extra complexity is justified. In the next section we make a case for stateful strategies and provide an example of their use.

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

8 Stateful strategies

We have mostly ignored the role of state in strategy definitions thus far. This is partly because stateless strategies simply look nicer than their stateful counterparts but also because the examples have all been small enough that the benefits of maintaining state are minimal.

Since the entire game execution history is available to every strategy at any time, state is not strictly necessary for the definition of any strategy. Any stateful strategy could be transformed to a stateless strategy by regenerating the state from scratch at every iteration. However, since experimental game theorists will often want to run thousands of iterations of each game with many different strategies, state becomes a necessity for any strategies which consider all past iterations, especially as the amount of work at each iteration increases. The construction of Markov models, which have particular relevance in iterated games, are one such example of state which would be prohibitively expensive to regenerate from scratch at every iteration.

On a smaller scale, consider the definition of Grim Trigger from Section 5.2. This implementation runs in linear time with respect to the number of iterations played, since it must search the list of previous moves each time it is called. By utilizing state, we can define a constant-time Grim Trigger as follows[2]:

```
grim' = Player "Stately Grim Trigger" False $
    play Cooperate `atFirstThen`
    do m <- her (last game's move)
       triggered <- update (|| m == Defect)
       if triggered then play Defect else play Cooperate
```

The function `update` used here applies a function to the state within a state monad and then stores and returns the resulting state. Instead of scanning the entire history of past games, Stately Grim Trigger considers only the most recent game and updates its state accordingly. Even with this simple example, 10,000 iterations involving `grim'` complete in a few seconds, while the same experiment with `grim` takes over 30 minutes on our hardware.

9 Conclusion

Although originally intended as analytical tools, game theoretic models have proven extremely conducive to research through simulation and experimentation. Despite the utility of experimental game theory, however, there does not seem to be much in the way of language support. Hagl attempts to lower the barriers to entry for researchers using experimental game theory. By utilizing existing formalisms and notations, Hagl provides a familiar interface to domain experts, and by embedding the language in Haskell, users can extend and supplement the language with arbitrary Haskell code as needed. Additionally, Hagl makes heavy use of the Haskell type system to statically ensure that list selectors and other language elements are ordered correctly.

[2] This example actually has a clever, stateless, constant-time algorithm as well – defect if either you or your opponent defected on the previous move – but this will not always be the case.

https://doi.org/10.1017/S0956796809990220 Published online by Cambridge University Press

In addition to the types of games presented here, Hagl can be used to define games which are more naturally described by transitions between states. For example, in the paper and pencil game of tic-tac-toe, a state of nine squares is maintained, each of which may be empty or contain an "X" or an "O"; players' moves are defined by transforming the state by marking empty squares. Similarly, board games like chess are defined by the state of their board, and moves alter that state by moving pieces around. Hagl provides smart constructors which transform state-based game definitions into standard Hagl game trees. Thanks to the laziness of the host language, these trees are generated on demand, making even large state-based games tractable.

This project is part of a larger effort to apply language design concepts to game theory. In our previous work we have designed a visual language for defining strategies for normal-form games, which focused on the explainability of strategies and on the traceability of game executions (Erwig & Walkingshaw 2008). In future work we plan to extend Hagl by allowing for multiple internal game representations. This would facilitate the use of existing algorithms for computing equilibria in normal-form games and ease the writing of strategies for state-based games and auctions, which are of particular interest in experimental game theory.

Acknowledgements

We would like to thank Richard Bird and an anonymous reviewer for their thorough and insightful reviews. This paper is significantly improved by their efforts.

References

Axelrod, R. M. (1984) *The Evolution of Cooperation*. Basic Books.

Camerer, C. (2003) *Behavioral Game Theory: Experiments in Strategic Interaction*. Princeton University Press.

Erwig, M. & Walkingshaw, E. (2008) A Visual Language for Representing and Explaining Strategies in Game Theory. *Pages 101–108 of:* Bottoni et al. (ed), *IEEE Int. Symp. on Visual Language and Human-Centric Computing*, Herrsching am Ammersee, Germany. Washington, DC: IEEE Computer Society.

Glasgow Haskell Compiler. (2004) The Glasgow Haskell compiler [online]. Available at: `http://haskell.org/ghc` (Accessed 23 August 2009).

Kendall, G., Darwen, P. & Yao, X. (2005) The Prisoner's Dilemma competition [online]. Available at: `http://www.prisoners-dilemma.com` (Accessed 23 August 2009).

Läufer, K. & Odersky, M. (1994) Polymorphic type inference and abstract data types, *ACM Trans. Program. Lang. Syst.*, 16 (5): 1411–1430.

Straffin, P. D. (1993) *Game Theory and Strategy*. The Mathematical Association of America.