Varying Domain Representations in Hagl

Extending the Expressiveness of a DSL for Experimental Game Theory

Eric Walkingshaw and Martin Erwig

School of Electrical Engineering and Computer Science,
Oregon State University, Corvallis, OR 97331, USA

Abstract. Experimental game theory is an increasingly important research tool in many fields, providing insight into strategic behavior through simulation and experimentation on game theoretic models. Unfortunately, despite relying heavily on automation, this approach has not been well supported by tools. Here we present our continuing work on Hagl, a domain-specific language embedded in Haskell, intended to drastically reduce the development time of such experiments and support a highly explorative research style.

In this paper we present a fundamental redesign of the underlying game representation in Hagl. These changes allow us to better utilize domain knowledge by allowing different classes of games to be represented differently, exploiting existing domain representations and algorithms. In particular, we show how this supports analytical extensions to Hagl, and makes strategies for state-based games vastly simpler and more efficient.

1 Introduction

Game theory has traditionally been used as an analytical framework. A game is a formal model of a strategic situation in which players interact by making moves, eventually achieving a payoff in which each player is awarded a value based on the outcome of the game. Classical game theorists have derived many ways of solving such situations, by mathematically computing "optimal" strategies of play, usually centered around notions of stability or equilibrium [1].

The derivation of these optimal strategies, however, almost always assumes perfectly rational play by all players—an assumption which rarely holds in practice. As such, while these methods have many practical uses, they often fair poorly at predicting *actual* strategic behavior by humans and other organisms.

One striking example of sub-optimal strategic behavior is the performance of humans in a class of simple guessing games [2]. In one such game, a group of players must each guess a number between 0 and 100 (inclusive). The goal is to guess the number closest to 1/2 of the average of all players' guesses. Traditional game theory tells us that there is a single equilibrium strategy for this game, which is to choose zero. Informally, the reasoning is that since each player is rational and assumes all other players are rational, any value considered above zero would lead the player to subsequently consider 1/2 of that value. That is, if

W.M. Taha (Ed.): DSL 2009, LNCS 5658, pp. 310–334, 2009.

a player initially thinks the average of the group might be 50 (based on randomly distributed guesses), 25 would initially seem a rational guess. However, since all other players are assumed to be rational, the player would assume that the others came to the same conclusion and would also choose 25, thereby making 12.5 a better guess. But again, all players would come to the same conclusion, and after halving their guesses once again it quickly becomes clear that an assumption of rationality among all players leads each to recursively consider smaller and smaller values, eventually converging to zero, which all rational players would then play.

It has been demonstrated experimentally, however, that most human players choose values greater than zero [3]. Interestingly, a nonzero guess does not necessarily imply irrationality on the part of the player, who may be rational but assumes that others are not. Regardless, it is clear that the equilibrium strategy is sub-optimal when playing against real opponents.

Experimental game theory attempts to capture, understand, and predict strategic behavior where analytical game theory fails or is difficult to apply. The use of game theoretic models in experiments and simulations has become an important research tool for economists, biologists, political scientists, and many other researchers. This shift towards empirical methods is also supported by the fact that game theory's formalisms are particularly amenable to direct execution and automation [4]. Interactions, decisions, rewards, and the knowledge of players are all formally defined, allowing researchers to conduct concise and quantifiable experiments.

Perhaps surprisingly, despite the seemingly straightforward translation from formalism to computer automation, general-purpose tool support for experimental game theory is extremely low. A 2004 attempt to recreate and extend the famous Axelrod experiments on the evolution of cooperation [5] exemplifies the problem. Experimenters wrote a custom Java library (IPDLX) to conduct the experiments [6]. Like the original, the experiments were structured as tournaments between strategies submitted by players from around the world, competing in the iterated form of a game known as the *prisoner's dilemma*. Excluding user interface and example packages, the library used to run the tournament is thousands of lines of Java code. As domain-specific language designers, we knew we could do better.

Enter Hagl[1], a domain-specific language embedded in Haskell designed to make defining games, strategies, and experiments as simple and fun as possible. In Hagl, the Axelrod experiments can be reproduced in just a few lines.

```
data Cooperate = C | D
dilemma = symmetric [C, D] [2, 0, 3, 1]
axelrod players = roundRobin dilemma players (times 100)
```

In our previous work on Hagl [7] we have provided a suite of operators and smart constructors for defining common types of games, a combinator library

[1] Short for "Haskell game language".
Available for download at: `http://web.engr.oregonstate.edu/~walkiner/hagl/`

for defining strategies for playing games, and a set of functions for carrying out experiments using these objects. Continuing the example above, we provide a few example players defined in Hagl below, then run the Axelrod experiment on this small sample of players.

At each iteration of the iterated prisoner's dilemma, each player can choose to either cooperate or defect. The first two very simple players just play the same move every time.

```
mum = "Mum" `plays` pure C
fink = "Fink" `plays` pure D
```

A somewhat more interesting player is one that plays the famous "Tit for Tat" strategy, winner of the original Axelrod tournament. Tit for Tat cooperates in the first iteration, then always plays the previous move played by its opponent.

```
tft = "Tit for Tat" `plays` (C `initiallyThen` his (prev move))
```

With these three players defined, we can carry out the experiment. The following, run from an interactive prompt (e.g. GHCi), plays each combination of players against each other 100 times, printing out the final scores.

```
> axelrod [mum,fink,tft]
Final Scores:
  Tit for Tat: 699.0
  Fink: 602.0
  Mum: 600.0
```

Since this original work we have set out to add various features to the language. In this paper we primarily discuss two such additions, the incorporation of operations from analytical game theory, and adding support for state-based games, which together revealed substantial *bias* in Hagl's game representation.

1.1 Limitations of the Game Representation

The definition of the prisoner's dilemma above is nice since it uses the terminology and structure of notations in game theory. Building on this, we would like to support other operations familiar to game theorists. In particular, we would like to provide analytical functions that return equilibrium solutions like those mentioned above. This is important since experimental game theorists will often want to use classically derived strategies as baselines in experiments. Unfortunately, algorithms that find equilibria of games like the prisoner's dilemma rely on the game's simple structure. This structure, captured in the concrete syntax of Hagl's notation above, is immediately lost since all games are converted into a more general internal form, making it very difficult to provide these operations to users. This is an indication of bias in Hagl—the structure inherent in some game representations is lost by converting to another. Instead, we would like all game representations to be first-class citizens, allowing games of different types to be structured differently.

The second limitation is related to *state-based games*. A state-based game is a game that is most naturally described as a series of transformations over some state. As we've seen, defining strategies for games like the prisoner's dilemma is very easy in Hagl. Likewise for the extensive form games introduced in Section 2.2. Unfortunately, the problem of representational bias affects state-based games as well, making the definition of strategies for these kinds of games extremely difficult.

In Section 2.3 we present a means of supporting state-based games within the constraints of Hagl's original game representation. The match game is a simple example of a state-based game that can be defined in this way. In the match game n matches are placed on a table. Each player takes turns drawing some limited number of matches until the player who takes the last match loses. The function below generates a two-player match game with `n` matches in which each player may choose some number from `ms` matches each turn. For example, `matches 15 [1,2,3]` would generate a match game in which 15 matches are placed on the table, and each move consists of taking away 1, 2, or 3 matches.

```
matches :: Int -> [Int] -> Game Int
matches n ms = takeTurns 2 end moves exec pay n
  where end n _ = n <= 0
        moves n _ = [m | m <- ms, n-m >= 0]
        exec n _ m = n-m
        pay _ 1 = [1,-1]
        pay _ 2 = [-1,1]
```

The `takeTurns` function is described in Section 2.3. Here it is sufficient to understand that the state of the game is the number of matches left on the table and that the game is defined by the series of functions passed to `takeTurns`, which describe how that state is manipulated.

Defining the match game in this way works fairly well. The problem is encountered when we try to write strategies to play this game. In Section 3.2, we show how state-based games are transformed into Hagl's internal, stateless game representation. At the time a strategy is run, the game knows nothing about the state. This means that a strategy for the match game has no way of seeing how many matches remain on the table! In Section 5.2 we provide an alternative implementation of the match game using the improved model developed in Section 4. Using this model, which utilizes type classes to support games with diverse representations, we can write strategies for the match game, an example of which is also given in Section 5.2.

1.2 Outline of Paper

In the following section we provide an interleaved introduction to Hagl and game theory. This summarizes our previous work and also includes more recent additions such as preliminary support for state-based games, improved dimensioned list indexing operations, and support for symmetric games. Additionally, it provides a foundation of concepts and examples which is drawn upon in the rest of

the paper. In Section 3 we discuss the bias present in our original game representation and the limitations it imposes. In Section 4 we show how we can overcome this bias, generalizing our model to support many distinct domain representations. In Section 5 we utilize the new model to present solutions to the problems posed in Section 3.

2 A Language for Experimental Game Theory

In designing Hagl, we identified four primary domain objects that must be represented: games, strategies, players, and simulations/experiments. Since this paper is primarily concerned with game representations, this section is correspondingly heavily skewed towards that subset of Hagl. Sections 2.1, 2.2, and 2.3 all focus on the definition of different types of games in Hagl, while Section 2.4 provides an extremely brief introduction to other relevant aspects of the language. For a more thorough presentation of the rest of the language, please see our earlier work in [7].

Game theorists utilize many different representations for different types of games. Hagl attempts to tap into this domain knowledge by providing smart constructors and operators which mimic existing domain representations as closely as possible given the constraints of Haskell syntax. The next two subsections focus on these functions, while Section 2.3 introduces preliminary support for state-based games. Each of these subsections focuses almost entirely on concrete syntax, the interface provided to users of the system. The resulting type and internal representation of these games, `Game mv`, is left undefined throughout this section but will be defined and discussed in depth in Section 3.

2.1 Normal Form Game Definition

One common representation used in game theory is *normal form*. Games in normal form are represented as a matrix of payoffs indexed by each player's move. Fig. 1 shows two related games which are typically represented in normal form. The first value in each cell is the payoff for the player indexing the rows of the matrix, while the second value is the payoff for the player indexing the columns.

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

(a) Prisoner's dilemma

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

(b) Stag hunt

Fig. 1. Normal form representation of two related games

The first game in Fig. 1 is the prisoner's dilemma, which was briefly introduced in Section 1; the other is called the *stag hunt*. These games will be referred to throughout the paper. In both games, each of two players can choose to either cooperate (**C**) or defect (**D**). While these games are interesting from a theoretical perspective and can be used to represent many real-world strategic situations, they each (as with many games in game theory) have a canonical story associated with them from which they derive their names.

In the prisoner's dilemma the two players represent prisoners suspected of collaborating on a crime. The players are interrogated separately, and each can choose to cooperate with his criminal partner by sticking to their fabricated story (not to be confused with cooperating with the interrogator, who is not a player in the game), or they can choose to defect by telling the interrogator everything. If both players cooperate they will be convicted of only minor crimes—a pretty good outcome considering their predicament, and worth two points to each player as indicated by the payoff matrix. If both players defect, they will be convicted of more significant crimes, represented by scoring only one point each. Finally, if one player defects and the other cooperates, the defecting player will be pardoned, while the player who sticks to the fabricated story will be convicted of the more significant crime in addition to being penalized for lying to the investigators. This outcome is represented by a payoff of $3, 0$, getting pardoned having the best payoff, and being betrayed leading to the worst.

In the stag hunt, the players represent hunter-gathers. Each player can choose to either cooperate in the stag hunt, or defect by spending their time gathering food instead. Mutual cooperation leads to the best payoff for each player, as the players successfully hunt a stag and split the food. This is represented in the payoff matrix by awarding three points to each player. Alternatively, if the first player cooperates but the second defects, a payoff of $0, 2$ is awarded, indicating zero points for the first player, who was unsuccessful in the hunt, and two points for the second player, who gathered some food, but not as much as could have been acquired through a hunt. Finally, if both choose to gather food, they will have to split the food that is available to gather, earning a single point each.

Hagl provides the following smart constructor for defining normal form games.

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

This function takes the number of players the game supports, a list of possible moves for each player, a matrix of payoffs, and returns a game. For example, the stag hunt could be defined as follows.

```
stag = normal 2 [[C,D],[C,D]] [[3,3],[0,2],
                               [2,0],[1,1]]
```

The syntax of this definition is intended to resemble the normal form notation in Fig. 1. By leveraging game theorists' existing familiarity with domain representations, we hope to make Hagl accessible to those within the domain.

Since two-player games are especially common amongst normal form games, game theorists have special terms for describing different classes of two-player,

normal form games. The most general of which is a *bimatrix* game, which is simply an arbitrary two player, normal form game.

```
bimatrix = normal 2
```

Somewhat more specialized are *matrix* games and *symmetric* games. A matrix game is a two-player, zero-sum game. In these games, whenever one player wins, the other player loses a corresponding amount. Therefore, the payoff matrix for the `matrix` function is a simple list of floats, the payoffs for the first player, from which the payoffs for the second player can be derived.

```
matrix :: [mv] -> [mv] -> [Float] -> Game mv
```

The traditional game rock-paper-scissors, defined below, is an example of such a game. When one player wins (scoring +1) the other loses (scoring −1), or the two players can tie (each scoring 0) by playing the same move.

```
data RPS = R | P | S
rps = matrix [R,P,S] [R,P,S] [0,-1, 1,
                              1, 0,-1,
                             -1, 1, 0]
```

Symmetric games are similar in that the payoffs for the second player can be automatically derived from the payoffs for the first player. In a symmetric game, the game appears identical from the perspective of both players; each player has the same available moves and the payoff awarded to a player depends only on the combination of their move and their opponent's, not on whether they are playing along the rows or columns of the grid.

```
symmetric :: [mv] -> [Float] -> Game mv
```

Note that all of the games discussed so far have been symmetric. Below is an alternative, more concise definition of the stag hunt, using this new smart constructor.

```
stag = symmetric [C,D] [3, 0, 2, 1]
```

Recall that the definition of the prisoner's dilemma given in Section 1 utilized this function as well.

Although it is theoretically possible to represent any game in normal form, it is best suited for games in which each player plays simultaneously from a finite list of moves. For different, more complicated games, game theory relies on other representations. One of the most common and general of which is *extensive form*, also known as decision or game trees.

2.2 Extensive Form Game Definition

Extensive form games are represented in Hagl by the following data type, which makes use of the preceding set of type synonyms.

```
type PlayerIx = Int
type Payoff = ByPlayer Float
type Edge mv = (mv, GameTree mv)
type Dist a = [(Int, a)]

data GameTree mv = Decision PlayerIx [Edge mv]
                 | Chance (Dist (Edge mv))
                 | Payoff Payoff
```

`Payoff` nodes are the leaves of a game tree, containing the score awarded to each player for a particular outcome. The type of a payoff node's value, `ByPlayer Float`, will be explained in Section 2.4, but for now we'll consider it simply a type synonym for a list of `Float` values.

Internal nodes are either `Decision` or `Chance` nodes. `Decision` nodes represent locations in the game tree where a player must choose to make one of several moves. Each move corresponds to an `Edge` in the game tree, a mapping from a move to its resulting subtree. The player making the decision is indicated by a `PlayerIx`. The following example presents the first (and only) player of a game with a simple decision; if he chooses move `A`, he will receive zero points, but if he chooses move `B`, he will receive five points.

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

Finally, `Chance` nodes represent points where an external random force pushes the game along some path or another based on a distribution. Currently, that distribution is given by a list of edges prefixed with their relative likelihood; e.g. given `[(1,a),(3,b)]`, the edge `b` is three times as likely to be chosen as `a`. However, this definition of `Dist` could easily be replaced by a more sophisticated representation of probabilistic outcomes, such as that presented in [8]. A random die roll, while not technically a game from a game theoretic perspective, 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, (n, Payoff [n])) | n <- [1..6]]
```

The function `extensive`, which has type `GameTree mv -> Game mv` is used to turn game trees into games which can be run in Hagl. Hagl also provides operators for incrementally building game trees, but those are not presented here.

2.3 State-Based Game Definition

One of the primary contributions of this work is the addition of support for state-based games in Hagl. While the normal and extensive forms are general in the sense that any game can be translated into either representation, games which can be most naturally defined as transitions between states seem to form a much broader and more diverse class. From auctions and bargaining games, where the state is the highest bid, to graph games, where the state is the location of players and elements in the graph, to board games like chess and tic-tac-toe,

where the state is the configuration of the board, many games seem to have natural notions of state and well-defined transitions between them.

The preliminary support provided for state-based games described here has been made obsolete by the redesign described in Section 4, but presenting our initial design is valuable as a point of reference for future discussion.

As with other game representations, support for state-based games is provided by a smart constructor, whose type is given below.

```
stateGame :: Int -> (s -> PlayerIx) -> (s -> PlayerIx -> Bool) ->
             (s -> PlayerIx -> [mv]) -> (s -> PlayerIx -> mv -> s) ->
             (s -> PlayerIx -> Payoff) -> s -> Game mv
```

The type of the state is determined by the type parameter `s`. The first argument to `stateGame` indicates the number of players the game supports, and the final argument provides an initial state. The functional arguments in between describe how players interact with the state over the course of the game. Each of these takes the current state and (except for the first) the current player, and returns some information about the game. In order, the functional arguments define:

- which player's turn it is,
- whether or not the game is over,
- the available moves for a particular player,
- the state resulting from executing a move on the given state, and
- the payoffs for some final state.

The result type of this function is the same `Game mv` type as the other games described in previous sections. Notice that `s`, the type of the state, is not reflected in the type signature of the resulting game. This is significant, reflecting the fact that the notion of state is completely lost in the generated game representation.

Since players alternate in many state-based games, and tracking which player's turn it is can be cumbersome, another smart constructor, `takeTurns`, is provided which manages this aspect of state games automatically. The `takeTurns` function has the same type as the `stateGame` function above minus the second argument, which is used to determine whose turn it is.

```
takeTurns :: Int -> (s -> PlayerIx -> Bool) -> (s -> PlayerIx -> [mv]) ->
             (s -> PlayerIx -> mv -> s) -> (s -> PlayerIx -> Payoff) ->
             s -> Game mv
```

The match game presented in Section 1.1 is an example of a state-based game defined using this function.

The definition of players, strategies, and experiments are less directly relevant to the contributions of this paper than the definition of games, but the brief introduction provided in the next subsection is necessary for understanding later examples. The definition of strategies for state-based games, in particular, are one of the primary motivations for supporting multiple domain representations.

2.4 Game Execution and Strategy Representation

All games in Hagl are inherently *iterated*. In game theory, the iterated form of a game is just the original game played repeatedly, with the payoffs of each

iteration accumulating. When playing iterated games, strategies often rely on the events of previous iterations. Tit for Tat, presented in Section 1.1, is one such example.

Strategies are built from a library of functions and combinators which, when assembled, resemble English sentences written from the perspective of the player playing the strategy (i.e. in Tit for Tat, `his` corresponds to the other player in a two player game). Understanding these combinators requires knowing a bit about how games are executed.

In Hagl, all games are executed within the following monad.

```
data ExecM mv a = ExecM (StateT (Exec mv) IO a)
```

This data type wraps a state transformer monad, and is itself an instance of the standard `Monad`, `MonadState` and `MonadIO` type classes, simply deferring to the monad it wraps in all cases. The inner `StateT` monad transforms the `IO` monad, which is needed for printing output and obtaining random numbers (e.g. for `Chance` nodes in extensive form games).

The state of the `ExecM` monad, a value of type `Exec mv`, contains all of the runtime information needed to play the game, including the game itself and the players of the game, as well as a complete history of all previous iterations. The exact representation of this data type is not given here, but the information is made available in various formats through a set of *accessor* functions, some of which will be shown shortly.

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 type `Name` is simply a synonym for `String`. More interestingly, notice that the `s` type parameter is existentially quantified, allowing players with different state types to be stored and manipulated generically within the `ExecM` monad.

The definition of the `Strategy` type introduces one more monadic layer to Hagl. The `StratM` monad adds an additional layer of state management, allowing players to store and access their own personal state of type `s`.

```
data StratM mv s a = StratM (StateT s (ExecM mv) a)
```

The type `Strategy mv s` found in the definition of the `Player` data type, is simply a type synonym for `StratM mv s mv`, an execution in the `StratM` monad which returns a value of type `mv`, the move to be played.

Since many strategies do not require additional state, the following smart constructor is provided which circumvents this aspect of player definition.

```
plays :: Name -> Strategy mv () -> Player mv
plays n s = Player n () s
```

This function reads particularly well when used as an infix operator, as in the definition of Tit for Tat.

Hagl provides smart constructors for defining simple strategies in game theory, such as the pure and mixed strategies discussed in Section 3.1, but for defining more complicated strategies we return to the so-called accessor functions mentioned above. The accessors extract data from the execution state and transform it into some convenient form. Since they are accessing the state of the `ExecM` monad, we would expect their types to have the form `ExecM mv a`, where `a` is the type of the returned information. Instead, they have types of the form `GameM m mv => m a`, where `GameM` is a type class instantiated by both `ExecM` and `StratM`. This allows the accessors to be called from within strategies without needing to be "lifted" into the context of a `StratM` monad.

A few sample accessors are listed below, with a brief description of the information they return. The return types of each of these rely on two data types, `ByGame a` and `ByPlayer a`. Each is just a wrapper for a list of `a`s, but are used to indicate how that list is indexed. A `ByPlayer` list indicates that each element corresponds to a particular player, while a `ByGame` list indicates that each element corresponds to a particular game iteration.

- `move :: GameM m mv => m (ByGame (ByPlayer mv))`
  A doubly nested list of the last move played by each player in each game.
- `payoff :: GameM m mv => m (ByGame Payoff)`
  A doubly nested list of the payoff received by each player in each game.
- `score :: GameM m mv => m (ByPlayer Float)`
  The current cumulative scores, indexed by player.

The benefits of the `ByGame` and `ByPlayer` types become apparent when we start thinking about how to process these lists. First, they help us keep indexing straight. The two different types of lists are organized differently, and the two indexing functions, `forGame` and `forPlayer`, perform all of the appropriate conversions and abstract the complexities away. Second, the data types provide additional type safety by ensuring that we never index into a `ByPlayer` list thinking that it is `ByGame`, or vice versa. This is especially valuable when we consider our other class of combinators, called *selectors*, which are used to process the information returned by the accessors.

While the accessors above provide data, the selector functions constrain data. Two features distinguish Hagl selectors from generic list operators. First, they provide the increased type safety already mentioned. Second, they utilize information from the current execution state to make different selections depending on the context in which they are run. Each `ByPlayer` selector corresponds to a first-person, possessive pronoun in an effort to maximize readability. An example is the `his` selector used in the definition of Tit for Tat above.

```
his :: GameM m mv => m (ByPlayer a) -> m a
```

This function takes a monadic computation that returns a `ByPlayer` list and produces a computation which returns only the element corresponding to the opposing player (in a two player game). Other selectors include `her`, a synonym for `his`, `my`, which returns the element corresponding to the current player, and `our`, which selects the elements corresponds to all players.

Similar selectors exist for processing `ByGame` lists, except these have names like `prev`, for selecting the element corresponding to the previous iteration, and `every`, for selecting the elements corresponding to every iteration.

While this method of defining strategies with a library of accessor and selector functions and other combinators has proven to be very general and extensible, we have run into problems with our much more rigid game representation.

3 A Biased Domain Representation

In Sections 2.1, 2.2, and 2.3 we introduced a suite of functions and operators for defining games in Hagl. However, we left the subsequent type of games, `Game mv`, cryptically undefined. It turns out that this data type is nothing more than an extensive form `GameTree` value, plus a little extra information.

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

Thus, all games in Hagl are translated into extensive form. This is nice since it provides a relatively simple and general internal representation for Hagl to work with. As we'll see, however, it also introduces a substantial amount of *representational bias*, limiting what we can do with games later.

In addition to the game tree, a Hagl `Game` value contains an `Int` indicating the number of players that play the game, and a function from nodes in the game tree to *information groups*. In game theory, information groups are an extension to the simple model of extensive form games. The need for this extension can be seen by considering the translation of a simple normal form game like the stag hunt into extensive form. A straightforward translation would result in something like the following.

```
Decision 1 [(C, Decision 2 [(C, Payoff (ByPlayer [3,3])),
                            (D, Payoff (ByPlayer [0,2]))]),
            [(D, Decision 2 [(C, Payoff (ByPlayer [2,0])),
                            (D, Payoff (ByPlayer [1,1]))])]
```

Player 1 is presented with a decision at the root of the tree; each move leads to a decision by player 2, and player 2's move leads to the corresponding payoff, determined by the combination of both players' moves.

The problem is that we've taken two implicitly simultaneous decisions and sequentialized them in the game tree. If player 2 examines her options, the reachable payoffs will reveal player 1's move. Information groups provide a solution by associating with each other a set of decision nodes for the same player, from within which a player knows only the group she is in, not the specific node.

An information group of size one implies *perfect information*, while an information group of size greater than one implies *imperfect information*. In Hagl, information groups are represented straightforwardly.

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

Therefore, the definition of a new game type in Hagl must not only be translated into a Hagl `GameTree` but must also provide a function for returning the information group corresponding to each `Decision` node. As we can see from the definition of the smart constructor for normal form games introduced earlier, this process is non-trivial.

```
normal :: Int -> [[mv]] -> [[Float]] -> Game mv
normal np mss vs = Game np group (head (level 1))
  where level n | n > np = [Payoff (ByPlayer v) | v <- vs]
                | otherwise = let ms = mss !! (n-1)
                                  bs = chunk (length ms) (level (n+1))
                              in map (Decision n . zip ms) bs
        group (Decision n _) = Imperfect (level n)
        group t = Perfect t
```

Herein lies the first drawback of this approach—defining new kinds of games is difficult due to the often complicated translation process. Until this point in Hagl's history, we have considered the definition of new kinds of games to be part of our role as language designers. This view is limiting for Hagl's use, however, given the incredible diversity of games in game theory, and that the design of new games is often important to a game theorist's research.

A more fundamental drawback of converting everything into a decision tree, however, is that it introduces representational bias. By converting from one representation to another we at best obscure, and at worst completely lose important information inherent in the original representation. The next two subsections demonstrate both ends of this spectrum.

3.1 Solutions of Normal Form Games

As mentioned in the introduction, classical game theory is often concerned with computing optimal strategies for playing games. There are many different definitions of optimality, but two particularly important definitions involve computing *Nash equilibria* and finding *Pareto optimal solutions* [1].

A Nash equilibrium is defined as a set of strategies for each player, where each player, knowing the others' strategies, would have nothing to gain by unilaterally changing his or her own strategy. More colloquially, Nash equilibria describe stable combinations of strategies—even if a player knows the strategies of the other players, he or she would still not be willing to change.

Nash equilibria for normal form games can be *pure* or *mixed*. A pure equilibrium is one where each player plays a specific move. In a mixed equilibrium, players may play moves based on some probability. A good example of a game with an intuitive mixed equilibrium is rock-paper-scissors. If both players randomly play each move with equal probability, no player stands to gain by changing strategies. Going back to our examples from Section 2.1, the stag hunt has two pure Nash equilibria, (**C**, **C**) and (**D**, **D**). If both players are cooperating, they are each earning 3 points; a player switching to defection would earn only 2 points. Similarly, if both players are defecting, they are each earning 1 point;

switching to cooperation would cause the switching player to earn 0 points. The prisoner's dilemma has only one Nash equilibrium, however, which is (**D**, **D**). In this case, mutual cooperation is not stable because a player unilaterally switching to defection will earn 3 points instead of 2.

While the focus of Nash equilibria are on ensuring stability, Pareto optimality is concerned with maximizing the benefit to as many players as possible. A *Pareto improvement* is a change from one solution (i.e. set of strategies) to another that causes at least one player's payoff to increase, while causing no players' payoffs to decrease. A solution from which no Pareto improvement is possible is said to be Pareto optimal. The only Pareto optimal solution of the stag hunt is mutual cooperation, since any other solution can be improved by switching to pure cooperation. In the prisoner's dilemma (**C**, **D**) and (**D**, **C**) are also both Pareto optimal, since any change would cause the defecting player's payoff to decrease.

Nash equilibria and Pareto optimal solutions are guaranteed to exist for every game, but they are hard to compute, in general. Finding a Nash equilibrium for an arbitrary game is known to be PPAD-complete (computationally intractable) [9], and even the much more constrained problem of deciding whether a game has a pure Nash equilibrium is NP-hard [10]. There do exist, however, simpler algorithms for certain variations of these problems on highly constrained games. Finding pure Nash equilibria and Pareto optimal solutions on the kinds of games typically represented in normal form is comparatively easy [11].

While Hagl's primary focus is experimental game theory, we would like to begin providing support for these kinds of fundamental, analytical operations. Unfortunately, the bias in Hagl's game representation makes this very difficult. We would like to add functions for finding pure Nash equilibria and Pareto optimal solutions, and have them only apply to simple normal form games, but this is impossible since all games have the same type. Additionally, although the structure of a game's payoff matrix is available at definition time, it is instantly lost in the translation to a decision tree.

While this subsection demonstrated how representational bias can obscure the original structure of a game, the next provides an example where the structure is completely and irrecoverably lost.

3.2 Loss of State in State-Based Games

Section 2.3 introduced preliminary support for games defined as transformations of some state. The smart constructor `stateGame` takes an initial state and a series of functions describing how to manipulate that state, and produces a standard Hagl game tree representation. Below is the implementation of this function.

```
stateGame np who end moves exec pay init = Game np Perfect (tree init)
  where tree s | end s p   = Payoff (pay s p)
               | otherwise = Decision p [(m, tree (exec s p m))
                                         | m <- moves s p]
          where p = who s
```

The most interesting piece here is the `tree` function, which threads the state through the provided functions to generate a standard, stateless decision tree.

Haskell's laziness makes this viable for even the very large decision trees often generated by state-based games. But there is still a fundamental problem with this solution: we've lost the state.

At first this way of folding the state away into a game tree was very appealing. The solution is reasonably elegant, and seemed to emphasize that even the most complex games could be represented in our general form. The problem is first obvious when one goes to write a strategy for a state-based game. Because the state is folded away, it is forever lost immediately upon game definition; strategies do not and cannot have access to the state during game execution. Imagine trying to write a strategy for a chess player without being able to see the board!

Of course, there is a workaround. Since players are afforded their own personal state, as described in Section 2.4, each player of a state-based game could personally maintain his or her own copy of the game state, modifying it as other players make their moves. In addition to being wildly inefficient, this makes defining strategies for state-based games much more difficult, especially considering strategies may have their own additional states to maintain as well.

This limitation of strategies for state-based games is perhaps the most glaring example of representational bias. The translation from a state-based representation to a stateless extensive form representation renders the game nearly unusable in practice.

4 Generalizing Game Representations

In setting about revising Hagl's game representation, we established four primary goals, listed here roughly in order of significance, from highest to lowest.

1. Better accommodate state in games. Many kinds of games are very naturally represented as transitions between states. The inability of strategies to directly access these states, as described in Section 3.2, was the most egregious instance of representational bias in the language.
2. Lessen representational bias in general by allowing for multiple different game representations. Although dealing with the state issue was a pressing need, we sought a more general solution to the underlying problem. This would also help us overcome the problems described in Section 3.1.
3. Lower the barrier to entry for defining new classes of games. Having one general game representation is nice, but requiring a translation from one representation to another is difficult, likely preventing users of the language from defining their own game types.
4. Minimize impact on the rest of the language. In particular, high-level game and strategy definitions should continue to work, with little to no change.

In the rest of this section we will present a design which attempts to realize these goals, touching briefly on the motivations behind important design decisions.

From the goals stated above, and the second goal in particular, it seems clear that we need to introduce a type class for representing games. Recall the game

representation given in Section 3: A game is represented by the `Game` data type which contains the number of players that play the game, an explicit game tree as a value of the `GameTree` data type, and a function for getting the information group associated with a particular node. One approach would be to simply redefine the `Game` data type as a type class as shown below.

```
class Game g where
  type Move g
  numPlayers :: g -> Int
  gameTree   :: g -> GameTree (Move g)
  info       :: g -> GameTree (Move g) -> Info (Move g)
```

Note that this definition uses associated types [12], an extension to Haskell 98 [13] available in GHC [14], which allow us to use type classes to overload types in the same way that we overload functions. `Move g` indicates the move type associated with the game type `g` and is specified in class instances, as we'll see later.

In effect, the use of a type class delays the translation process, maintaining diverse game representations and converting them into game trees immediately before execution. However, this solution still does not support state-based games, nor does it make defining new types of games any easier since we must still provide a translation to a game tree. Enabling state-based games would be fairly straightforward; we could simply add a state value to each node in the game tree, making it trivial to retain the state that was previously folded away, and to provide easy access to it by strategies. Making game definition easier, on the other hand, requires a more radical departure from our original design.

4.1 Final Game Representation

Ultimately, we decided to abandon the game tree representation altogether and define games as arbitrary computations within the game execution monad. The final definition of the `Game` type class is given below.

```
class Game g where
  type Move g
  type State g
  initState :: g -> State g
  runGame   :: ExecM g Payoff
```

Note that we provide explicit support for state-based games by adding a new associated type `State` and a function `initState` for getting the initial state of the game. This state is then stored with the rest of the game execution state and can be accessed and modified from within the monadic `runGame` function, which describes the execution of the game.

Also note that we have removed the `numPlayers` method from the previous type class. This change eliminates another small source of bias, reflecting the fact that many games support a variable number of players, violating the functional relationship between a game and a constant number of players that was

previously implied. Game instances should now handle this aspect of game definition on their own, as needed.

To ease the definition of new types of games, we also provide a library of game definition combinators. These combinators provide a high-level interface for describing the execution of games, while hiding all of the considerable minutiae of game execution briefly discussed in Section 2.4. Some of these combinators will be introduced in the next subsection, but first we discuss some of the trade-offs involved in this design change.

Perhaps the biggest risk in moving from an *explicit* representation (game trees) to an *implicit* representation (monadic computations) is the possibility of overgeneralization. While abstraction helps in removing bias from a representation, one has to be careful not to abstract so far away from the domain that it is no longer relevant. After all, we initially chose game trees since they were a very general representation of *games*. Does a monadic computation really capture what it means to be a game?

Another subtle drawback is that, with the loss of an explicit game representation, it is no longer possible to write some generic functions which relied on this explicitness. For example, we can no longer provide a function which returns the available moves for an arbitrary game. Such functions would have to be provided per game type, as needed. For games where an explicit representation is more suitable, another type class is provided, similar to the initial type class suggested in Section 4 that defines a game in terms of a game tree, as before.

These risks and minor inconveniences are outweighed by many substantial benefits. First and foremost, the monadic representation is much more flexible and extensible than the tree-based approach. Because games were previously defined in terms of a rigid data type, they were limited to only three types of basic actions, corresponding to the three constructors in the `GameTree` data type. Adding a fundamentally new game construct required modifying this data type, a substantial undertaking involving the modification of lots of existing code. Adding a new construct in the new design, however, is as easy as adding a new function.

Another benefit of the new design is that games can now vary depending on their execution context. One could imagine a game which changes during execution to handicap the player with highest score, or a game where the payoff of a certain outcome depends on past events, perhaps to simulate diminishing returns. These types of games were not possible before. Since games now define their execution directly and have complete access to the execution context through the `ExecM` monad, we can define games which change as they play.

Finally, as the examples given in Section 5 demonstrate, defining game execution in terms of the provided combinators is substantially easier than translating game representations into extensive form. This lowers the barrier to entry for game definition in Hagl, making it much more reasonable to expect users of the system to be able to define their own game types as needed. In the next subsection we provide an introduction to some of the combinators in the game definition library.

4.2 Game Definition Combinators

Game execution involves a lot of bookkeeping. Various data structures are maintained to keep track of past moves and payoffs, the game's current state, which players are involved, etc. The combinators introduced in this section abstract away all of these details, providing high-level building blocks for describing how games are executed.

The first combinator is one of the most basic and will be used in most game definitions. This function executes a decision by the indicated player.

```
decide :: Game g => PlayerIx -> ExecM g (Move g)
```

This function retrieves the indicated player, executes his or her strategy, updates the historical information in the execution state accordingly, and returns the move that was played.

The `allPlayers` combinator is used to carry out some action for all players simultaneously, returning a list of the accumulated results.

```
allPlayers :: Game g => (PlayerIx -> ExecM g a) -> ExecM g (ByPlayer a)
```

Combining these first two combinators as `allPlayers decide`, we define a computation in which all players make a simultaneous (from the perspective of the players) decision. This is used, for example, in the definition of normal form games in Section 5.1.

While the `allPlayers` combinator is used for carrying out simultaneous actions, the `takeTurns` combinator is used to perform sequential actions. It cycles through the list of players, executing the provided computation for each player, until the terminating condition (second argument) is reached.

```
takeTurns :: Game g => (PlayerIx -> ExecM g a) -> ExecM g Bool -> ExecM g a
```

This combinator is used in the revised definition of the match game given in Section 5.2.

There are also a small number of functions for constructing common payoff values. One example is the function `winner`, whose type is given below.

```
winner :: Int -> PlayerIx -> Payoff
```

When applied as `winner n p`, this function constructs a payoff value (recall that `Payoff` is a synonym for `ByPlayer Float`) where the winning player `p` receives 1 point, and all other players, out of `n`, receive -1 point. There is a similar function `loser`, which gives a single player -1 point and all other players 1 point, and a function `tie` which takes a single `Int` and awards all players 0 points.

5 Flexible Representation of Games

In this section we demonstrate the improved flexibility of the game representation by solving some of the problems posed earlier in the paper. Section 5.1 demonstrates the new representation of normal form games and provides functions for computing analytical solutions. Section 5.2 provides a new implementation of the match game and a strategy that can access its state in the course of play.

5.1 Representing and Solving Normal Form Games

The fundamental shortcoming of the original game representation with regard to normal form games was that the structure of the games was lost. Now, we can capture this structure explicitly in the following data type.

```
data Normal mv = Normal Int (ByPlayer [mv]) [Payoff]
```

Note that the arguments to this data constructor are almost exactly identical to the normal form smart constructor introduced in Section 2.1. A second data type resembles another smart constructor from the same section, for constructing two-player, zero-sum games.

```
data Matrix mv = Matrix [mv] [mv] [Float]
```

Even though these games are closely related, we represent them with separate data types because some algorithms, such as the one for computing saddle point equilibria described below, apply only to the more constrained matrix games. To tie these two data types together, we introduce a type class that represents all normal form games, which both data types implement.

```
class Game g => Norm g where
  numPlayers :: g -> Int
  payoffFor  :: g -> Profile (Move g) -> Payoff
  moves      :: g -> PlayerIx -> [Move g]
```

This type class allows us to write most of our functions for normal form games in a way that applies to both data types. The `Profile` type here refers to a *strategy profile*, a list of moves corresponding to each player. This is reflected in the definition of `Profile mv`, which is just a type synonym for `ByPlayer mv`. The `payoffFor` method returns the payoff associated with a particular strategy profile by looking it up in the payoff matrix. The `moves` method returns the moves available to a particular player.

Using the type class defined above, and the combinators from Section 4.2, we can define the execution of normal form games as follows.

```
runNormal :: Norm g => ExecM g Payoff
runNormal = do g <- game
               ms <- allPlayers decide
               return (g `payoffFor` ms)
```

Contrast this definition with the smart constructor in Section 3 that translated normal form to extensive form. We think that this dichotomy is extremely compelling motivation for the generalized representation. Defining new types of games is now a much simpler process.

Finally, in order to use normal form games within Hagl, we must instantiate the `Game` type class for both data types. This is very straightforward using the `runNormal` function above, and we show only one instance here since the other is nearly identical.

```
instance Eq mv => Game (Normal mv) where
  type Move (Normal mv) = mv
  type State (Normal mv) = ()
  initState _ = ()
  runGame = runNormal
```

Since normal form games do not require state, the associated state type of is just the unit type `()`. Similarly, for any normal form game, the initial state value is the unit value.

Now we can run experiments on, and write strategies for, normal form games just as before. And with the addition of some very straightforward smart constructors, we can directly reuse our earlier definitions of the prisoner's dilemma and stag hunt. We can also, however, write the analytical functions which we previously could not. Below are type definitions for functions which return the pure Nash equilibria and Pareto optimal solutions of normal form games.

```
nash   :: (Norm g, Eq (Move g)) => g -> [Profile (Move g)]
pareto :: (Norm g, Eq (Move g)) => g -> [Profile (Move g)]
```

Using these we can define a function to find Pareto-Nash equilibria, solutions which are both Pareto optimal and a Nash equilibrium, and which represent especially desirable strategies to play [15].

```
paretoNash g = pareto g `intersect` nash g
```

Applying this new analytical function to the stag and hunt and prisoner's dilemma provides some interesting results.

```
> paretoNash stag
[ByPlayer [C,C]]
> paretoNash pd
[]
```

This likely demonstrates why prisoner's dilemma is a far more widely studied game than the stag hunt. The stag hunt is essentially solved—a strategy which is both Pareto optimal and a Nash equilibria is a truly dominant strategy. Compare this to the following analysis of the prisoner's dilemma.

```
> nash pd
[ByPlayer [D,D]]
> pareto pd
[ByPlayer [C,C],ByPlayer [C,D],ByPlayer [D,C]]
```

In the prisoner's dilemma, the only Nash equilibrium is not Pareto optimal, while all other solutions are. This makes for a much more subtle and complex game.

Saddle points represent yet another type of equilibrium solution, but one that only applies to matrix games. Essentially, a saddle point of a matrix game is strategy profile which corresponds to a value which is both the smallest value in its row and the largest value in its column [11]. Such a value is ideal for both players, and thus we would expect two rational players to always play a strategy corresponding to a saddle point, if one exists. Hagl provides the following function for finding saddle points in matrix games.

```
saddle :: Eq mv => Matrix mv -> [Profile mv]
```

A key feature of this function is that the type system prevents us from applying it to a normal form game which is not of the right form. Being able to better utilize the type system is another advantage of the more flexible representation enabled by type classes.

5.2 Representing and Playing the Match Game

In Section 1.1 we presented a definition of the match game only to discover that we could not write a strategy for it. In this subsection we present a redefinition of the game with the improved representation, and an unbeatable strategy for playing it.

First, we create a data type to represent an instance of the match game. As with our normal form game definition in Section 5.1, this data type resembles the original smart constructor for building match games.

```
data Matches = Matches Int [Int]
```

Here, the first argument indicates the number of matches to set on the table, while the second argument defines the moves available to the players (i.e. the number of matches that can be drawn on a turn).

Instantiating the `Game` type class for the match game is mostly straightforward, with only the implementation of `runGame` being non-trivial.

```
instance Game Matches where
  type Move Matches = Int
  type State Matches = Int
  initState (Matches n _) = n
  runGame = ...
```

The associated `Move` type of the match game is `Int`, the number of matches to draw; the `State` type is also `Int`, the number of matches remaining on the table; and the initial state of the match game is just extracted from the data type. To define the execution of the match game, we build up a series of helper functions.

First we define two simple functions which make manipulating the state of the match game a little nicer.

```
matches = gameState
draw n = updateGameState (subtract n)
```

Both the `gameState` and `updateGameState` functions are part of the game definition combinator library. `gameState` is used to return the current game state, while `updateGameState` updates the current game state by applying a provided function. These functions have the following types.

```
gameState :: (Game g, GameM m g) => m (State g)
updateGameState :: Game g => (State g -> State g) -> ExecM g (State g)
```

Note that, the type of `gameState` is somewhat more general than the type of `updateGameState`. The type variable `m` in the type of `gameState` ranges over the monadic type constructors `ExecM` and `StratM`, which are both instances of `GameM`, while `updateGameState` applies only to computations in the `ExecM` monad. This means that access to the game's state is available from within both game definitions (`ExecM`) and strategy definitions (`StratM`), whereas modifying the game state is only possible from within game definitions—strategies may only manipulate the state indirectly, by playing moves.

Thus, the `matches` function returns the current number of matches on the table, while the `draw` function updates the state by removing the given number of matches from the table.

From here we can define a computation which determines when the game is over, that is, when there are no matches remaining on the table.

```
end = do n <- matches
         return (n <= 0)
```

And a function which executes a turn for a given player.

```
turn p = decide p >>= draw >> return p
```

On a player's turn, the player makes a decision and the move indicates how many matches to draw from the table. The reason this function returns the player's index will be seen in a moment. First, we define a computation which returns the payoff for the game, given the player who draws the last match.

```
payoff p = do n <- numPlayers
              return (loser n p)
```

The player who draws the last match loses, earning -1 point, while other players win 1 point.

We now have everything we need to define the execution of the match game. Recall the `takeTurns` function introduced in Section 4.2 which takes two arguments, the first of which represents a player's turn, and the second is a function indicating when to stop looping through the players, applying the turn functions. We have just shown the definition of both of these functions for the match game, `turn` and `end`.

```
instance Game Matches where
  ...
  runGame = takeTurns turn end >>= payoff
```

The result of the `takeTurns` function is the value produced on the last player's turn, in this case, the player's index, which is passed to the payoff function indicating that the player taking the last match lost.

With the match game now defined, and with a representation that allows easy access to the state of state-based games, we can look towards defining strategies. First, we define two helper functions. The first returns the moves that are available to player.

```
moves = do n <- matches
           (Matches _ ms) <- game
           return [m | m <- ms, n-m >= 0]
```

This function extracts the available moves from the game representation, and filters out the moves that would result in a negative number of matches. Second, we define a function which returns a move randomly.

```
randomly = moves >>= randomlyFrom
```

The `randomlyFrom` function is part of Hagl's strategy combinator library, and returns a random element from a list.

Finally, a strategy for playing the match game is given below. The two-player match game is solvable for the first player, which means that the first player to move can always win if he or she plays correctly. The following player demonstrates one such solution.

```
matchy = "Matchy" `plays`
    do n  <- matches
       ms <- moves
       let winning m = mod (n-1) (maximum ms + 1) == m
        in maybe randomly return (find winning ms)
```

The `winning` function, defined in this strategy, takes a move and returns true if it is a move that leads to an eventual win. The strategy plays a winning move if it can find one (which it always will as the first player), and plays randomly otherwise. While a proof that this player always wins when playing first is beyond the scope of this paper, we can provide support for this argument by demonstrating it in action against another player. The following player simply plays randomly.

```
randy = "Randy" `plays` randomly
```

We now run an experiment with these two players playing the match game against each other. The following command runs the game 1000 times consecutively, then printing the accumulated score.

```
> execGame (Matches 15 [1,2,3]) [matchy,randy] (times 1000 >> printScore)
Score:  Matchy: 1000.0
        Randy: -1000.0
```

This shows that, when playing from the first position, the optimal strategy won every game. Even from the second position Matchy is dominant, since it only takes one poor play by Randy for Matchy to regain the upper hand.

```
> execGame (Matches 15 [1,2,3]) [randy,matchy] (times 1000 >> printScore)
Score:  Randy: -972.0
        Matchy: 972.0
```

These types of simple experiments demonstrate the potential of Hagl both as a simulation tool, and as a platform for exploring and playing with problems in experimental game theory. By providing support for the large class of state-based games, and making it easier for users to define their own types of games, we greatly increase its utility.

6 Conclusions and Future Work

Hagl provides much needed language support to experimental game theory. As we extended the language, however, we discovered many problems related to bias in Hagl's game representation. In this work we fundamentally redefine the concept of a Hagl game in a way that facilitates multiple underlying representations, eliminating this bias. In addition, we provide a combinator library which drastically simplifies the process of defining new classes of games, enabling users of the language to define new games and new game constructs as needed. Finally, we added explicit support for the broad class of games defined as transitions between states, resolving one of the primary weaknesses of earlier versions of the language.

In Section 4.1 we briefly introduced the idea of games which vary depending on their execution context. This represents a subset of a larger class of games which vary depending on their environment. Examples include games where the payoffs change based on things like the weather or a stock market. Since we have access to the `IO` monad from with execution monad, such games would be possible in Hagl. Other games, however, change depending on the *players* that are playing them. Auctions are a common example where, since each player may value the property up for auction differently, the payoff values for winning or losing the game will depend on the players involved. While it would be possible to simply parameterize a game definition with valuation functions corresponding to each player, it would be nice if we could capture this variation in the representation of the players themselves, where it seems to belong. Since auctions are an important part of game theory, finding an elegant solution to this problem represents a potentially useful area for future work.

Another potential extension to Hagl is support for human-controlled players. This would have many possible benefits. First, it would support more direct exploration of games; often the best way to understand a game is to simply play it yourself. Second, it would allow Hagl to be used as a platform for experiments *on* humans. Experimenters could define games which humans would play while the experimenters collect the results. Understanding how people actually play games is an important aspect of experimental game theory, and a significant niche which Hagl could fill.

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 [16]. In future work we hope to utilize ideas from both of these projects to make game theory accessible to a broader audience. One of the current limitations of Hagl, common in DSELs in general, is a presupposition of knowledge of the host language, in this case, Haskell. Our visual language is targeted at a much broader audience, but has a correspondingly smaller scope than Hagl. Somewhere in between there is a sweet spot. One possibility is using Hagl as a back-end for an integrated game theory tool which incorporates our visual language as part of an interface for defining, executing and explaining concepts in game theory.

References

1. Fudenberg, D., Tirole, J.: Game Theory, xvii–xx, pp. 11–23. MIT Press, Cambridge (1991)
2. Nagel, R.: Unraveling in Guessing Games: An Experimental Study. American Economic Review 85, 1313–1326 (1995)
3. Ho, T., Camerer, C., Weigelt, K.: Iterated Dominance and Iterated Best-response in p-Beauty Contests. American Economic Review 88(4), 947–969 (1998)
4. Crawford, V.: Introduction to Experimental Game Theory. Journal of Economic Theory 104(1), 1–15 (2002)
5. Axelrod, R.: The Evolution of Cooperation. Basic Books, New York (1984)
6. Kendall, G., Darwen, P., Yao, X.: The Prisoner's Dilemma Competition (2005), `http://www.prisoners-dilemma.com`
7. Walkingshaw, E., Erwig, M.: A Domain-Specific Language for Experimental Game Theory. Under consideration for publication in the Journal of Functional Programming (2008)
8. Erwig, M., Kollmansberger, S.: Functional Pearls: Probabilistic functional programming in Haskell. Journal of Functional Programming 16(01), 21–34 (2005)
9. Papadimitriou, C.: The Complexity of Finding Nash Equilibria. Algorithmic Game Theory, 29–52 (2007)
10. Gottlob, G., Greco, G., Scarcello, F.: Pure Nash Equilibria: Hard and Easy Games. In: Proceedings of the 9th conference on Theoretical aspects of rationality and knowledge, pp. 215–230. ACM, New York (2003)
11. Straffin, P.: Game Theory and Strategy, pp. 7–12, 65–80. The Mathematical Association of America, Washington (1993)
12. Chakravarty, M., Keller, G., Jones, S., Marlow, S.: Associated types with class. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, vol. 40, pp. 1–13. ACM, New York (2005)
13. Peyton Jones, S.L.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)
14. GHC: The Glasgow Haskell Compiler (2004), `http://haskell.org/ghc`
15. Groves, T., Ledyard, J.: Optimal Allocation of Public Goods: A Solution to the Free Rider Problem. Econometrica 45(4), 783–809 (1977)
16. Erwig, M., Walkingshaw, E.: A Visual Language for Representing and Explaining Strategies in Game Theory. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing, pp. 101–108 (2008)