# AN ABSTRACT OF THE THESIS OF

Eric Walkingshaw for the degree of Master of Science in Computer Science presented on December 20, 2011.

Title: Domain-Specific Language Support for Experimental Game Theory

Abstract approved: _____

Martin Erwig

Experimental game theory is the use of game theoretic abstractions—games, players, and strategies—in experiments and simulations. It is often used in cases where traditional, analytical game theory fails or is difficult to apply. This thesis collects three previously published papers that provide domain-specific language (DSL) support for defining and executing these experiments, and for explaining their results.

Despite the widespread use of software in this field, there is a distinct lack of tool support for common tasks like modeling games and running simulations. Instead, most experiments are created from scratch in general-purpose programming languages. We have addressed this problem with Hagl, a DSL embedded in Haskell that allows the concise, declarative definition of games, strategies, and executable experiments. Hagl raises the level of abstraction for experimental game theory, reducing the effort to conduct experiments and freeing experimenters to focus on hard problems in their domain instead of low-level implementation details.

While analytical game theory is most often used as a prescriptive tool, a way to analyze a situation and determine the best course of action, experimental game theory is often applied descriptively to explain why agents interact and behave in a certain way. Often these interactions are complex and surprising. To support this explanatory role, we have designed visual DSL for explaining the interaction of strategies for iterated games. This language is used as a vehicle to introduce the notational quality of traceability and the new paradigm of explanation-oriented programming.

# Domain-Specific Language Support for Experimental Game Theory

by

Eric Walkingshaw

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 20, 2011
Commencement June 2012

Master of Science thesis of Eric Walkingshaw presented on December 20, 2011.

APPROVED:

---

Major Professor, representing Computer Science

---

Director of the School of Electric Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

---

Eric Walkingshaw, Author

# ACKNOWLEDGEMENTS

Thanks go first to my collaborator and advisor Martin Erwig, whose contributions to my development as a researcher and thinker go far beyond our published work. I am immensely grateful for his confidence in me and consistent enthusiasm.

A heartfelt thank you to Ronnie Macdonald and Laurie Meigs for their kindness and financial support through the ARCS Foundation. Laurie and Ronnie are fantastically warmhearted, funny, and interesting women, and I have loved getting to know them and their families over the last three years. Their generosity motivates me daily.

Finally, thank you to my friend and partner Allison for believing in me, keeping me nourished in the days preceding paper deadlines, and constantly showing me the meaning of hard work by her example.

# CONTRIBUTION OF AUTHORS

Martin Erwig is co-author of all three of the papers contained in this manuscript. This section briefly describes our relative contributions to these papers in order to demonstrate that they satisfy the requirements for a manuscript format thesis.

The design and implementation of the Hagl language featured in Chapters 2 and 3 was done mostly by myself, though Martin provided essential criticism along the way. The two corresponding papers were also written primarily by me. Martin contributed revisions and many suggestions to the written content of these papers.

In contrast, the design of the visual strategy language presented in Chapter 4 paper was highly collaborative. The formal description of the notation is mostly Martin's work. We contributed approximately equally to the written content of the paper and subsequent revisions.

# TABLE OF CONTENTS

Page

1 Introduction 1

2 Hagl: A Haskell DSEL for Experimental Game Theory 4

2.1 Introduction . . . 5

2.2 The Iterated Prisoner's Dilemma . . . 6

2.3 Normal-Form Games . . . 8

2.4 Extensive-Form Games . . . 9

2.5 Defining Strategies . . . 12

2.5.1 Data Accessors . . . 13

2.5.2 List Selectors . . . 14

2.5.3 Initializing Strategies . . . 16

2.6 Game Execution . . . 17

2.7 Player and Strategy Representation . . . 21

2.8 Stateful Strategies . . . 21

2.9 Conclusion . . . 22

3 Extending the Expressiveness of Hagl 24

3.1 Introduction . . . 25

3.1.1 Limitations of the Game Representation . . . 27

3.1.2 Outline of Paper . . . 28

3.2 A Language for Experimental Game Theory . . . 28

3.2.1 Normal Form Game Definition . . . 29

3.2.2 Extensive Form Game Definition . . . 31

3.2.3 State-Based Game Definition . . . 32

3.2.4 Game Execution and Strategy Representation . . . 34

3.3 A Biased Domain Representation . . . 36

3.3.1 Solutions of Normal Form Games . . . 38

3.3.2 Loss of State in State-Based Games . . . 39

3.4 Generalizing Game Representations . . . 40

3.4.1 Final Game Representation . . . 41

3.4.2 Game Definition Combinators . . . 43

3.5 Flexible Representation of Games . . . 44

3.5.1 Representing and Solving Normal Form Games . . . 44

3.5.2 Representing and Playing the Match Game . . . 46

# TABLE OF CONTENTS (Continued)

Page

3.6 Conclusions and Future Work . . . . . 50

4 A Visual Language for Representing and Explaining Strategies 52

4.1 Introduction . . . . . 53

4.2 A Notation for Game Theory . . . . . 55

4.2.1 Strategy Notation . . . . . 55

4.2.2 Representing Traces . . . . . 58

4.2.3 Relating Strategies to Traces . . . . . 60

4.3 Discussion of Language Design . . . . . 61

4.3.1 Design of Move Patterns . . . . . 61

4.3.2 Design of Game Traces . . . . . 63

4.4 Traceability as a Cognitive Dimension . . . . . 63

4.4.1 Interaction with Existing Cognitive Dimensions . . . . . 64

4.4.2 Independence of Trace Role and Integration . . . . . 65

4.4.3 Traceability as a Notation Design Tool . . . . . 66

4.5 Related Work . . . . . 66

4.6 Conclusions and Future Work . . . . . 68

5 Conclusion 69

Bibliography 71

# LIST OF FIGURES


| Figure | | Page |
|---|---|---|
| 2.1 | Normal-form representation of the prisoner's dilemma | 7 |
| 3.1 | Normal form representation of two related games | 29 |
| 4.1 | Prisoner's Dilemma in normal form | 54 |
| 4.2 | Definition of Tit for Tat | 56 |
| 4.3 | Definition of Grim Trigger | 57 |
| 4.4 | Example mixed strategy | 57 |
| 4.5 | Tit for Two Tats strategy | 58 |
| 4.6 | Opponent Alternator strategy | 58 |
| 4.7 | Example game play | 59 |
| 4.8 | Example game trace | 59 |
| 4.9 | Traces showing individual payoffs | 59 |
| 4.10 | Game play explanation | 60 |
| 4.11 | Initial player move design | 61 |

# Chapter 1 – Introduction

In game theory, a *game* is a formal representation of a situation involving multiple agents, called *players*, who interact by making *moves* within the structure of the game. A player chooses which moves to make according to a *strategy* and is awarded a *payoff* at the end of the game that quantifies their success. The goal of each player is to maximize their own individual payoff.

Games are abstract mathematical entities that are usually intended to reflect real-world problems of practical importance, but they are most often presented and discussed in terms of simple, contrived scenarios. As an example (adapted from [24]), consider a game show in which four contestants are each given $100. Each contestant makes a single move in which they secretly and independently choose how much money to allocate to a common pool. Any money not allocated is theirs to keep, while the pool of allocated money is doubled by the show's host and dispersed evenly to all contestants (regardless of how much they allocated). Obviously, each player wants to maximize the amount of money they take home. How should they proceed?

This game is interesting because it induces a very famous and well-studied dilemma known as the "tragedy of the commons" [18]. The dilemma is that it is in the best interest of the players as a group to allocate all of their money to the pool (since total allocation results in the most total money earned), but each player is individually dissuaded from this because each dollar allocated yields only 50 cents in return. The best interests of the group are in direct opposition to the best interests of the individual, but if all players play selfishly, each will be in a worse position than if they had cooperated (taking home $100 each instead of $200). This basic dilemma is reflected in many important world problems like the conservation of natural resources, protection of the environment, and the proliferation of nuclear weapons [18, 24].

Traditionally, game theory has been used as an analytical tool for prescribing the best course of action for an individual player by mathematically determining an optimal strategy. This can be used to predict how a group will play by assuming that all players are rational and so play their optimal strategies. Unfortunately for our players, the analytical approach prescribes and predicts selfish play. Since there is no incentive for any rational player to unilaterally deviate from keeping all of their money (since doing so always results in a decreased individual payoff), all rational players will play this strategy.[1]

However, the expectation of selfish play depends on two important factors. First, it assumes that the game accurately reflects the scenario of interest. In the case of the above world problems,

[1] A stable situation like this is called a *Nash equilibrium* [34].

the game obviously drastically oversimplifies things. Factors like shared information, continuous play, and reputation may offer routes of escape from the tragedy of the commons [24]. These factors can be difficult to quantify in the analytical approach. Second, it assumes that all players are rational. Experimental results have shown that when games like the above are executed literally with real people, this assumption does not hold [20, 24]. Actual humans "irrationally" cooperate, and sometimes achieve better results because of it.

Experimental game theory is a complimentary alternative to analytical game theory in which strategies and players are evaluated not mathematically, but through competitive experimentation. It can be used to help understand and explain strategic behavior in situations where the analytical approach fails or is difficult to apply, and it can be used to help identify seemingly sub-optimal strategies like cooperation that perform well in practice or under certain conditions.

This thesis collects three papers that provide domain-specific language (DSL) support for research using experimental game theory. In many ways, game theory is an ideal experimental platform. Games provide structure with well-defined variation points and payoffs provide a built-in means to quantify the results. The execution of games and strategies can also be automated, making it cheap to empirically analyze the interaction of several different alternative strategies over many iterations [7]. This is a very powerful research tool, as demonstrated by Robert Axelrod's breakthrough behavioral studies on cooperation [1], discussed several times throughout this thesis. Experimental game theory is increasingly used by economists, evolutionary biologists, and behavioral scientists [4].

Despite the reliance on software for this type of research, there is not much tool support for experimental game theory. Most experiments seem to be implemented from scratch in general-purpose languages. Not only does this pose a huge amount of effort, but it forces researchers to realize and manage their experiments at a very low-level of abstraction. In Chapter 2 we address this problem with Hagl, a domain-specific embedded language (DSEL) in Haskell that supports the declarative definition and easy execution of game-theoretic experiments [35].

The language of Hagl can be separated into three main pieces for representing games, strategies, and experiments, each of which can be considered a sort of mini-DSL within the larger language. Games are represented in Hagl in *extensive form*, an explicit tree representation where each internal node represents a player's decision point and each leaf is a set of payoffs [34]. Smart constructors (which produce values of the internal extensive form representation directly) are provided for defining other common types of games, such as *normal form* games, where each player makes a simultaneous move that indexes a grid of potential payoffs, and *state-based games*, where players interact by manipulating some shared state, as in tic-tac-toe. Games are also inherently iterative in Hagl, meaning that players can expect to play several (perhaps hundreds or thousands) games in a row against the same opponents, accumulating payoffs. Almost all research with experimental game theory involves iterated games. Strategies are defined by a flexible com-

binator library with natural language-like support for declaratively accessing and analyzing the history of previous iterations, together with a suite of high-level constructs for defining the most common types of strategies. Experiments are defined in a simple monadic command language.

Chapter 3 presents several extensions to Hagl [36]. On the surface, this work adds implementations of common solution strategies from analytical game theory, which are often useful as control or reference points in experiments, and better support for state-based games. These additions are supported, however, by much more significant changes to the core game representation of Hagl. The design away from the explicit tree representation, adopting instead a more flexible type-class representation. Previously, the immediate translation (via smart constructors) of normal form and state-based games into extensive form lost important structural information essential for analyzing and strategizing about these types of games. The new representation allows normal form and state-based games to be represented and used directly. It also makes it significantly easier to add entirely new game representations to Hagl, expanding the range of potential applications.

Interaction is at the heart of game theory—the effectiveness of a player's strategy often depends critically on the strategies of the other players. As the two papers on Hagl show repeatedly, often a strategy will be very successful against one strategy and very unsuccessful against another, and these interactions can become quite complex. While Hagl can reduce the amount of effort needed to say *which* strategies are the most effective against which opponents, it can often remain difficult to unravel why.

In Chapter 4 we switch gears by presenting a domain-specific visual language for *explaining* the interaction of strategies in experimental game theory [12]. This language is used to introduce the new paradigm of *explanation-oriented programming* (EOP). EOP focuses on producing explanations of a result rather than on attaining the result itself, and promotes explainability as a principal design criterion. This work also introduces the notational quality of *traceability*, which emphasizes the explanatory importance of a clear mapping between program notation and program output. In this case, two game strategies (which constitute the program) combine to form a visual representation of their execution, called a game trace.

Compared to Hagl, the visual language supports a much narrower domain, focusing exclusively on pattern-based strategies for iterated games represented in normal form. These restrictions sacrifice expressiveness for explainability. The restriction to normal form games allows us to directly reuse the normal form notation used by game theorists to attain a high closeness of mapping with the domain [16]. The restriction to pattern-based strategies flattens the relationship between strategies and their execution, resulting in high *traceability*.

Since the work collected in this thesis, we have further refined Hagl and developed the paradigm of EOP through application to other domains. Chapter 5 describes some of this subsequent work.

# Chapter 2 – Hagl: A Haskell DSEL for Experimental Game Theory

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.

Published as:
Eric Walkingshaw and Martin Erwig.
A Domain-Specific Language for Experimental Game Theory.
*Journal of Functional Programming*, Volume 19, pages 645–661, 2009.

## 2.1 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 [4], 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]
```

`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.

Internal nodes are either `Decision` nodes or `Chance` nodes and can be sequenced arbitrarily with the tree. `Decision` nodes represent locations in the game tree where 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, `Chance` nodes represent points where 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]]
```

[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/

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 within 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.

## 2.2 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 Fig. 2.2, a notation known as *normal form*. In the iterated form, the game is played repeatedly, with the payoffs of each iteration accumulating.

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

Figure 2.1: Normal-form representation of the prisoner's dilemma

Robert Axelrod's seminal book *The Evolution of Cooperation* [1] 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 [21]. 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, then a tournament in which each player faces every player (including itself) 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)
```

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 4.2.1, but first let's turn our attention to the definition of normal-form games.

## 2.3 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 as arguments:

- the number of players, $n$, that play the game,
- a list of lists of available moves for each player, $m_1, m_2, ..., m_n$,
- and a list of $|m_1| \times |m_2| \times ... \times |m_n|$ 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 3.1 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 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.

```
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, 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.

## 2.4 Extensive-Form Games

Recall from Section 3.1 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 [34].

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 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 `Decision` and `Chance` nodes this means concatenating all branches, for `Payoff` nodes, the payoffs are simply added together. If the US chooses action, either by a naval blockade or an air strike, the USSR must counter-respond.

```
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 helper functions for accessing game tree nodes in both breadth-first and depth-first order) 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 sub-optimally. 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 makes adding these additional payoff modifiers, and additional decisions that may result, very straightforward.

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.

## 2.5 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 2.6 and 2.7, 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 2.4.

More complex strategies, like Tit for Tat defined in Section 2.2 and Khrushchev in Section 2.4, can be built from a suite of functions designed for the task. These functions primarily fall into two categories: *data accessors* and *list selectors*.

### 2.5.1 Data Accessors

Strategies have access to a wealth of information about the current state of a game's execution. In Section 2.6 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 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 when 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 2.7.

- `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, 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.

### 2.5.2 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.

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 2.5.1 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 GHC [14].

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. 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.

### 2.5.3 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 2.2 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.

```
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 2.3. The strategy plays randomly in the first game, 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 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.

## 2.6 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,
- 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 3.1. The definition of the `Player` type will be given in Section 2.7. Here, let's examine the `Transcript` and `History` types, 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 2.5.1 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.

```
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`, then runs the given computation on that state. The following, run from an interactive prompt (e.g. `GHCi`), would play ten iterations of the prisoner's dilemma between players `a` and `b`.

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

Unfortunately, running ten 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 2.2. 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 2.2 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 2.2. Only the final score is shown below; scores of individual matchups are omitted for space.

```
> 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, but also 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.

## 2.7 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 amongst different players within Hagl, and for labelling output like player scores. Also notice that `Player` is a locally quantified data constructor [23], another extension to Haskell 98 available in 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 2.6, 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 2.5.1. 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.

## 2.8 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 2.5.2. 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, 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.

## 2.9 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.

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 [10]. 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.

# Chapter 3 – Extending the Expressiveness of Hagl

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.

Published as:
Eric Walkingshaw and Martin Erwig.
Varying Domain Representations in Hagl – Extending the Expressiveness of a DSL for Experimental Game Theory.
*IFIP Working Conf. on Domain-Specific Languages*, LNCS 5658, pages 310–334, 2009.

## 3.1 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 [13].

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 [25]. 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 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 [20]. 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 [7]. 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 [1] exemplifies the problem. Experimenters wrote a custom Java library (IPDLX) to conduct the experiments [21]. 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 [35] we have provided a suite of operators and smart constructors for defining common types of games, a combinator library 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.

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

```
> 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.

### 3.1.1 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 3.2.2. 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 3.2.3 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 3.2.3. 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 3.3.2, 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 3.5.2 we provide an alternative implementation of the match game using the improved model developed in Section 3.4. 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 3.5.2.

### 3.1.2 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 3.3 we discuss the bias present in our original game representation and the limitations it imposes. In Section 3.4 we show how we can overcome this bias, generalizing our model to support many distinct domain representations. In Section 3.5 we utilize the new model to present solutions to the problems posed in Section 3.3.

## 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.

| | **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

Figure 3.1: Normal form representation of two related games

Sections 3.2.1, 3.2.2, and 3.2.3 all focus on the definition of different types of games in Hagl, while Section 3.2.4 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 [35].

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 3.2.3 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 3.3.

### 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. 3.1 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.

The first game in Fig. 3.1 is the prisoner's dilemma, which was briefly introduced in Section 3.1; 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. 3.1. 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 3.1 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.

### 3.2.2 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 3.2.4, 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 [9]. 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.

### 3.2.3 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 3.4, 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 3.1.1 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.

### 3.2.4 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 3.1.1, 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 3.3.1, 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.

## 3.3 A Biased Domain Representation

In Sections 3.2.1, 3.2.2, and 3.2.3 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.

### 3.3.1 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* [13].

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 3.2.1, 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) [26], and even the much more constrained problem of deciding whether a game has a pure Nash equilibrium is NP-hard [15]. 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 [34].

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.

### 3.3.2 Loss of State in State-Based Games

Section 3.2.3 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 3.2.4, 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.

## 3.4 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 3.3.2, 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 3.3.1.

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 3.3: 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 [5], an extension to Haskell 98 [28] 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.

### 3.4.1 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 3.2.4. 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 3.4 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 3.5 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.

### 3.4.2 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 3.5.1.

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 3.5.2.

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.

## 3.5 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 3.5.1 demonstrates the new representation of normal form games and provides functions for computing analytical solutions. Section 3.5.2 provides a new implementation of the match game and a strategy that can access its state in the course of play.

### 3.5.1 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 3.2.1. 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 3.4.2, 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 3.3 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 [17].

```
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 [34]. 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.

### 3.5.2 Representing and Playing the Match Game

In Section 3.1.1 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 3.5.1, 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 3.4.2 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.

## 3.6 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 3.4.1 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 [10].

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.

# Chapter 4 – A Visual Language for Representing and Explaining Strategies

Abstract: We present a visual language for strategies in game theory, which has potential applications in economics, social sciences, and in general science education. This language facilitates explanations of strategies by visually representing the interaction of players' strategies with game execution. We have utilized the cognitive dimensions framework in the design phase and recognized the need for a new cognitive dimension of "traceability" that considers how well a language can represent the execution of a program. We consider how traceability interacts with other cognitive dimensions and demonstrate its use in analyzing existing languages. We conclude that the design of a visual representation for execution traces should be an integral part of the design of visual languages because understanding a program is often tightly coupled to its execution.

Published as:
Martin Erwig and Eric Walkingshaw.
A Visual Language for Representing and Explaining Strategies in Game Theory.
*IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 101–108, 2008.

## 4.1 Introduction

Game theory has had a broad impact on the scientific world, with applications in economics, computer science, and many social sciences. From this breadth arises a need for representing and explaining concepts in game theory to an equally broad audience.

But in addition to scientists and other experts that apply game theory in one way or another as part of their job, the general public needs to be educated as well. In general, it is becoming more and more important to educate the general public about scientific findings and to increase scientific literacy. Broad public support for science and research, which is needed to justify the research (and its funding), can be achieved only by establishing a basic understanding of the scientific principles. How the lack of scientific literacy can negatively impact scientific research and education can be observed today in the United States in sometimes bizarre discussions about issues like the teaching of evolution or funding for stem cell research.

Why does game theory deserve public attention? In addition to offering a mathematically grounded way to strategize in particular situations, game theory has had a huge impact providing descriptive explanations of naturally occurring phenomena. One of the best examples is insight into why humans and animals cooperate and how such cooperation has evolved, described in Robert Axelrod's seminal book *The Evolution of Cooperation* [1]. The book focuses on one of the most important and well-known games in game theory: the Prisoner's Dilemma. In addition to cooperation, the iterated form of this simple game has been used to describe situations from the use of performance enhancing drugs in sports [30] to the nuclear arms race of the Cold War era [34].

Therefore, a notation that can help experts and lay people alike with describing and understanding games and strategies should not only be regarded as the basis for potential programming and simulation tools, but also as a language/medium to communicate and explain scientific results to a broad audience.

Although several visual notations already exist for concisely and clearly defining different types of games, the existing representations of strategies for iterated games are pseudo-code, inflexible and unscalable tables, and unstructured text, which leaves open the question of a visually appealing notation that can be used to explain game strategies and how they work.

In game theory, a game is a formal representation of a situation in which players interact and attempt to maximize their own return. Players interact by making discrete moves and a player's return is quantified by a payoff value. Players are only concerned with maximizing their own payoff. An iterated game simply has the same set of players play the same game repeatedly with the payoffs of each iteration accumulating.

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 what

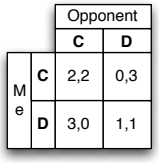


Figure 4.1: Prisoner's Dilemma in normal form

makes the game interesting is that if both players cooperate they will do better than if they both defect. 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", submitted by Anatol Rapaport. Tit for Tat was much simpler than many of its opponents, it cooperates on the first move and thereafter plays the last move 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 [1].

A formal definition of the Prisoner's Dilemma is is given in Figure 4.1 in a notation known as "normal form". While normal form can technically be used to represent any game in game theory [34], its use is usually restricted to games with two players, each of whom must select one move from a finite list of moves without knowledge of the other player's move. The payoffs for each player are given by a pair of numbers in the cell indexed by each player's move. In this game, for example, if "Me" chooses to cooperate (indicated by the "C") and "Opponent" chooses to defect ("D"), the relevant cell contains the pair $0, 3$ indicating a payoff of 0 for "Me" and a payoff of 3 for "Opponent".

In this paper we introduce an *integrated* notation for defining game strategies and game execution traces that is based on this well-known representation. The emphasis is not just on the visual language for describing strategies, but rather on the design of a notation that allows the combined representation of strategies *and* game execution traces. The ultimate design goal is to obtain a representation that supports explanations of how strategies work. We have utilized the cognitive dimensions framework in the design phase of our notation. Having realized that the motivation for the integrated notation and the explanatory component was only partially covered by existing cognitive dimensions, we have identified a new cognitive dimension of "traceability", which measures the ability of a notation to represent execution and the relationship between a program and its execution.

The remainder of this paper is structured as follows. In Section 4.2 we introduce the notation for strategies, traces, and their composition. In Section 4.3 we describe the design process based on cognitive dimensions. This section also shows how some crucial design decisions could not be

supported by existing cognitive dimensions and thus provides the motivation for the new cognitive dimension of traceability that is defined and illustrated with further examples in Section 4.4. We discuss related work in Section 4.5 and present some conclusions in Section 4.6.

## 4.2 A Notation for Game Theory

The notation we have designed is composed of three distinct but related components. The first is the notation for defining strategies, defined in Section 4.2.1; the second is the representation of game traces, defined in Section 4.2.2; and the third is a view which demonstrates how any given game instance within a game trace occurred, given the players' strategies and the game trace, defined in Section 4.2.3.

### 4.2.1 Strategy Notation

Our notation is based on the well-known normal-form game representation that we have briefly shown in the previous section. This representation views a game as a matrix indexed by moves that contains the payoff result for each possible combination of moves in its cells. More formally, if the set $M_i$ represents the set of moves available to player $i$, the *domain* of an $n$-player game is given by $D = M_1 \times \ldots \times M_n$, and a game is a mapping $G : D \to \mathbb{R}^n$ so that the number $r_j$ in the tuple $(r_1, \ldots, r_n) = G(m_1, \ldots, m_n)$ represents the payoff for player $j$ in the game that results when player $i$ plays move $m_i$. Since we consider only 2-player games, our domain will always be $D = M_1 \times M_2$, and games are given by mappings $G : D \to \mathbb{R}^2$.

A strategy is defined by a list of rules (given below the dotted line) and a possibly empty horizontal list of initial *move patterns* (given above the dotted line). A rule $L \to R$ is given by a *matching pattern* ($L$) and a move pattern ($R$). Figure 4.2 demonstrates our visual notation for strategies with a definition of Tit for Tat.

A move pattern is a special case of a *simple pattern*, which is obtained from the normal-form matrix notation by marking a subset of cells in the matrix. Formally, a simple pattern can be represented by a binary partition of the matrix domain $D$ into marked and unmarked cells, and this partition is uniquely determined by the set of marked cells. A move pattern for player 1 is a simple pattern such that the marked set equals $\{m\} \times M_2$ where $m \in M_1$. So, for example, the initial move of Tit for Tat is defined by $C \times \{C, D\} = \{(C, C), (C, D)\}$.

If present, a list of $k$ initial move patterns defines the first $k$ moves of the player that uses this strategy. In the case of Tit for Tat, the first move is always $C$. Note that the moves (and patterns) in the strategy are defined from the perspective of the player "Me". A strategy for the player "Opponent" can be obtained from a "Me" strategy (and vice versa) by flipping rows and column markings in all matching and move patterns, as defined by the function *flip*.

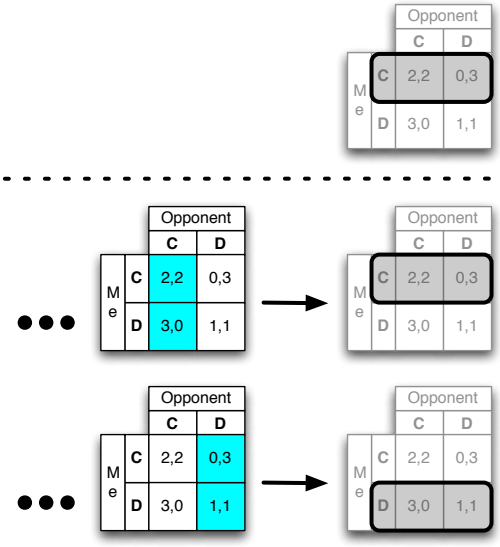


Figure 4.2: Definition of Tit for Tat

$$\begin{aligned} \mathit{flip}(\{m\} \times M) &= M \times \{m\} \\ \mathit{flip}(M \times \{m\}) &= \{m\} \times M \\ \mathit{flip}(L \to R) &= \mathit{flip}(L) \to \mathit{flip}(R) \end{aligned}$$

A *game play* is given by a pair of moves, one for each player. That is, $p \in D = M_1 \times M_2$, and a *game trace* is defined as a sequence of game plays, $p_1 \ldots p_n$.

The meaning of a set of rules is defined relative to an existing game trace, and the meaning of a single rule is given by the move pattern on the right. The first rule whose matching pattern matches the current game trace is selected, and its move pattern defines the next move.

Matching patterns, as found on the left side of rules, take one of the following five forms where $P_i$ is the set of cells representing a simple pattern. The conditions under which each pattern matches a game trace $p_1 \ldots p_n$ are shown on the right.

| | | |
|---|---|---|
| *Recent* | $\bullet\bullet\bullet P_{n-k} \ldots P_n$ | if $\forall i \in \{n-k, \ldots, n\}, p_i \in P_i$ |
| *Initial* | $P_1 \ldots P_k \bullet\bullet\bullet$ | if $\forall i \in \{1, \ldots, k\}, p_i \in P_i$ |
| *Always* | $P \bullet\bullet\bullet P$ | if $\forall i \in \{1, \ldots, n\}, p_i \in P$ |
| *Sometimes* | $\bullet\bullet\bullet P \bullet\bullet\bullet$ | if $\exists i \in \{1, \ldots, n\}, p_i \in P$ |
| *Default* | (nothing) | always matches |

Both rules in the definition of Tit for Tat utilize the *recent* matching pattern—they both consider only the most recent entry in the trace. The definition of Grim Trigger given in Figure 4.3 is an example of a strategy that utilizes both the *sometimes* pattern and the *default* form.

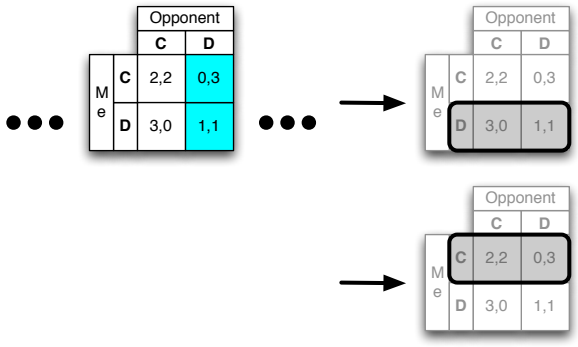


Figure 4.3: Definition of Grim Trigger

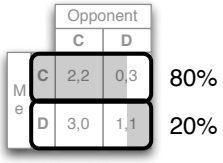


Figure 4.4: Example mixed strategy

Grim Trigger is like Tit for Tat in that it will continue to cooperate as long as the opponent cooperates, but unlike Tit for Tat, Grim Trigger never forgives. Once an opponent defects, Grim Trigger will defect forever thereafter. Colloquially, the visual definition would read "if the opponent has defected any time in the past, then defect; otherwise, cooperate". Also note that the inclusion of a default matching pattern obviates the need for initial moves.

Strategies also often make use of "mixed strategies", which are strategies in which a move is selected randomly based on some probability distribution. Thus a *mixed pattern* can be used in place of a move pattern anywhere a move pattern would otherwise appear. A mixed pattern is represented by multiple selected moves, where the shading of each selection is altered to correspond to its likelihood of selection. Annotations describe the distribution more exactly. The example in Figure 4.4 shows a mixed pattern that will cooperate with 80% probability and defect with 20% probability.

Two more strategies to be used in the following discussion are Tit for Two Tats, which is shown in Figure 4.5 and defects only after two consecutive defects of the opponent, and Opponent Alternator, shown in Figure 4.6.

The yellow color used in the matching pattern of the Opponent Alternator not only helps to distinguish an "Opponent" from a "Me" strategy, it also supports the integrated display of rules

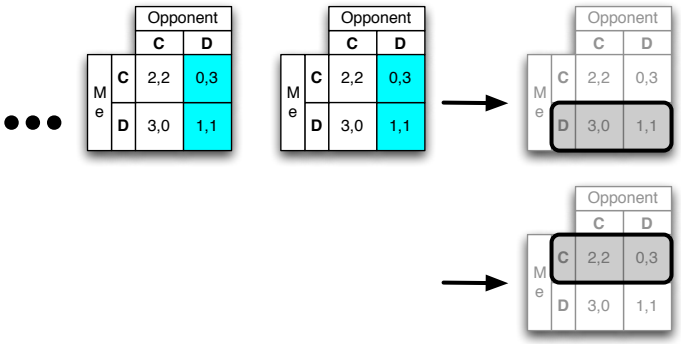


Figure 4.5: Tit for Two Tats strategy

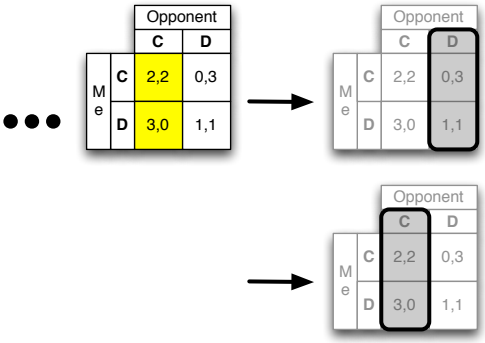


Figure 4.6: Opponent Alternator strategy

and traces as shown in Section 4.2.3. Note that since the Opponent Alternator wants to do the opposite of *its own* last move, it is the column that must be colored.

### 4.2.2 Representing Traces

As defined earlier, a game play is given by a pair of moves. Visually, a game play is represented by the intersection of two move patterns, as shown on the right. The resulting game outcome is not only indicated by the intersecting frames of the move patterns, but also by the darker cell shading that result from the overlay of the pattern coloring. In the example in Figure 4.7, both players chose to cooperate, resulting in a payoff of 2 for each player.

For representing game traces, we introduce a more abstracted view of a game play. The normal form representation of the game is stripped of headers and labels and represented as a

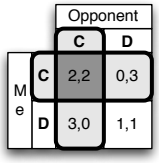


Figure 4.7: Example game play

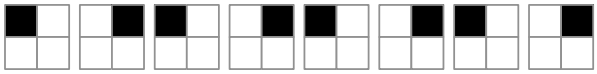

Figure 4.8: Example game trace

simple grid of potential outcomes. The outcome that was achieved in a particular game play is indicated by filling in the corresponding box in the grid.

Figure 4.8 shows a trace of iterated Prisoner's Dilemma played Tit for Two Tats against Opponent Alternator. Here we can clearly see the pattern of outcomes that results when these two strategies face each other.

A second view of a game trace utilizes colors on a green-red gradient for outcome shading to represent the quality of the payoff from the perspective of either one of the players or for both players combined. The shading color is scaled linearly based on the rank of the outcome so that the best possible outcome from the given perspective will be bright green, the worst possible bright red and intermediate outcomes will be hues of yellow-green, yellow, and orange.

Figure 4.9 shows the same trace as above, first from the perspective of "Me", playing the Tit for Two Tats strategy and then from the perspective of "Opponent" playing the Alternator strategy. At a glance, it is clear that Alternator is winning.

A trace representation can be useful for analyzing the output of two strategies, but for understanding how a trace was generated, we also provide a notation for explaining how a pair of strategies produce a given game play.

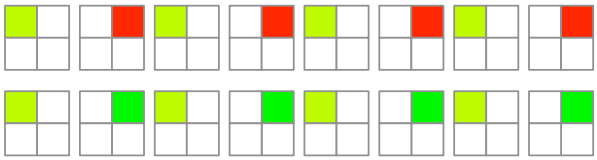

Figure 4.9: Traces showing individual payoffs

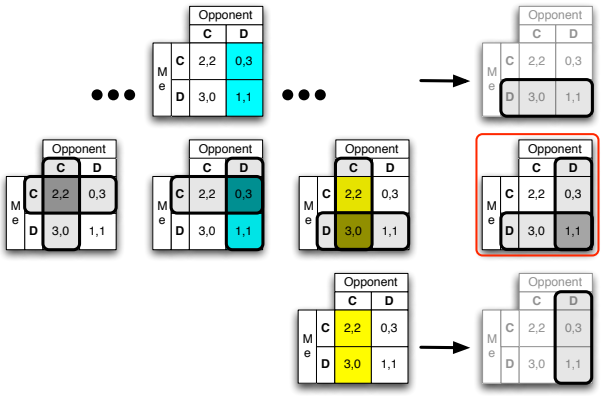


Figure 4.10: Game play explanation

### 4.2.3 Relating Strategies to Traces

By selecting a specific game play from a high-level view of a trace, we can zoom in to a detailed visual explanation as shown in Figure 4.10, which shows an explanation of the fourth game play in a trace of Grim Trigger vs. Opponent Alternator. The selected game play is enclosed in a red box with earlier game plays in the trace shown on the same row to the left. The matching rule from the Me strategy is shown above the game trace, and the matching rule from the Opponent strategy is shown below the game trace. Game patterns from the matching rules are shown centered above the matched game plays, and their patterns are mapped onto the trace. The different colors highlight the different places in the trace where the patterns matched.[1]

In this way a user can see exactly how a specific outcome occurred given the trace that preceded it. For any previous game play that is relevant to the selection, a matching pattern will appear, and the outcome of that play will fall within the pattern.

A further expanded view (omitted here for lack of space) can be given in which the strategies are given in full, with matching strategies indicated by an asterisk, to allow the user to view the entire strategies of each player in conjunction with the trace.

Also, due to space limitations, we had to focus in this paper on only one game, the Prisoner's Dilemma. Other games can be dealt with similarly. For example, the well-known game of Rock-Paper-Scissors can be inspected at: `http://eecs.oregonstate.edu/~erwig/rps.pdf`.

[1] When a Me and an Opponent pattern match and overlap in one game play, the overlapping cells are colored in green.

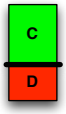


Figure 4.11: Initial player move design

## 4.3 Discussion of Language Design

Throughout the design process we utilized the cognitive dimensions framework supplemented with our new cognitive dimension of traceability (discussed in depth in Section 4.4). In this section we discuss some of the decisions that were made using these intellectual tools.

Individual cognitive dimensions often conflict with each other. To use them effectively requires identifying the dimensions that are most relevant considering the goals of the language, then weighting these more heavily when making decisions involving tradeoffs. Since we believe that traceability greatly impacts understandability, and facilitating strategy explanations was one of our primary goals, traceability emerged as our most heavily weighted dimension.

A central element of both strategy notation and game traces is the normal-form game representation. By basing our language on this ubiquitous and concise notation, we attain a high closeness of mapping with the target domain. Our decision to utilize this notation was made very early on, and many subsequent decisions were centered on conveying various meanings within this framework.

That a game play should be represented by marking the corresponding outcome in a game instance, and that a game trace should be composed of a sequence of such game plays seemed intuitive, and is justified by a high level of role-expressiveness for these aspects of the language. How to represent player moves and matching patterns was less obvious. Since traceability was of such high importance, we knew that we wanted these aspects of our language to integrate well with game traces.

### 4.3.1 Design of Move Patterns

Our initial design for player moves is shown in Figure 4.11–a vertical rectangle with a horizontal slider dividing it into two areas. The primary strength of this design is that it concisely represents mixed strategies. Patterns with pure right hand sides would simply have the slider dragged all the way to one side or the other, representing a 100% probability of choosing the given strategy. This syntax could be extended to games with more than two moves by adding more parallel sliders, partitioning the rectangle into the desired number of areas.

We struggled to integrate this notation with program traces, however. We decided that move patterns, like matching patterns, should be represented on a normal form representation of the game itself. This would increase closeness of mapping and consistency, and we hoped it would lead to better traceability.

Our final design, as described in Section 4.2.1 portrays a move by outlining the corresponding row or column on a normal form game representation. In the definition, we describe a move pattern as a special case of a simple pattern where the marked cells are all in one row or column (depending on the player). We extend the outline to include the corresponding row or column label to make this otherwise hidden dependency more explicit.

When move patterns from each player are combined, we get a game play, and the outcome is defined by the intersection of the two moves. Marking the relevant cells by outlining them (rather than coloring) enables game plays, in turn, to be combined with pattern matches, which are indicated by color. Making sure all of these elements combined well together was critical to attaining a high level of traceability.

We found, however, that just using outlines to indicate moves made the outcomes difficult to detect at a glance, indicating poor perceptual mapping. To counter this impression, we added a light shading to move outlines such that when they are overlaid they produce a darker shading (which we actually exaggerate) to clearly indicate the outcome of the game. Although the shading causes some conflict with pattern matching, the use of a gray shading vs. colorful patterns mitigates this issue, and we considered the tradeoff to be worth the change.

Fortunately, this change also facilitated a solution to the problem of representing mixed strategies, which we had set aside. As described above, mixed strategies are represented by adding additional outlines and adjusting the amount of each that is shaded. When a mixed move pattern is resolved into an actual move (that is, a move is selected from the distribution), the mixed pattern is replaced with a regular move pattern in the trace. This decision violates consistency, which is unfortunate, but not doing so would make game traces much more confusing and cause a huge sacrifice in traceability, which made us accept this deficiency. We also considered using different degrees of shading to represent mixed strategies, but it turned out that differences in shading were too subtle. Moreover, with shading the aspect of choice between moves is largely lost.

Finally, we decided to gray out the game representation that move patterns are defined on. The game beneath a move pattern represents a game that is only a future possibility which has not happened yet and is fundamentally different from realized games in a game trace or in matching patterns (which correspond directly to games in the trace). This decision was also made for consistency reasons—the notation reflects the fact that future games and realized games are semantically different objects.

### 4.3.2 Design of Game Traces

Game traces can be viewed at two abstraction levels. The first is at the level that strategies are defined. Here we can view an explanation of the trace relative to the strategies that produced it as described in Section 4.2.3. Many of the design decisions that were involved in making this level of abstraction work are described in the previous section, but a few others warrant mentioning. First, aligning the matching pattern of the applicable rule with the corresponding game plays in the trace was an easy decision made to maximize role-expressiveness and minimize hidden dependencies. Second, we had to consider a tradeoff between low diffuseness and hidden dependencies in determining whether or not to only show the matching rule of each strategy, or to show the strategy in its entirety with the matching rule indicated separately. Ultimately, we decided that each had a role in the language and decided on the dual explanation and expanded explanation views.

The second trace abstraction level is the high-level view described in Section 4.2.2. Our goal here was to simplify the notation as much as possible to allow users to focus solely on the game trace. We believe that the simplified game representations used in this view maximize traceability, terseness, and visibility while maintaining reasonable levels of consistency and role expressiveness.

By examining the game trace with minimal distractions, our hope is that users can clearly see how strategies interact with each other by the pattern (or lack thereof) represented in their game trace. The high-level view also includes a way to visualize the success of a strategy or combination of strategies by color-coding the outcome based on the quality of the payoff. We believe that this representation provides excellent perceptual mapping as it makes the relative success indicated by a trace recognizable at a glance.

## 4.4 Traceability as a Cognitive Dimension

Traceability measures a notation's ability to represent and to relate to its semantics. A more general name for this aspect would be something like "semantic accountability", but since in many cases the semantics is represented by a trace, we have chosen the more concrete name. The rationale for investigating the traceability of a notation is to get a sense of the overall understandability of the notation and, in particular, of how much program understanding can be supported by the trace notation.

The investigation of traceability requires a definition of the trace notation, which depends on the program notation. This also means that traceability can be investigated only for those notations for which the notion of a trace makes sense, which will be most program notations.

In our domain of iterated games in game theory, a trace is given by a sequence of individual

game instances. When considering imperative languages, a trace may be a series of memory fragments. A trace for lambda calculus is given by a sequence of lambda expressions. In each case the sequence of elements in a trace is not arbitrary, but rather the result of an effect caused by part of the program (a pair of rules, a statement from the imperative program, or an application of a lambda abstraction to an expression).

Traceability involves two distinct, but closely related aspects. First, it expresses how well traces can explain the meaning of a program. Second, it reflects how integrated the notations for traces and programs are, and in particular, how well a specific program part can be related to the part(s) of the trace it is affecting.

In this context, we should also mention the distinction between "static" and "dynamic" traces. A static trace represents the time component explicitly, that is, spatially. For example, our notation features static traces, representing the passage of time by moving left to right along the horizontal axis. On the other hand, typical debuggers for imperative languages produce a dynamic trace, where the passage of time is represented by replacing one state with another.

It seems that static traces are prefereable in principle since they reduce the need for memorizing trace elements and can thus reduce the cognitive load on the viewer whereas complex dynamic traces often prompt users to rerun the program because old state can be "forgotten". On the other hand, when the trace notation becomes too complex or too large, a dynamic trace might be required as a modularization device to make the communication of the huge amount of information possible at all.

### 4.4.1 Interaction with Existing Cognitive Dimensions

The existing cognitive dimension of closeness of mapping measures how well a program relates to its abstract semantic domain. In contrast, traceability measures how well a program relates to a concrete representation of its semantic effects. Therefore, traceability is similar in nature to closeness of mapping, but uses a trace as a different, more concrete target. Traceability may be considered something of a second-order cognitive dimension since it reflects closeness of mapping between program and execution notation rather than program notation and abstract domain.

The ultimate goal of traceability is to make programs easier to understand. We believe that by visually relating a program to its effects, relationships between a program and its output can be made clearer and easier to understand by people. Other cognitive dimensions have this same motivation. One is visibility, which considers how easily elements of a program's representation can be viewed. Traceability considers how easily a program's relationship to its execution can be viewed. In some sense, traceability is an extension of visibility from static representation to a program's execution.

Traceability is also related to the existing cognitive dimension of progressive evaluation. Pro-

gressive evaluation considers whether or not a program can be executed before it is completely written. The intention is to help a user check their program's correctness as they are writing it. Although traceability is primarily concerned with viewing the execution of completed programs, by increasing the visibility of program execution, it may also help increase a user's confidence in program correctness.

In designing our notation for strategies we recognized a tradeoff between traceability and abstraction. We considered adding an abstraction mechanism to our language that would have allowed other strategies, referenced by name, on the right-hand side of a strategy definition. Besides the usual well-known benefits of abstraction, this change would have also made our language fundamentally more expressive, allowing us to create strategies with more complex matching patterns (for example, multiple existential patterns).

This abstraction mechanism, however, made it much more difficult to illustrate how strategies interact with a game trace. In other words, it negatively affected traceability. Since high traceability was one of our primary design goals, we decided not to adopt this language feature despite its benefits elsewhere.

### 4.4.2 Independence of Trace Role and Integration

Traceability as a measure for program understandability depends on how much the trace plays a part in understanding the program and how well the trace can be related to the program. By examining traceability for several existing languages we would like to demonstrate that these two aspect are orthogonal.

In some languages, such as Logo [19] or Alice [6], the trace *is* the program output, that is, it does not need motivation or additional definitions and thus is an integral part of understanding programs. On the other hand, in traditional imperative languages or lambda calculus [2] a trace is only employed to show intermediate results, which means that looking at a trace requires a mode of executing a program that is different from running the program just for its result and that is typically employed only if the result asks for an explanation.

The prominent role of traces in Logo or Alice does not imply a tight integration of traces and program notation, which is obvious since in both cases the program notation is textual, but the traces are pictures or animations. Explicit mechanisms to annotate the program notation and synchronize it with the visual program output are required to enable tracing. In this respect, the situation for Logo and Alice is very similar to that of debugging interfaces of traditional imperative languages.

On the other hand, traces explaining the evaluation of lambda calculus expressions are highly integrated with the program notation since each part of the trace is a (partially evaluated) program (that is, a lambda expression). All that is needed as additional notation is typically the

underlining of the subexpression being reduced in the next step.

### 4.4.3 Traceability as a Notation Design Tool

To elevate the notion of traceability to a cognitive dimension we should give advice as to what kind of changes in notation could increase traceability [3]. Such guidelines are important if we want to have traceability not only as an evaluative tool, but also as a design tool.

One strategy to improve traceability is to share notation between traces and programs to support the identification of program parts with relevant parts of the trace. Also, aiming for modular program notation allows the isolation of program parts to facilitate the combination with traces.

Moreover, based on the observations in Section 4.4.2, we believe that observing and making explicit the two aspects of role and integration for a particular language, helps to find ways to improve the language. For example, to increase traceability of Alice we could try to define a notation that lets program (parts) be associated with the animated objects, which would increase the integration aspect tremendously.

## 4.5 Related Work

Our notation builds on the normal-form representation of games—a matrix of payoffs indexed by each players' move. While very common, it is not the only well-used game representation.

Games in *extensive form* are represented as trees. Decisions and variables are represented as non-leaf nodes and payoffs as leaves. Moves are represented as edges from a decision node to its children. This representation can more easily represent games in which a single player must make more than one decision, games involving random variables, and games with more than two players. This increased flexibility, however, comes at the expense of terseness.

*Multi-agent influence diagrams* are high-level graphs showing the relationships between variables and the decisions made by agents in a game [22]. This representation excels at showing relevance in real-world games with many external variables, but usually omits the move and payoff information required to define a game precisely.

Some aspects of our design could be extended to these other representations. Matching patterns of previous payoffs, for example, could work with extensive form games by simply replacing matrices with trees and marking the outcomes by highlighting the corresponding leaves. Move patterns could be represented by highlighting edges from decision nodes; these could be combined to form a game play, represented by a complete path from the root to a leaf node. There are many obstacles that would have to be addressed for such an approach, however. Inheriting from the normal-form representation, our notation assumes *imperfect information*,

which means that a player makes his move without knowledge of the other player's move for this game. Extensive form games often lack this assumption, however. Extensive form games also easily accommodate situations in which a player must make more than one decision in a single game, which is not something our current design considers.

Representing strategies is a surprisingly unexplored field. Strategies are typically defined textually or with psuedo-code. One attempt at providing a more user-friendly means to specify strategies for the Iterated Prisoner's Dilemma competition provided tables enumerating every possible combination of moves for the past one to three rounds of play, allowing the user to specify an action for every case [21]. Besides not scaling past a few games of recent history, there are a many common strategies that are simply not expressible in this notation.

The cognitive dimensions framework provides a terminology relating programming language design to cognitive issues in programming [16]. The framework has been built upon and extended many times, for example, [8, 32, 37]. Actively encouraging extensions, Alan Blackwell outlines some guidelines for proposing new dimensions in [3]. We have tried to take these guidelines into account in our definition of traceability in Section 4.4.

Our visual notation for strategies employs the idea of visual rewrite rules that is part of many visual languages. Two well-known examples are AgentSheets [29] and StageCast creator [31]. In the AgentSheets approach, for instance, a program is given by a set of visual rewrite rules that map one graphic state, represented as a 2D tile, to the next. When a program is run, these rules are applied repeatedly producing an animation. These animations can be considered a form of dynamic trace which helps to visualize and understand the effects of the rules. AgentSheets lacks a mechanism, present in our language, to explicitly relate an element of the current state to the corresponding rules that generated that state, which would be difficult since the state manipulated is more complex than that of game instances. Moreover, AgentSheet rules can refer only to the previous state, which means that actions described by rules cannot be predicated on the history of the state, and existential or forall quantifications cannot be expressed. Since AgentSheets offers many more general-purpose programming elements than our game theory notation, it is not surprising that it is more difficult to achieve the same degree of traceability, an observation that reflects the tradeoff between traceability and abstraction.

Software visualization considers the representation of programs and their execution to facilitate reasoning and understanding throughout the software development cycle [33]. Our new cognitive dimension of traceability extends these considerations to the language design phase. Petre and de Quincey state, in an overview of the field of software visualization, "Currently, it is still arguable that what is visualized is what can be visualized, not necessarily what needs to be visualized" [27]. By addressing traceability in the language design phase, languages can make their programs more amenable to visualization, which will make it easier to increase visualization coverage to those areas that most need it.

## 4.6 Conclusions and Future Work

This work is part of our effort to develop a new paradigm of explanation-oriented languages, which are languages whose objective is not only to describe the computation of values, but also to create explanations of how and why those values are obtained. By directing the language designer's attention to the notation of traces and the integration with the program notation the cognitive dimension of traceability advocates the design of notations with explanatory value.

As we have demonstrated, the presented notation for game strategies and traces exhibits a high degree of traceability and can thus serve as an explanatory tool for game theory for a broad audience.

For future work we intend to apply design explanatory notations in other domains of public interest (for example, to explain simple probabilistic reasoning) using the traceability criterion as a design guideline. Moreover, we plan to consider the redesign of existing languages to improve their traceability.

# Chapter 5 – Conclusion

Since the publication of the papers included in this thesis, both the Hagl language and our ideas about the explanation-oriented programming paradigm have continued to evolve. This section will briefly describe some of this more recent work.

The current representation of games in Hagl strikes a balance between the representations presented in Chapters 2 and 3, and in many ways offers the best of both worlds. Games are again transformed into a single internal tree representation, offering the generic transformation and traversal advantages of a simple data type representation. However, this transformation is delayed and encapsulated in a type class rather than applied immediately in a smart constructor. This allows essentially all of the benefits described in Chapter 3. Internally, games are trees, while outwardly they may have a more restricted structure that supports representation-specific analyses and strategies.

Another significant change is that games in Hagl are no longer inherently sequentially iterative. Instead, games are defined more simply in terms of a single execution. Everything associated with iterated games has been modularized and a single-execution game can be made iterative by applying the type constructor `Iterated g`. For example, if the type `PD` represents the prisoner's dilemma, `Iterated PD` represents its iterated form. Besides being conceptually appealing, this opens up the possibility to explore new ways of organizing game plays within experiments. For example, instead of a simple linear sequence, we can imagine generating a probabilistic tree of game plays according to the potential outcomes of `Chance` events in the game.

Our work on EOP has continued into new domains concerned with explaining different types of formal reasoning. For example, in [11] and [12] we have developed a Haskell DSEL and a visual language for explaining probabilistic reasoning problems. The main idea explored in this work is the use of a story-telling metaphor as a basis for explanation. In this view, a story represents a sequence of states, where the initial state is some trivial or known state, and the final state is the *explanandum*—the thing that is to be explained. The goal of the story is to bring the reader from the initial state to the explanandum using a sequence of simple and well-understood events that transform the state. For this particular domain, the state at each point in the story is a probability distribution, while the events are a small set of distribution-transforming functions.

Although we identified it only later, the story-telling metaphor is also weakly reflected in the visual strategy explanation language shown in Chapter 4. For example, Figure 4.10 presents an explanation of a single game play. This explanation has a clearly defined explanandum, circled in red, and presents a sequential list of the preceding game plays that contributed to its occurrence.

The correspondence is not exact since significant contributors to the explanandum, the matching rules of each strategy, are indicated spatially rather than as steps in the story. However, a feature of the visual language not presented in Chapter 4 is that it can also be used to produce simple animations of the development of a game trace. In this case, the story-telling metaphor is more fully realized, embedded in time rather than space.

# Bibliography


[1] R. M. Axelrod. *The Evolution of Cooperation.* Basic Books, New York, 1984.

[2] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics.* North-Holland, Amsterdam, 1984.

[3] A. F. Blackwell. Dealing with New Cognitive Dimensions. In *Workshop on Cognitive Dimensions: Strengthening the Cognitive Dimensions Research Community*, University of Hertfordshire, December 2000.

[4] C. Camerer. *Behavioral Game Theory: Experiments in Strategic Interaction.* Princeton University Press, Princeton, 2003.

[5] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated Types with Class. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, volume 40, pages 1–13, 2005.

[6] M. J. Conway. *Alice: Easy-to-Learn 3D Scripting for Novices.* PhD thesis, University of Virginia, December 1997.

[7] V. P. Crawford. Introduction to Experimental Game Theory. *Journal of Economic Theory*, 104(1):1–15, 2002.

[8] R. M. Dondero Jr. and S. Wiedenbeck. Subsetability as a New Cognitive Dimension? In *Workshop of the Psychology of Programming Interest Group*, pages 230–243, 2006.

[9] M. Erwig and S. Kollmansberger. Probabilistic Functional Programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.

[10] M. Erwig and E. Walkingshaw. A Visual Language for Representing and Explaining Strategies in Game Theory. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 101–108, 2008.

[11] M. Erwig and E. Walkingshaw. A DSL for Explaining Probabilistic Reasoning. In *IFIP Working Conf. on Domain-Specific Languages*, volume 5658 of *LNCS*, pages 335–359, 2009.

[12] M. Erwig and E. Walkingshaw. Visual Explanations of Probabilistic Reasoning. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 23–27, 2009.

[13] D. Fudenberg and J. Tirole. *Game Theory.* MIT Press, 1991.

[14] GHC. The Glasgow Haskell Compiler, 2004. `http://haskell.org/ghc`.

[15] G. Gottlob, G. Greco, and F. Scarcello. Pure Nash Equilibria: Hard and Easy Games. In *Conf. on Theoretical Aspects of Rationality and Knowledge*, pages 215–230, 2003.

[16] T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[17] T. Groves and J. Ledyard. Optimal Allocation of Public Goods: A Solution to the Free Rider Problem. *Econometrica*, 45(4):783–809, 1977.

[18] G. Hardin. The Tragedy of the Commons. *Science*, 162(3859):1243–1248, 1968.

[19] Brian Harvey. *Computer Science Logo Style*. MIT Press, 2nd edition, 1997.

[20] T. H. Ho, C. Camerer, and K. Weigelt. Iterated Dominance and Iterated Best-Response in p-Beauty Contests. *American Economic Review*, 88(4):947–69, 1998.

[21] G. Kendall, P. Darwen, and X. Yao. The Prisoner's Dilemma Competition, 2005. `http://www.prisoners-dilemma.com`, last accessed September 26, 2011.

[22] D. Koller and B. Milch. Multi-Agent Influence Diagrams for Representing and Solving Games. *Games and Economic Behavior*, 45(1):181–221, 2003.

[23] K. Läufer and M. Odersky. Polymorphic Type Inference and Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.

[24] M. Milinski, D. Semmann, and H. J. Krambeck. Reputation Helps Solve the "Tragedy of the Commons". *Nature*, 415(6870):424–426, 2002.

[25] R. Nagel. Unraveling in Guessing Games: An Experimental Study. *American Economic Review*, 85:1313–1326, 1995.

[26] C. H. Papadimitriou. The Complexity of Finding Nash Equilibria. *Algorithmic Game Theory*, pages 29–52, 2007.

[27] M. Petre and E. de Quincey. A Gentle Overview of Software Visualisation. *Psychology of Programming Interest Group Newsletter*, September 2006.

[28] Peyton Jones, S. L. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003.

[29] A. Repenning and T. Sumner. Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *Computer*, 28(3):17–25, 1995.

[30] B. Schneier. Drugs: Sports' Prisoner's Dilemma. *Wired News*, August 2006.

[31] D. C. Smith, A. Cypher, and L. Tesler. Programming by Example: Novice Programming Comes of Age. *Communications of the ACM*, 43(3):75–81, 2000.

[32] M. Stacey. Distorting Design: Unevenness as a Cognitive Dimension of Design Tools. In *Adjunct Proceedings of the Int. Conf. on Human-Computer Interaction*, pages 90–95, 1995.

[33] J. T. Stasko. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998.

[34] P. D. Straffin. *Game Theory and Strategy.* The Mathematical Association of America, Washington, 1993.

[35] E. Walkingshaw and M. Erwig. A Domain-Specific Language for Experimental Game Theory. *Journal of Functional Programming*, 19:645–661, 2009.

[36] E. Walkingshaw and M. Erwig. Varying Domain Representations in Hagl – Extending the Expressiveness of a DSL for Experimental Game Theory. In *IFIP Working Conf. on Domain-Specific Languages*, volume 5658 of *LNCS*, pages 310–334, 2009.

[37] S. Yang, M. M. Burnett, E. DeKoven, and M. Zloof. Representation Design Benchmarks: A Design-Time Aid for VPL Navigable Static Representations. *Journal of Visual Languages and Computing*, 8:563–599, 1997.