A Language For Board Games

An embedded domain-specific language for the streamlined creation of positional board games.

Bachelor's thesis in Computer science and Engineering

Jennifer Krogh
Mattias Mattsson
Emma Pettersson
Simon Sundqvist
Carl Wiede
Mårten Åsberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Bachelor's thesis 2021

A Language For Board Games

An embedded domain-specific language for the streamlined creation of positional board games.

Jennifer Krogh

Mattias Mattsson

Emma Pettersson

Simon Sundqvist

Carl Wiede

Mårten Åsberg

UNIVERSITY OF GOTHENBURG

CHALMERS

UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering

Chalmers University of Technology

University of Gothenburg

Gothenburg, Sweden 2021

A Language For Board Games
An embedded domain-specific language for the streamlined creation of positional board games.

Jennifer Krogh Mattias Mattson Emma Pettersson Simon Sundqvist Carl Wiede Mårten Åsberg

© Jennifer Krogh, Mattias Mattson, Emma Pettersson, Simon Sundqvist, Carl Wiede, Mårten Åsberg 2021.

Supervisor: Ulf Norell, Department of Computer Science and Engineering
Examiner: Ana Bove, Department of Computer Science and Engineering

Bachelor's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

A Language For Board Games
An embedded domain-specific language for the streamlined creation of positional board games.

Jennifer Krogh, Mattias Mattsson, Emma Pettersson,
Simon Sundqvist, Carl Wiede, Mårten Åsberg

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis covers the process and theory behind creating a domain-specific language for board games. This language is specifically made for positional board games and allows for further implementation of games in that category. A selection of example games were created and shipped with the language to showcase its usage. The language also has a graphical user interface allowing the user to easily play the games together with another player on the same computer. Possible future work regarding this project is suggested in the *Discussion* chapter, such as expanding the language to work with more game categories and the possibility to play online or against a computer.

Keywords: board games, positional games, connection games, domain-specific language

Ett Språk För Brädspel
Ett inbäddat domänspecifikt språk för effektiviserat skapande av positionella brädspel.

Jennifer Krogh, Mattias Mattsson, Emma Pettersson,
Simon Sundqvist, Carl Wiede, Mårten Åsberg

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola och Göteborgs Universitet

Sammanfattning

Denna rapport omfattar processen och teorin bakom skapandet av ett domänspecifikt språk för brädspel. Detta språk är specifikt anpassat för positionsspel och erbjuder framtida möjligheter att implementera spel i den kategorin. Ett urval av exempelspel är redan skapade med språket för att visa dess användande. Språket har också ett grafiskt gränssnitt som tillåter användaren att lätt spela spelen tillsammans med en annan spelare som använder samma dator. Framtida utveckling av projektet föreslås, som exempelvis att bygga ut språket så det fungerar med fler spelkategorier och möjligheten att spela online eller mot en dator.

Nyckelord: brädspel, positionsspel, kopplingsspel, domänspecifikt språk

Acknowledgements

We would like to thank **Nils Anders Danielsson** for proposing this project, as well as our supervisor, **Ulf Norell**, for all the advice he gave us during the project. We would also like to thank **The Division for Language and Communication** at Chalmers for their help with handling images and sources, and also group **MVEX01-21-01** for proofreading our initial draft of the thesis.

Contents

List of Figures viii

List of Tables ix

1 Introduction 1
- 1.1 Purpose 1
- 1.2 Problem 1
- 1.3 Scope 2
- 1.4 Related Works 2
- 1.5 Structure 3

2 Theory 4
- 2.1 Domain Specific Languages 4
  - 2.1.1 Why use a DSL? 4
  - 2.1.2 DSL examples and their purpose 5
  - 2.1.3 Embedded DSLs 5
- 2.2 Games 6
  - 2.2.1 Game Theory 6
  - 2.2.2 Classification of Games 7
  - 2.2.3 Positional Games 7
  - 2.2.4 Examples of positional games 8
  - 2.2.5 Table with Implemented Board Games 15
- 2.3 Haskell and Web Assembly 16
  - 2.3.1 Asterius Haskell Compiler 16
  - 2.3.2 Foreign Function Interface 16

3 Method 18
- 3.1 Implementing board games 18
- 3.2 Generalizing 18
- 3.3 Developing a GUI 19
  - 3.3.1 Communication between JavaScript and Haskell 19
  - 3.3.2 GUI components 20
- 3.4 Distribution 20

4 Results 21
- 4.1 The Boardgame module 21
  - 4.1.1 The PositionalGame Type Class 22

4.1.2 Creating gameOver functions . . . 22
4.1.3 The ColoredGraph Type . . . 23
4.2 Using the Language . . . 24
4.3 Graphical User Interface . . . 26
4.3.1 Using the GUI library . . . 27
4.4 Example Games . . . 28
4.4.1 Hex . . . 28
4.4.2 M,n,k-game and Connect Four . . . 29
4.5 Distribution . . . 29

5 Discussion **30**
5.1 Meeting the purpose . . . 30
5.2 Getting started . . . 30
5.3 Important choices . . . 31
5.3.1 Haskell . . . 31
5.3.2 Web based GUI . . . 31
5.3.3 Maybe Maybe . . . 31
5.3.4 Modules . . . 32
5.4 Future work . . . 32
5.4.1 Expanding categories . . . 32
5.4.2 Strategy-stealing argument . . . 32
5.4.3 Threshold Bias . . . 33
5.4.4 vs. Computer and Online Multiplayer . . . 33
5.5 Sources . . . 33
5.6 Impact on society . . . 34

6 Conclusion **35**

Bibliography **36**

A Appendix **I**
A.1 Cross Code . . . I

List of Figures

2.1 A game of tic-tac-toe where X is the winner. 9
2.2 Dotted (red) edges have been deleted by **Cut**. Coloured (blue) edges have been coloured in by **Short**. The green nodes are the ones **Short** is trying to connect. . 10
2.3 A win for red in a game of Gale. 10
2.4 Blue closing off a move for red. 11
2.5 Hex game board with blue as the winner. 11
2.6 Havannah game board with the three possible winning configurations. 12
2.7 A win for red in a game of Y. 12
2.8 Game of Yavalath where blue lost, since they placed three pieces in a row before four. 13
2.9 A win for red in Cross. 13
2.10 A game of Connect Four with the red player as the winner. 14
2.11 A win for red in a game where n = 10 and k = 4. 14

4.1 The list of games available to play using the GUI. 27
4.2 The GUI for Shannon Switching Game on an arbitrary graph. 27
4.3 The GUI for Hex. 28

List of Tables

2.1 Overview of the implemented board games. 15

1 Introduction

Board games have long been an entertainment tool which contributes to fellowship with family and friends. Some board games even go beyond that, having their own designated world championships. As the world as we know it is slowly but surely transforming into a digitized society, it is in favour of board games to do so as well.

This project aimed to create an embedded domain-specific language with the purpose of developing digital board games. The language's objective is to give the user the ability to model board games and present them, making it possible to play said board games. The project also offers a set of example games for the user to play. The possibility to create your own games makes this product different when compared to regular, physical board games. It is not as easily achieved physically, as you are limited to the disposable material you have available, which will seldom be fit to create a structured and enjoyable board game.

1.1 Purpose

The purpose of this project was to develop an embedded domain-specific language that lets the user create digital board games. The language contains all the elements needed for modelling the board games inside the scope of this project. The purpose was fulfilled when a user could create a board game from the positional games category. The user must define the specific game logic themselves, but the language handles the overall structure and facilitates the implementation process. An additional goal was to provide a simple GUI base where the user can easily create a customized interface fitting for the implemented game.

1.2 Problem

The group chose to focus on positional games for this project. Positional games are a category of board games, and will be explained later in this report. This decision was based on the interest for positional games together with the suitability they have for a functional language like Haskell. This too, together with why Haskell was chosen, is explained more in-depth later on, namely in chapter 5.

Positional games have no hidden components, all information is visible for both players, which is an essential aspect as part of the scope is to create the product for single computer use. Deciding to focus on positional games was also important

as to not make the scope too big. Considering a smaller domain, namely only focusing on one board game category, makes the language easier to start building. It also makes it possible for more specialized functionality, which in turn makes the language easier to use.

The games implemented are deterministic and does not contain any elements of chance. The players fully decide their moves, provided that the rules of the game are followed. All games in this category ends with either **a)** player one winning, **b)** player two winning, or **c)** a tie. It is worth noting that not all games implemented allow for a tie, some of the games require a winner which can mathematically be proven.

1.3 Scope

When creating a new language it is necessary to impose limitations on the project to not make it impossible to manage. As this was a bachelor's thesis project, there was also a time limit to consider, 18 weeks in the case of this project. It was thus important to decide what to prioritize, and what ideas had to be scrapped. Below are the limitations the group imposed on themselves for the creation of this project.

- This project only intends to create a language that can implement **positional games**. The reasons for this were briefly explained in the previous section, and are further explored in chapter 2.
- No online play, no computer player — the games can only be played by two human players on the same computer.
- Initially, the plan was to implement only three board games using the language. Since it turned out to be quite easy to implement new games, this limit was removed.
- It should be possible to easily create a graphical user interface for the games, but the user needs to put in some effort to make it work—the language will not automatically modify the GUI when a new game is implemented.

1.4 Related Works

The idea of writing a language for the creation of digital board games is not unique. These types of languages and programs already exist, with one quite well-known example being **Tabletop Simulator**. What makes the language created in this project unique is not its complexity nor efficiency, but rather its strong functional language fundamentals. While Tabletop Simulator, for instance, is made in the game engine Unity [1], this work relies on Haskell's strict mathematical definitions and its lightweight deployability, which is more suitable for academic use.

There is also **Ludii**, a general game system designed to play, evaluate and design a wide range of games [2]. Ludii is not limited to board games, it can also be used for mathematical games, card games, dice games, and many more.

1.5 Structure

The report begins with a **theory** chapter, covering what an embedded domain-specific language is as well as the theory behind the board games that were implemented using the created language. The **method** chapter describes the process behind creating the language and using it to implement the aforementioned board games. The next chapter covers the **results** which present the language and GUI that was created, followed by a **discussion** of them. The report ends with a **conclusion**, summing up the work.

2

Theory

This chapter covers the theory behind the main components that were used in this project. The chapter begins with covering the method for implementing the language, a *domain-specific language*. Following this, the field of Game Theory is briefly covered. This leads up to choice of board game category, *positional games*. All games implemented using the created language are from that category and the rules for these games are covered next.

Since it is assumed that the reader has a basic understanding of Haskell, it is not covered independently in this chapter. The same assumption also applies to the other languages used. This includes, but is not limited to, TypeScript and ReactJS. However, WebAssembly, the Asterius Haskell compiler and what a Foreign Function Interface is will be briefly explained due to these programming tools being lesser known compared to the previously mentioned.

2.1 Domain Specific Languages

A domain-specific language (DSL) [3] is, in contrast to a general purpose [programming] language, a language restricted to a specific domain. The purpose of a DSL is to solve a problem in a specific domain. An example of a well-known DSL is HTML (HyperText Markup Language). It is used for designing documents to display in a web application, making web applications its domain. You cannot use it for, say, solving math problems.

In this project, the domain is positional board games. The language will for that reason only be useful for implementing digital board games in the positional game category since the functionality is specified to work for just that.

2.1.1 Why use a DSL?

There are plenty of good reasons to prefer working with a DSL over a more general approach. For starters, it's easier to analyze and safer to use as it narrows down the scope of tools you are working with. Because we have a clearly defined interface, it can also be less difficult to port to another technology if it is packaged correctly. Moreover, the main reason that it's so useful in the context of this project is that *it's easier to learn*. To create a board game using this project's DSL, the user doesn't have to learn advanced Haskell nor ReactJS. The user only needs to know the basics

and can thus rely on the backbone that has already been created.

2.1.2 DSL examples and their purpose

Besides reading documentation, the easiest way to understand what a DSL is and why it can be useful may be through examples of a few popular applications. Listed below are three useful implementations of DSLs, and why they are attractive in their respective domains.

DOT This first example is quite simple, DOT is purely a language for creating graphs. The input is plain text and the output is a mathematical graph, most commonly exported in vector format. Complexities range from simple flowcharts to a complex graph displaying intricate mathematical relations. One could unambiguously just draw their desired graph by hand or in an image editing program, but DOT allows for an organized, practical, and quite simple way to create the graph needed.

SQL A generally more well known DSL, used to insert, modify or extract data from relational databases, is Structured Query Language. This functionality could be implemented with a general purpose programming language, but with far more hardship. The primary motivation for the existence of SQL ought to be that the user can be skilled enough to create and manage complex databases entirely without being an adept software developer.

LaTeX The currently most intimate DSL that is used in this very thesis, is LaTeX. The document preparation software takes plain text as input together with its domain specific syntax to create structured and manageable academic documents. This is opposed to just writing and keeping the work in a plain text file, or approaching documentation by a “what you see is what you get” style which, for example, is used in Google Docs.

2.1.3 Embedded DSLs

The previously discussed domain specific languages are known as *external*, meaning they are implemented with an independent interpreter or compiler. An embedded (or *internal*) DSL (eDSL) [4]instead uses a general purpose language as a host. Simply speaking, it is a language inside another language. It is defined using the host language and can thus use the tools already implemented in said language. An example of an eDSL is jQuery, a JavaScript library. Compare the code for fetching all the elements of the class “example”. In pure JavaScript, you would use the following code:

```
document.getElementsByClassName("example");
```

If you instead use jQuery, the code is much simpler:

```
$('example')
```

The board game language created in this project was an **embedded domain-specific language mainly implemented in Haskell** together with a smaller graphical library implemented in ReactJS. Haskell and ReactJS are therefore the host languages and functionality from those are used.

2.2 Games

This section covers all the game rules of the implemented board games. It also provides some general game theory as well as a more detailed definition of what a positional game is.

2.2.1 Game Theory

Although this thesis does not cover the analysis of the mathematical aspects of games—however, it does briefly mention some aspects—here is a small overview and background of the field.

Game Theory has mainly been studied in the following three different branches of research [5, p. 705]:

- *Traditional Game Theory*, as pioneered by von Neumann et. al. concerning games with incomplete information. A prominent example is the game of Poker which involves concepts such as bluffing.
- *Nim-like games*, a kind of combinatorial games with complete information that can be decomposed into simple sub-games.
- *Positional games*, two player games with complete information and no random moves. Examples include Hex and Tic-tac-toe.

In this project the focus was on the third category. Another, more seldom used but perhaps better name for this category of games is *Tic-tac-toe-like* games. Although confusing, the more established term *Positional Games* will be used to refer to the category in this report.

It is interesting to note that the last two categories have largely been ignored by Traditional Game Theory and seen as "trivial" as evident, e.g., by this quote by von Neumann:

> *"Chess is not a game. Chess is a well-defined form of computation. You may not be able to work out the answers, but in theory there must be a solution, a right procedure in any position."* [6]

Of course, in practice, all but the most simple positional games are far from trivial. This is due to the combinatorial explosion of options at each move which makes examination of the whole game tree as a way to find an optimal solution totally infeasible. This property is what motivates the studying of these kinds of games.

2.2.2 Classification of Games

According to József Beck in *Inevitable Randomness in Discrete Mathematics* [7, p. x], games can be split into the following three "natural" categories:

1. Games of Chance (e.g. coin-flipping)
2. Games of Incomplete Information (e.g. Poker)
3. Games of Complete Information (e.g. Chess)

The first category is studied mainly in probability theory and statistics, and will not be covered in this report. The second category, due to its incompatibility with games played using only one monitor, is not covered either. As for the last category, a subclass of it is the main focus of this report.

2.2.3 Positional Games

Positional games are two-player games where the players take turns occupying previously unoccupied positions on the board. This group of games is a subcategory of abstract strategy games with complete information derived from Tic-tac-toe.

An *abstract strategy game* is a type of board game characterised by its focus on player's decision-making and most often lacks a theme, since it is not important for the playing experience. These games are also often combinatorial which makes them have no element of chance and no non-deterministic elements (such as rolling a dice).

The details will be explained further on, but one important aspect is that positional games are *static*, i.e. a claimed position on the board is permanent and cannot be unclaimed. This definition of static games specifically excludes some well-known abstract strategy games like Chess, Go, and Checkers from the category positional games.

All games implemented with this board game language are abstract strategy and combinatorial games of the subcategory positional games.

Definition

A positional game is defined as the following [8]:

- $\boldsymbol{X}$ - a finite set of elements. More commonly, X is called the *board* and the elements of $\boldsymbol{X}$ are called the *positions*.
- $\mathcal{F}$ - a family of subsets of $\boldsymbol{X}$. The elements of $\mathcal{F}$ are the sets that determine a win—the *winning-sets*.
- A winning criteria.

An example of a positional game is the classic Tic-tac-toe. The set $\boldsymbol{X}$ contains the 9 squares of the 3×3 game board, $\mathcal{F}$ contains the 8 possible lines that can be created by placing 3 pieces in a row, and the winning criteria is *"the first player to hold one winning set"* [9].

There exists several types of positional games, with different winning criteria and different rules. The three types that are relevant to this project are Strong Positional (Maker-Maker), Maker-Breaker and Avoider-Enforcer games. All games we look at except Yavalath fall into one of these three categories.

Strong Positional / Maker-Maker Game In a strong positional game, also called Maker-Maker game, both players' goal is to hold an element of $\mathcal{F}$, that is, one winning set. This results in both players being "makers". If all the positions have been taken, but no player holds a winning set, the game ends in a draw.

Maker-Breaker Game When it comes to Maker-Breaker games, the two players are given roles; one is the *Maker*, and the other is the *Breaker*. If a winning set is held by the Maker, the Maker wins. If the Breaker manages to prevent this from occurring, the Breaker wins. Draws are therefore not possible.

Avoider-Enforcer This family of games has the exact opposite (misère) winning criteria of the corresponding Maker-Breaker game for the same $\boldsymbol{X}$ and $\mathcal{F}$. The *Enforcer* wins if the *Avoider* claims an element of $\mathcal{F}$. However, if the Avoider manages to avoid claiming an element of $\mathcal{F}$, and all elements of $\boldsymbol{X}$ are claimed, he wins. When describing Avoider-Enforcer games, elements of $\mathcal{F}$ are thus called *the losing sets* instead.

Other Variants There exist other variants, but since they are not relevant to this thesis, they will not be covered here. If the reader wishes to learn about these, they may consult the standard literature on Positional Games[5] and read about the terms *"Discrepancy Game"*, *"Waiter-Client Games"* (also called *Picker-Chooser*) and *"Biased Positional Games"*.[10]

Connection Games

Sometimes another facet of the games we present are used to categorize them, independently of the positional game variety they fall into. In a *connection game*, the goal is to connect pieces together. A common goal is to connect two or more sides of a board by taking turns placing pieces, but this is not the only possible option. Forming a closed loop or connecting two nodes are other possible objectives. [11]

2.2.4 Examples of positional games

This section covers the theory behind the board games that were implemented using the created board game language as well as the ones implemented as the base to be able to create the board game language.

Figure 2.1: A game of tic-tac-toe where X is the winner.

Tic-tac-toe

Probably one of the most well known games, which is not surprising considering the numerous times it has already been mentioned in this report. It is played on a 3×3 grid where the two players take turns placing their pieces, typically X and O. The winner is the player who manages to place three of their pieces in a row, either horizontally, vertically or diagonally. The game can be played without much equipment, a pen and a paper being more than sufficient.

The game can either be played with unlimited pieces or a total of three pieces each. If there are unlimited pieces, unless someone has won, the game ends when all positions on the board have been filled. If each player instead only has three pieces, then the game can continue indefinitely since the players are allowed to move their pieces. However, if the players are allowed to move their pieces, it is no longer a positional game.

Tic-tac-toe can be further generalized to *m,n,k-games*. These games are played on a $m \times n$ grid with the goal of getting k pieces in a row. Using this definition, ordinary Tic-tac-toe is thus a 3,3,3-game. Another example is Gomoku, which is a 15,15,5-game since it is usually played on a 15×15 board where the goal is to place five pieces in a row. [12]

Category: Strong (Maker-Maker) Positional

Shannon Switching Game

The Shannon switching game is played on an arbitrary, finite graph. There are two special nodes in the graph, called A and B. The goal is for one player—the *Maker*, called **Short**—to connect A and B by coloring edges. The goal of the other player—the *Breaker*, called **Cut**—is to prevent this, by deleting non-colored edges. Short wins when the two special nodes have been connected. Cut wins when there is no possible way for Short to connect the two nodes. [13]

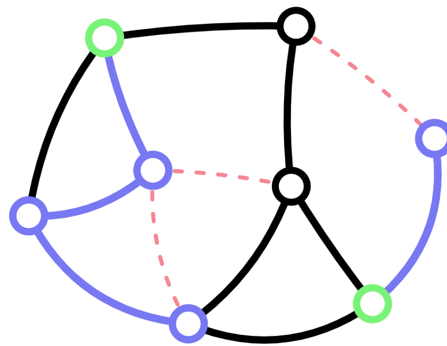

Figure 2.2: Dotted (red) edges have been deleted by **Cut**. Coloured (blue) edges have been coloured in by **Short**. The green nodes are the ones **Short** is trying to connect.

Categories: Positional Maker-Breaker; Connection

Gale

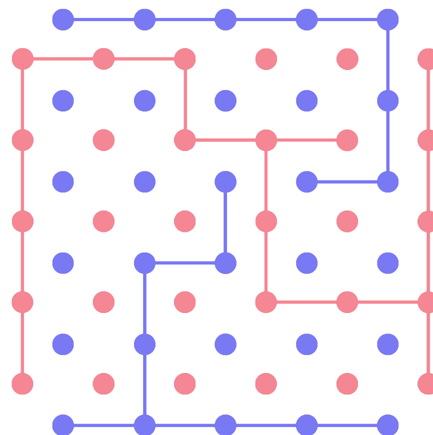

Figure 2.3: A win for red in a game of Gale.

Gale is a variant on the Shannon switching game. Instead of being played on an arbitrary graph, the game is played on grids. The two grids are offset to each other and have different colors. One player's goal is to connect the left and right side by placing edges, while the other's goal is to connect the top and bottom. With the right strategy, player one can always win. [14]

It might not be obvious how Gale fits into the Maker-Breaker category since, after all, both players are trying to make a path between their respective sides, thus appearing as a Maker-maker game. But since the grids are placed on top of each other, for every move a player makes, a possible move for the other player is closed off. If we look at figure 2.4, the blue player has made a move that makes it impossible for the red player to make the move that is the dashed line. This and the fact that when the board is filled someone has to have won makes Gale a Maker-Breaker game.
Categories: Positional Maker-Breaker; Connection

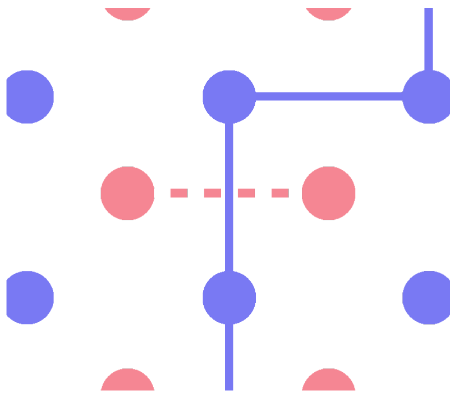

Figure 2.4: Blue closing off a move for red.

Hex

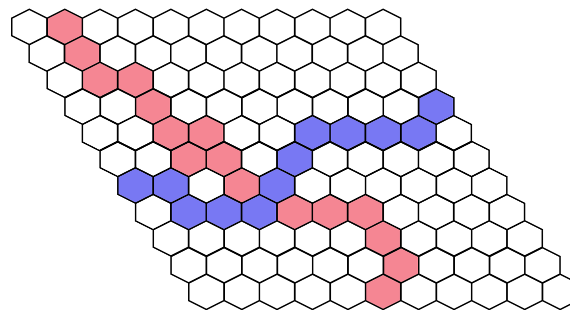

Figure 2.5: Hex game board with blue as the winner.

Hex is conventionally played on a 11×11 board. What makes it different is the hexagonal shape of each position, making the board appear as if it is leaning. Taking turns placing the pieces, the players' goals are to connect their two sides. [15]

Interesting enough, Hex cannot end in a tie. This is due to the fact that at most three fields meet in each spot. The proof showing that no game of Hex can end in a draw is equivalent to Brouwer's fixed point theorem which was shown by Gale in 1979.[16]

Categories: Positional Maker-Breaker; Connection

Havannah

Havannah is another game from the connection game family. It is based on Hex and is played on a hexagonal board with either 8 or 10 positions on each side. There are three ways to win the game. These can be seen in figure 2.6, and are listed with explanations below.

- **Ring** (blue) - A loop around at least one cell.
- **Bridge** (green) - Connecting two corners.
- **Fork** (red) - Connecting three edges. It is not allowed to count the corner as an edge.

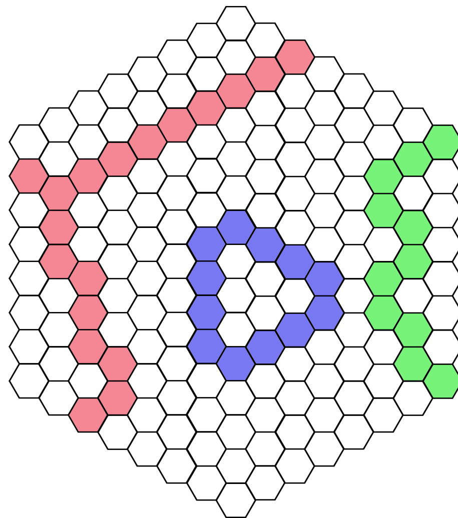

Figure 2.6: Havannah game board with the three possible winning configurations.

Categories: Positional Maker-Maker; Connection

Y

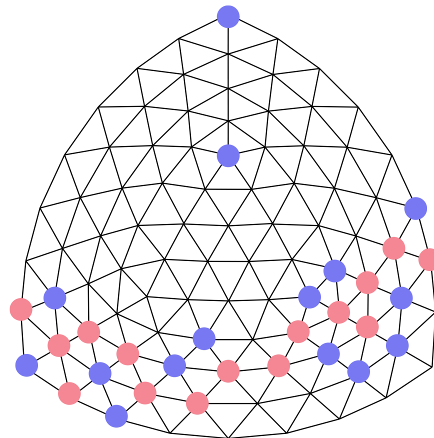

Figure 2.7: A win for red in a game of Y.

The game called Y is a descendant of Hex, and just like Hex it cannot end in a tie. It is played on a triangular board with hexagonal shaped spaces. The objective is to connect all the three edges using your pieces. Placing a piece in a corner counts as having connected two sides. When the board is small, tactic is more prominent while on larger boards, strategy is. A larger board offers the possibility for a longer game, which is why strategy is more useful since it reflects your long-term goals and how to achieve them, namely winning. Tactics being revolved around short time frames and small steps is therefore naturally more effective on small boards. [17]

Categories: Positional Maker-Maker; Connection

Yavalath

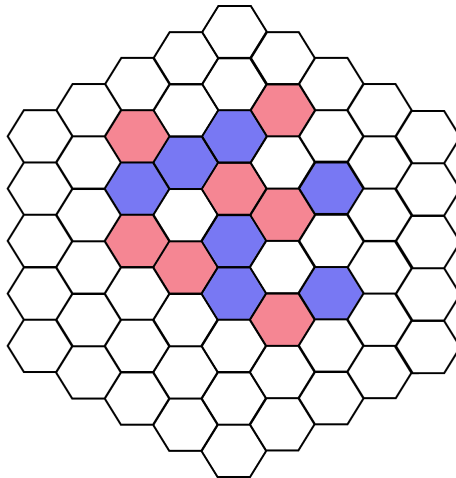

Figure 2.8: Game of Yavalath where blue lost, since they placed three pieces in a row before four.

Each player's goal in Yavalath is to place four pieces in a row. However, if three pieces are placed in a row before that, the player loses. Yavalath is another game that is played on a hexagonal board, with 5 spaces on each side. What makes Yavalath special is the fact that it was designed by a computer program instead of a human.[18]

Because the players are both trying to capture and avoid sets, it is neither a Maker-Maker, Maker-Breaker nor an Avoider-Enforcer game.

Categories: Positional

Cross

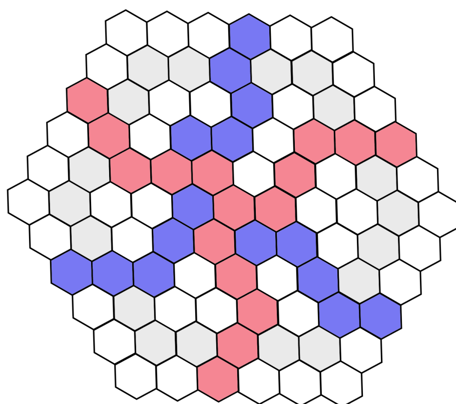

Figure 2.9: A win for red in Cross.

Another game played on a hexagonal board, Cross has a size of 2 to 10 pieces per side where 6 is the standard size. The first player to connect three non adjacent

sides with their pieces wins. Just like in Y, placing a piece in a corner counts as connecting the two sides. If a player connects two opposite sides while not connecting three adjacent sides in the same move, the player loses.

Categories: Positional; Connection

Connect Four

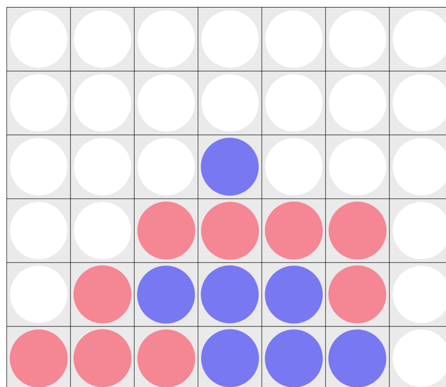

Figure 2.10: A game of Connect Four with the red player as the winner.

Connect four is similar to m,n,k-games, with the board being 7×6 big and the objective being to connect four pieces in a row. The difference from regular m,n,k-games is that the board is flipped 90°, making it stand on its smallest side. This in turn adds the constraint of only being able to place pieces at the lowest level, and in order to place on higher levels, the positions below must be occupied. While trying to connect four of their own pieces, the player must also prevent the opponent from doing the same.

Categories: Maker-Maker Positional; Connection;

Arithmetic progression game

1 2 3 4 5 6 7 8 9 10

Figure 2.11: A win for red in a game where n = 10 and k = 4.

In the Arithmetic progression game (APG), two players take turns picking numbers on the board that is the number axis from 1 to n. The first player to create an arithmetic progression of length k wins. The number k must be less than n. It is also possible to play the game as a Maker-Breaker version, where the Breaker's objective is to hinder the Maker from creating an arithmetic progression.

Categories: Maker-Maker Positional

2.2.5 Table with Implemented Board Games

Below is a table containing all the board games that have been implemented using the language. It also shows how each board game relates to $\boldsymbol{X}$, $\mathcal{F}$, and winning criteria.

Name	$\boldsymbol{X}$	$\mathcal{F}$	Winning Criteria
Tic-tac-toe	The 9 squares of the 3×3 game board.	The 8 lines that can be created by placing 3 pieces in a row.	The first player to hold one winning-set.
Shannon	All edges of a graph.	All paths between two distinguished vertices s and t.	Short wins if a path between s and t is created, cut wins if all edges are claimed and no path is present.
Gale	All edges of a squared graph. Each player has their own graph.	All paths connecting the left and the right side on the first player's grid.	Player wins if they can hold one winning set. If the first player cannot hold a winning set player two wins
Hex	The hexagonal cells on a squared board, commonly 11×11.	All paths between the sets of positions at opposite sides for the first player (Maker).	Maker wins if a path between two opposite sides is created, cut wins if all positions are claimed and no path is present.
Havannah	The hexagonal cells on the hexagonal board, often 8 or 10 cells on each side.	Ring, Bridge and Fork.	The first player to hold one winning-set.
Y	All hexagonal spaces in the triangular board.	All paths between the three sides.	The first player to hold one winning-set.
Yavalath	The hexagonal cells on the hexagonal board, 5 cells on each side.	All combinations of four or three in a row.	The first player to hold one winning-set with size 4 wins if not then the first player to hold one winning-set with size 3 loses.
Cross	The hexagonal cells on the hexagonal board, generally 6 cells on each side.	All paths between three adjacent sides.	The first player to hold one winning-set.
Connect Four	All spaces in a 7×6 board,	All combinations of four in a row, vertically diagonally and horizontally.	The first player to hold one winning-set.
APG	The set 1,..,n.	All arithmetic progressions of length k.	The first player to hold one winning-set.

Table 2.1: Overview of the implemented board games.

2.3 Haskell and Web Assembly

There are quite a few ways to develop a GUI front-end for a model written in Haskell, one of the most portable ways being with the use of WebAssembly. WebAssembly (WASM) is a specification of a binary-code format for executable programs that, rather than running natively on a machine, runs interpreted or compiled in web browsers [19].

With a Haskell model compiled to WASM, a GUI front-end can be developed with the assistance of regular web technologies such as HTML, CSS, and JavaScript, while still utilizing the full potential of the Haskell model. Unfortunately, compiling when targeting WASM is rather different from targeting, for instance, a 64-bit Windows machine. As luck would have it, a company named Tweag I/O has developed a compiler named Asterius for just this purpose.

2.3.1 Asterius Haskell Compiler

The Tweag I/O developed Asterius Haskell Compiler (AHC) is meant to be a drop-in replacement for the more traditional Glasgow Haskell Compiler (GHC) used to compile Haskell to WASM [20].

Asterius comes with a number of tools that can be used instead of their traditional counterparts when developing WASM targeting projects. One such tool is AHC-Cabal, which is a replacement for the habitual Cabal. Cabal is the package and project management tool used in many Haskell projects. Asterius does not only produce `.wasm` files, but also `.js` files that can be included in a web page to load the `.wasm`. These JavaScript files are used both to initialize the WASM module and to facilitate the communication between Haskell and JavaScript.

2.3.2 Foreign Function Interface

To allow communication between Haskell and JavaScript, Asterius makes use of a Haskell feature called Foreign Function Interface, or FFI for short. FFI in Haskell is usually utilized to allow Haskell code to call, for example, C functions, but in the case of Asterius it is used to execute JavaScript expressions. The syntax for using FFI is displayed in listing 1.

```
foreign import <FFI_type> "<imported_name>" <haskell_name> :: <type>
```

Listing 1: The syntax for FFI in Haskell.

The syntax for FFI in Haskell, shown in listing 1, contains the following four variable parts.

- `<FFI_type>`: An identifier for the language of the imported function. In Asterius' case this should be `javascript`.
- `"<imported_name>"`: This is a string with the name of the imported function. For Asterius, a JavaScript expression should be provided. When dealing with

arguments from Haskell, `"$n"` can be used in the expression to denote the n-th argument (starting from 1).

- `<haskell_name>`: Is a new name of the function to be used from the Haskell code.
- `<type>`: Is the Haskell type of the imported function. Asterius requires that the result of the function is some `IO a`.

Another feature of FFI is the `safe` keyword which can be placed after the `<FFI_type>` to execute the foreign function synchronously. In Asterius this works with JavaScript expressions that return a `Promise`, which is then `await`ed before the called Haskell code is resumed.

Asterius features a special `foreign import` function called `"wrapper"`, that turns a Haskell function into a function that can be called from JavaScript. This can be used to create callbacks, or to invoke Haskell from JavaScript.

3
Method

The first parts of the development process were split in a few loosely defined modules. Initially, a few command line based board games were implemented in Haskell to get a foothold in the task at hand. Once those games were developed to satisfaction, the next step was to find similarities between the games. This part was crucial as it gave a hint of what code could be generalized, and what code could not. A GUI was also implemented with the aim to function for one game in the first instance, and hopefully an assortment of games later on. As the comprehension of the task widened in the development process, more games were implemented with a progressive sense of ease. This entire process is described in more detail in this chapter.

3.1 Implementing board games

Research about what games to implement was conducted at the beginning of the project. This was done in order to decide what game categories to focus on. After some consideration, the final choice landed on *positional games*, with a focus on a few of its subcategories.

The commencing work of programming within the frames of the project was aimed at implementing a few specific games. The games chosen were *Tic-tac-toe*, *Shannon Switching Game* and *Hex*. These were chosen because at the time, they were considered fairly similar but as knowledge grew about games within the positional games category, the group came to realize that there are games even more similar to each other.

The three games were virtually implemented in complete isolation from each other. No elaborate thinking was done about how to proceed from this point as the objective was simply to make the games work. Once they were all working separately, the process of generalizing the code began.

3.2 Generalizing

The generalization started with the first implemented games. Similar aspects shared between the games were identified in order to build a language as abstract as possible yet still functional with the different games. A key aspect was identifying that almost all games in the positional game category could be represented by a graph. The games can then be played either on the vertices or the edges. Thus, the

`ColoredGraph` data type was created.

New data types were implemented to describe important aspects of the game. These were the players, a position on the board, and the outcome of the game. Having these data types made the language easier to work with and understand, but they were not necessary to generalize the code as Haskell's already existing types could work as well.

To describe a game of any type, a type class called `PositionalGame` which takes the game type (Havannah, Tic-tac-toe, etc.) as an argument was created. This class contains a variety of different functions useful for a positional game, and any game of that type class could therefore use these functions.

During the process of generalizing the code, more games were implemented. Now, instead of implementing them from scratch, the language was used. As the code was refactored many times over, more games naturally had to be adjusted as a result of having an ever increasing number of games implemented. Interestingly enough, all the three games that were implemented in the beginning were implemented a second time in a different way as a result of the generalization.

3.3 Developing a GUI

After some discussion about what method should be applied regarding the graphical implementation of the project, the decision landed on a web oriented approach. This means that if the user wants to customize their created board games, they only need a skill set of relatively basic web design tools instead of a more advanced skill set of, for instance, 2D or 3D graphics rendering. The following subsections describe the steps taken to deploy a graphical user interface (GUI) for a developed Haskell board game.

3.3.1 Communication between JavaScript and Haskell

As discussed earlier in section 2.3.2, the compiler Asterius outputs a simple JavaScript file that acts as the "glue" between JavaScript and Haskell. However, from the JavaScript side, Asterius does not provide any help for interacting with Haskell. To help users of the language with the JavaScript side, an accompanying JavaScript library was developed.

This JavaScript library delivers an interface, bound to the JavaScript global `window`, that makes for easier interaction with the Haskell back-end. The interface contains a series of events that can be used to react to state changes in the games, such as when the game is over or when a new input, such as a move, is expected. It also has a function for passing moves to Haskell, and a collection of the games that is available from the back-end.

3.3.2 GUI components

To further help users of the language develop GUI's, a separate, but compatible, JavaScript library with GUI components was developed. This is a ReactJS[1] base library which contains all the necessary parts to build a complete GUI for the games. While the components are all functional, they have no default styling. Because of this, users can — and must — style them themselves, giving the GUI their own look and feel.

There is also a component for displaying the current state of the game, available for games implemented with the `ColoredGraph` data type as a base. With this, all the user has to do is define what move should be made when an end-user clicks on a node or edge in the graph. While the default look of the graph can work for some games, the user can style it themselves either with easy-to-use templates or a fully custom style with the use of JavaScript's Canvas API.

3.4 Distribution

To make the process of using the language and GUI library as easy as possible, they were packaged and uploaded to a package repository. For both the language and the GUI library this involved writing a well structured package description, a `.cabal` file and a `package.json` respectively. These descriptions contain names, descriptions, etc, and more importantly, they list dependencies. Dependencies are, in both cases, listed as ranges of versions of other packages that the dependant can work with, this is important for building dependency trees.

Uploading a package to npm (for the JavaScript library) is as simple as registering an account and pushing a button. Hackage (for the language) is a bit more involved. Hackage has a vetting of uploaders to make sure only serious packages are uploaded. Thankfully, this was not a problem for the board game package.

[1] The JavaScript library ReactJS `https://reactjs.org/`

4

Results

The finished product, namely the language, will be presented in detail in this section. The games implemented share the same type class and all the boards are represented by an underlying graph data structure. For almost all functions needed, standard implementations are provided due to the similarities between the different games. An exception is the function specifying the winning criteria of the game. However, the games generally still fall under one of the main classes of positional games mentioned in section 2.2.3, knowledge which is used to simplify the implementation of the winning criteria as well.

This section also shows the finished look of the GUI, a closer look at some of the games, and a step by step explanation of how to go about implementing a game with the language. Furthermore, all the code discussed in this chapter can be found on the GitHub page for this project, `https://github.com/Boardgame-DSL`. There is also a website with documentation and playable example games available at `https://boardgame-dsl.github.io/`.

4.1 The Boardgame module

This module contains everything needed in order to implement a game. Firstly, it contains three data types that are the core of any game.

```
data Player = Player1 | Player2
```

Represents one of the two players.

```
data Position = Occupied Player | Empty
```

A 'Position' can either be 'Occupied' by a 'Player' or 'Empty'.

```
data Outcome = Win Player | Draw
```

A game ends with either a 'Win' for one of the players, or a 'Draw'.

Secondly, it contains a wide variety of functions that are useful for defining the specifics of a game. Some examples are functions for combining winning criteria, checking if a player has lost, and checking if a position is occupied. Lastly, it contains the `PositionalGame` type class.

4.1.1 The PositionalGame Type Class

```
class PositionalGame a c | a -> c where ...
```

A game in the type class `PositionalGame` is described by `a`, the type of the implemented game, and `c`, a type to describe a position on said game's board (the set **X**). For simplicity, `c` is often called a coordinate. Initially, almost every game had to implement its own specific version of the five functions defined in the type class.

```
makeMove :: a -> Player -> c -> Maybe a
```

The function `makeMove` and its standard implementation `takeEmptyMakeMove` is a direct realization of the core concept of a positional game: The players take turns *claiming previously unoccupied elements of X*.

```
takeEmptyMakeMove :: PositionalGame a c => a -> Player -> c -> Maybe a
takeEmptyMakeMove a p coord = case getPosition a coord of
    Just Empty -> setPosition a coord p
    _          -> Nothing
```

```
getPosition :: a -> c -> Maybe Position
```

Returns which player (or Empty) has taken the position at the given coordinate, or 'Nothing' if the given coordinate is invalid.

```
setPosition :: a -> c -> Position -> Maybe a
```

Takes the position at the given coordinate for the given player and returns the new state, or 'Nothing' if the given coordinate is invalid.

```
positions :: a -> [Position]
```

Returns a list of all positions, not in any particular order.

```
gameOver :: a -> Maybe (Outcome, [c])
```

The function `gameOver` is what defines the specifics of a certain positional game. It takes a `PositionalGame` *a* and returns the result if the game is finished and `Nothing` if it has not. The result can either be a win for a player, or a draw. If the function returns a result, it can also return a list of relevant coordinates which can be used to explain the outcome of the game.

4.1.2 Creating gameOver functions

There are three functions useful for constructing each games' `gameOver` function. These functions are based on winning sets, and correspond to the three main classes of positional games as described in section 2.2.3: `makerMakerGameOver`, `makerBreakerGameOver` and `avoiderEnforcerGameOver`.

```
makerMakerGameOver :: (Eq c, PositionalGame a c)
    => [[c]] -> a -> Maybe (Outcome, [c])
```

This function simply makes the game continue until one player holds a winning set, or all the board's positions are filled. If someone wins, the function also returns the winning set that led to a win as the relevant coordinates.

```
makerBreakerGameOver :: (Eq c, PositionalGame a c)
    => [[c]] -> a -> Maybe (Outcome, [c])
```

This function checks if player 1 (maker) holds a winning set, if so, they win. It also checks if player 2 (breaker) holds at least one element in every winning set since by doing so, player 1 cannot possibly win and ultimately, player 2 wins instead. Choosing relevant coordinates to return, when player 1 wins one of the winning sets he controls is returned. When player 2 wins, a minimum set of coordinates he owns, that he needs in order to win, is returned. For example, one of our implementations of Hex utilizes this, which is described in section 4.4.1.

```
avoiderEnforcerGameOver :: (Eq c, PositionalGame a c)
    => [[c]] -> a -> Maybe (Outcome, [c])
```

This function does the same as makerBreakerGameOver except it swaps who wins since the goal is not for player 1 to hold a winning set, but to avoid it.

4.1.3 The ColoredGraph Type

Most positional games can be represented using a graph data structure. For this purpose, a type with "colors" (a property) for both edges and vertices was implemented. This type is called `ColoredGraph`:

```
type ColoredGraph i a b = Map i (a, Map i b)
```

It is simply a `Map` which has a game's board's vertex (specified as a "coordinate", `i`) as key. The mapping has a tuple as its value with the first element representing the vertex color `a`. The second value of the tuple is a mapping from target vertex `i` to edge color `b`. This mapping in effect represents how the vertices in our graph structure are connected together by edges.

The `ColoredGraph` type is located in the Boardgame.ColoredGraph module. This module has a number of helper functions for games modelled as graphs as well as functions for creating graphs of different shapes and with different properties. Here follows an example of a function that creates a graph:

```
paraHexGraph :: Int -> ColoredGraph (Int, Int) Position (Int, Int)
```

This function creates a parallelogram shaped graph of hexagon vertices. Each vertex has edges to its neighbours in a hexagon tiling and the input decides how many vertices each side has.
In the module there are also a few standard functions that can be used when defin-

ing the functions from the PositionalGame type class. These are named after the function they implement, prefixed with *coloredGraph* together with how they are implemented. For games represented as a `ColoredGraph` played on vertices, the following function can be used to implement `positions`:

```
coloredGraphVertexPositions :: (ColoredGraphTransformer i a b g, Ord i)
    => g -> [a])
```

To use these functions, the game must create an instance of `ColoredGraphTransformer`. Using this results in a cleaner and more concise code.

4.2 Using the Language

To create new games using the language, three things are generally needed:

1. A new type for the game that is to be implemented.
2. An instance of a `PositionalGame` of the newly created type.
3. A function for creating an empty board of the game.

The easiest way to learn how to use the language is by studying one of the implemented example games. Below, we will take a look at the implementation of the game **Cross**. The rules for this game can be found in section 2.2.4, and the full code (including necessary imports) in appendix A.1.

Step 1. The new type can be created using the `ColoredGraph` type.

```
newtype Cross = Cross (ColoredGraph (Int, Int) Position (Int, Int))
```

Step 2. In the instance of a `PositionalGame`, the five functions mentioned in section 4.1.1 are defined following the rules of the game implemented.

```
instance PositionalGame Cross (Int, Int) where
    positions   (Cross b)     = values b
    getPosition (Cross b) c   = fst <$> lookup c b
    setPosition (Cross b) c p = if member c b
        then Just $ Cross $ adjust (\(_, xs) -> (p, xs)) c b
        else Nothing
    makeMove = takeEmptyMakeMove
```

As stated, several games can use the same implementation for the functions above, but the `gameOver` function almost always needs to have a specific definition for each game.

```
gameOver (Cross b) = criterion b
  where
    criterion =
      -- It's a draw if all tiles are owned.
      drawIf (all isOccupied . values) `unless`
      -- Here we say that in any position where one player wins,
      -- the other player would win instead if the pieces were swapped.
```

```
          symmetric (mapValues $ mapPosition nextPlayer)
          -- You lose if you have connected 2 opposite sides.
          (criteria (player1LosesWhen <$>
            [ anyConnections (==2) [side1, side4]
                . filterValues (== Occupied Player1)
            , anyConnections (==2) [side2, side5]
                . filterValues (== Occupied Player1)
            , anyConnections (==2) [side3, side6]
                . filterValues (== Occupied Player1)
            ]) `unless`
          -- You win if you have connected 3 non-adjacent sides.
          criteria (player1WinsWhen <$>
            [ anyConnections (==3) [side1, side3, side5]
                . filterValues (== Occupied Player1)
            , anyConnections (==3) [side2, side4, side6]
                . filterValues (== Occupied Player1)
            ]))

    -- A list of coordinates for every side based
    -- on which neighboring tiles are empty.
    [side1, side2, side3, side4, side5, side6] =
      missingDirections b <$> [[hexDirections !! i,
          hexDirections !! ((i+1) `mod` 6)] | i <- [0..5]]
```

This implementation of the `gameOver` function uses several helper functions defined in the **Boardgame** and **ColoredGraph** modules.

`isOccupied :: Position -> Bool`
Checks if the position is occupied or not.

```
symmetric :: (a -> a) -> (a -> Maybe (Outcome, [c]))
          -> (a -> Maybe (Outcome, [c]))
```
Create a symmetric `gameOver` function from a `gameOver` function defined for only one player.

`criteria :: [a -> Maybe (Outcome, [c])] -> a -> Maybe (Outcome, [c])`
Combines several `gameOver` functions into one. Their results are prioritized by their order in the list.

`player1LosesIf :: (a -> Bool) -> a -> Maybe (Outcome, [c])`
If the predicate holds, a winning state for player 2 is returned. If not, a "game running" state is returned.

`player1WinsIf :: (a -> Bool) -> a -> Maybe (Outcome, [c])`
If the predicate holds, a winning state for player 1 is returned. If not, a "game running" state is returned.

```
unless :: (a -> Maybe (Outcome, [c]))
       -> (a -> Maybe (Outcome, [c]))
       -> (a -> Maybe (Outcome, [c]))
```

Combines two `gameOver` functions into one where the first function's result is returned, unless the second criterion returns a `gameOver` state.

```
anyConnections :: Ord i => (Int -> Bool) -> [[i]]
               -> ColoredGraph i a b -> Maybe [i]
```

For every component of the graph, count how many groups of nodes they overlap with and check if the predicate holds on the count. If it matches on any component then return that component. We also try to return only the parts of the component that are necessary for our predicate to hold.

```
filterValues :: Ord i => (a -> Bool) -> ColoredGraph i a b
             -> ColoredGraph i a b
```

Filters out any vertices whose value is not accepted by the predicate.

```
missingDirections :: (Ord i, Eq b) => ColoredGraph i a b -> [b] -> [i]
```

Return a list of all nodes which do not have neighbours in any of the specified directions.

```
hexDirections :: [Coordinate]
```

The six directions of neighbours on a hexagonal grid.

Step 3. After that the empty board can be implemented. The `hexHexGraph` in this example creates a hexagonal shaped board with hexagonal board positions, which is the board needed for Cross, Yavalath and Havannah amongst others.

```
emptyCross :: Int -> Cross
emptyCross = Cross . hexHexGraph
```

4.3 Graphical User Interface

For each of the implemented board games, a GUI was created. The GUI can be seen in action on the website linked at the start of this chapter. The user is first met with a list of all the available games, as seen in figure 4.1. Clicking on the text for one of the games starts and displays the board for said game. The game can always be restarted or exited by using the buttons in the top left corner. A small text also tells the user which players' turn it currently is.

The two players are represented by the colors red and blue respectively. Depending on which game is played, the user can simply click on either a tile on the board, or a line in the graph, in order to make a move. When a player has won, or a draw has occurred, a text providing the result is displayed on top of the board. The pieces that make up the winning set also become larger when the game is over to show which moves led to the win.

Figure 4.1: The list of games available to play using the GUI.

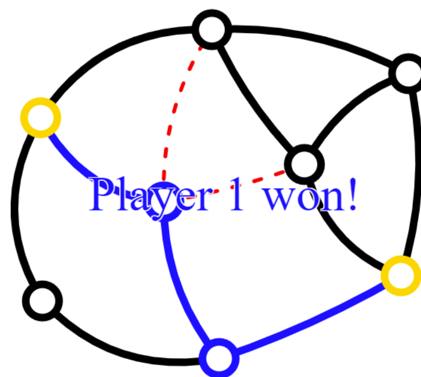

Figure 4.2: The GUI for Shannon Switching Game on an arbitrary graph.

4.3.1 Using the GUI library

The GUI library comes with a JavaScript object necessary for interaction with the Haskell model. This object also contains a few additional properties that aim to make the use of it easier.

This object can be used to start games, listen to events, input moves, and more.

```
window.boardgame.games.TicTacToe();
window.boardgame.addEventListener("state", updateState);
window.boardgame.inputMove([1, 1]);
```

In addition to this necessary "glue" object, the GUI library contains a suite of React components that can be used to more easily build a fully functional GUI.

A GUI built in this way would work out-of-the-box for all games modeled with

Figure 4.3: The GUI for Hex.

a `ColoredGraph` in the Haskell model. Although, without customising the `ColoredGraphDisplay`, by extending it and overriding some of its methods, the GUI would look quite bland.

```
<div className="controls">
        <StartButton/>
        <TurnDisplay/>
</div>
<ColoredGraphDisplay/>
<GameOverStatus/>
<NotificationArea/>
```

4.4 Example Games

This section discusses some of the implemented board games provided as examples for how to use the language. Most of the games follow the structure mentioned in section 4.2, but some require additional code, or don't work with the `ColoredGraph` type, and this section aims to explain some of these games further.

4.4.1 Hex

There are two implementations of Hex which are the same except for the code in their `gameOver` functions. The first version checks if the game is over by directly examining the graph the game is played on. The game ends if player 1 has a path from left to right or if player 2 has a path from top to bottom. The second version implements the `gameOver` function by constructing all the winning sets for player 1 (i.e. the minimal set of paths from left to right) and passes them to the function `makerBreakerGameOver`. This function then uses the winning sets to construct a `gameOver` function.

As both implementations follows the rules of Hex, they are equivalent when playing,

though one might still prefer one version over the other when it comes to writing the code. In the first version, both players' winning conditions have to be specified while in the second version, player 2 wins automatically when player 1 can't own any of the winning sets. Sometimes it can also be easier to write a function that checks if someone has won instead of constructing all the winning sets, and vice versa. Finally, one big drawback with the second implementation is that it can take a long time for the language to construct all the winning sets for large board sizes, which it has to do every time we want to check if someone has won.

4.4.2 M,n,k-game and Connect Four

M,n,k-games is played on vertices and implements the `ColoredGraphTransformer`. The game can therefore use the standard functions already implemented and the only function that needed to be implemented was the `gameOver` function.

```
instance ColoredGraphTransformer (Int, Int) Position String MNKGame
```

Due to the similarities with m,n,k-games, we could use the code that was already implemented to create Connect Four as well. The game could thus be called using the function for m,n,k-games with the specific dimensions, a 7×6 board and k as 4, being the variables. Connect Four is one of the few games that cannot use the standard implementation of the function `takeEmptyMakeMove`, and a new version of that had to be implemented in order to accommodate the constraint the game has. The constraint is that only the positions at the bottom can be picked, and that positions at higher levels only can be picked if the position below is occupied as well.

4.5 Distribution

The language was uploaded to Hackage, under the name `boardgame`[1], and can be used by adding its name to the dependencies list in a projects cabal file. The JavaScript library was uploaded to npm, under the name `@boardgame-dsl/boardgame`[2]. It can be installed to a project using the following command:
`npm install --save-dev @boardgame-dsl/boardgame`.

[1] `https://hackage.haskell.org/package/boardgame`
[2] `https://www.npmjs.com/package/@boardgame-dsl/boardgame`

5 Discussion

In this section, the result achieved from this project is discussed. Firstly, an opinion on the finished product as well as the obstacles and problems encountered is presented. Secondly, the important choices that had to be made, and how they impacted the project is discussed. This includes; why Haskell was chosen as the primary language; why a Web based GUI was decided as the method for implementing the GUI; the decision to refactor the code by removing “Maybe (Maybe xx)”; and the decision to split the games into different modules. After that, future work is suggested such as expanding to more game categories, the strategy stealing argument and adding online multiplayer. Lastly, the impact the project has on society is briefly discussed.

To reiterate, the purpose of the project was to *develop an embedded domain-specific language that lets the user create digital board games*. Assisted by the results stated in the previous section, an attempt to give our interpretations and assess where the project stands in terms of fulfilling the purpose will now be given.

5.1 Meeting the purpose

It is believed that the purpose has been achieved with this project. Using the language makes it significantly easier to create any game in the positional game category due to the provided game structure and many useful functions that the language supplies. The language is not suitable to use by individuals who have no prior knowledge of Haskell since it is, after all, embedded in Haskell and uses its functionality. However, the language already has a variety of games implemented that can undoubtedly be played by almost anyone.

5.2 Getting started

One problem we encountered was simply where to start. The description of the task, to create a language for board games, is very broad and it was clear that the scope had to be narrowed down. We decided to look for three games or groups of games to focus on and it was at that point the decision landed on positional games. Still, we struggled on how to proceed but ultimately determined that the group would be divided into two subgroups, each subgroup being assigned one specific board game to implement. From there on, the project was set in motion.

5.3 Important choices

Several choices were made during the process of creating the board game language. In this section we will discuss the reasons why we made some choices and how they impacted the project. The choices we will discuss are; why we picked Haskell as our host language, our decision to go for a web based visual presentation, the decision to create more data types, and dividing the language into modules.

5.3.1 Haskell

There are many reasons why Haskell was the most suitable choice for this project. Firstly, it is good for creating a domain-specific language since it is a functional and pure language. Being a functional language means that everything is built up of functions, and being pure means that it has referential transparency. This makes it possible at a glance to see similarities between different parts of your program so that you can extract them into a single function that can be reused later. This is in contrast to an imperative language where some part of your code can have unwanted side effects which makes it hard to extract into a separate function. Haskell also offers good abstraction opportunities and makes it easy to build data structures, both aspects are important for our project. Lastly, positional games have a precise mathematical definition, and Haskell is useful for mapping mathematical properties. [21]

5.3.2 Web based GUI

Regarding the visual presentation of the board games, we narrowed our alternatives down to two very different approaches. The former option was to develop a desktop app in native Haskell, using simple 2D graphics or a more advanced 3D procedure, using something like Open Graphics Library[1]. The latter alternative, which we found much more preferable than the former, was to compile the Haskell source code into a format that can be used in JavaScript. The web based approach turned out to be a clear victor in each crucial step in the development process. Using the Foreign Function Interface, compiling with the AHC compiler and developing a web interface with ReactJS allows a much more streamlined implementation, mainly due to the fact that you don't need any advanced computer graphics skills to render the simplest of board games.

5.3.3 Maybe Maybe

Another important choice we made was creating more data types than we initially had made. After a suggestion from our supervisor, that having "Maybe (Maybe xx)" in our type signature is bad practice, we replaced some of our Maybes with data types we created. Although resulting in us having to implement functions that otherwise are already created for the Maybe type, the decision makes it significantly easier for the intended user to work with the code.

[1]The open source API for rendering 2D and 3D graphics `https://www.opengl.org//`

5.3.4 Modules

As with the choice to switch out "Maybe (Maybe xx)", deciding to divide all the games into separate modules was also done following recommendation from our supervisor. At the beginning, all games were in one big file named main where also the code for running the games were placed. As the number of games grew, the need to divide the code became all the more obvious. The code for each game was thus moved to their own separate files. Games that have more than one version implemented naturally have all their code collected.

However, the result of this decision was more code. Many of the games need the same properties imported, which they could share when the games were all placed in the same file. When the games were moved into separate files, the imported properties had to be repeated in those files to accommodate each and every game's needs. Each game file also had to be imported into the main file in order for the run game function to be able to initiate each game. Although there were some evident drawbacks, we still deemed that the cleaner structure and easier locating of each game's code were more valuable than the extra code that had to be added.

5.4 Future work

As with most things in life, there is always more work to be done. The creation of a new language is not an exception. If we had more time, there are several things we would have liked to add to the language. This section discusses those.

5.4.1 Expanding categories

The first, and possibly most obvious, thought on how to further develop the project is to extend it to more categories of board games. The ultimate goal would naturally be to be able to implement and play any kind of board game using this language, but as of now, that wish is far down the road. There are several aspects of our language that are specific to positional games because of the similar nature between them, and those would need to be further generalized to work with more categories, amongst other things.

5.4.2 Strategy-stealing argument

Something we discovered while reading about all the different positional games that exist is the fact that in a lot of the games, player one can always force at least a draw, i.e. player two cannot not win provided perfect play by player one. The existence of this first-player advantage can be proved (non-constructively) by what is known as the **strategy-stealing argument**. [22, p. 74-75][23].

The first player advantage is something to keep in mind if one wants to extend our project to implement computer players with perfect, or near-perfect play. As with perfect players, the first player always has an advantage, and can force at least

a draw (i.e. never lose).

The fact that the second player is at a disadvantage obviously discourages anyone from wanting to go second. One way to make it harder for player one to make use of his first-player-advantage in a game of two human players is to allow the second player to take over the first player's first move, if he wishes for that move to be his own. The first player then gets to place another piece which will be the new first move for player one. This does not solve the problem of player one having an advantage, but we believe that for human players, a first move swap makes it harder for player one to use this advantage to not lose the game.

Our implementation considers only human players and even though, theoretically, the first player can force at least a draw, we believe that due to the games complexity (for a human player) this does not make the gameplay too unfair. Especially if one uses the above method and also takes turns in who plays first.

5.4.3 Threshold Bias

A current research topic surrounding Positional Games involves trying to find the *threshold bias* for games [8]. The bias for a positional game $(X, \mathcal{F})$ expressed as $(p : q)$ is the game where the first player claims p positions for each move and the second player claims q positions per move. For some games, as described in the preceding section, where player one always wins in the ordinary, namely a non-biased $(1 : 1)$ game, there in fact exists a known constant p/q, *the threshold bias*, where the second player starts winning [24]. One possible extension could be to implement setting a biased value for games in the language to make the play more fair between the players.

5.4.4 vs. Computer and Online Multiplayer

Part of our scope was to make it only possible to play on a single computer with two human players. Expanding beyond this is an interesting idea for future work. Making it possible to both play against the computer, and online against other people, would be good additions to the language.

5.5 Sources

While looking through our sources for this thesis, there might be displeasure regarding the frequent use of Wikipedia. It is often not the preferred source to use in academic theses. However, when it comes to the rules of board games, we would argue that Wikipedia is a sufficiently good source.

5.6 Impact on society

Due to the nature of the project, we could not find any obvious societal or ethical aspects to take into consideration. The project is about developing a language to describe board games, which does not affect anyone's autonomy or integrity. It has no negative impact on society. The only impact we can detect is from an entertainment perspective, and it is hardly worth noting if someone does not find our result entertaining.

It can be argued that the gaming medium as a whole has had a negative impact on society, but we feel that aspects such as game addiction and excessive spending (i.e. gambling) does not apply to our project.

6 Conclusion

This thesis has shown the progress of implementing a language for board games from the positional game category. The language was implemented as an embedded domain-specific language (eDSL) using Haskell as the host language. To create the language, new data types, type classes and various different functions were defined which together build up this new language.

Implementing a game using the language is generally done in three steps, two of which are very simple. After that the GUI, which was implemented using ReactJS, can be customized to fit the implemented game. The purpose of the project has thus been met since the language allows for easy implementation of both the game logic and GUI. With the numerous example games provided, the user can easily see the language's possibilities, as well as study them to understand how the language works.

There is still much that can be done to further expand the project. Several examples mentioned in section 5.4, such as expanding to more categories and being able to play online, are but a few of the possibilities.

Bibliography

[1] B. Games, *Tabletop simulator: About*, 2020. [Online]. Available: https://www.tabletopsimulator.com/about.

[2] É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne, "Ludii – the ludemic general game system," in *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, G. D. Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, Eds., ser. Frontiers in Artificial Intelligence and Applications, vol. 325, IOS Press, 2020, pp. 411–418.

[3] D. Ghosh, "DSL for the Uninitiated," *Commun. ACM*, vol. 54, no. 7, pp. 44–50, Jul. 2011.

[4] T. Jelvis, *What is an embedded domain-specific language?* 2018. [Online]. Available: https://www.quora.com/What-is-an-embedded-domain-specific-language.

[5] J. Beck, *Combinatorial Games: Tic-Tac-Toe Theory.* Cambridge: Cambridge University Press, 2008.

[6] J. Bronowski, "The long childhood," in *The Ascent of Man.* Boston: Little, Brown, 1973.

[7] J. Beck, *Inevitable Randomness in Discrete Mathematics*, ser. University Lecture Series. Providence, Rhode Island: American Mathematical Society, Sep. 2009, vol. 49.

[8] M. Stojaković, "Games on Graphs," in *Graph-Based Representation and Reasoning*, N. Hernandez, R. Jäschke, and M. Croitoru, Eds., Cham: Springer International Publishing, 2014, pp. 31–36.

[9] Wikipedia, *Positional game.* [Online]. Available: https://en.wikipedia.org/wiki/Positional_game.

[10] D. Hefetz, M. Krivelevich, M. Stojaković, and T. Szabó, *Positional games.* Springer, 2014.

[11] Wikipedia, *Connection game.* [Online]. Available: https://en.wikipedia.org/wiki/Connection_game.

[12] ——, *M,n,k-game.* [Online]. Available: https://en.wikipedia.org/wiki/M,n,k-game.

[13] ——, *Shannon switching game.* [Online]. Available: https://en.wikipedia.org/wiki/Shannon_switching_game.

[14] ——, *Shannon switching game / variants / gale.* [Online]. Available: https://en.wikipedia.org/wiki/Shannon_switching_game#Gale.

[15] ——, *Hex (board game).* [Online]. Available: https://en.wikipedia.org/wiki/Hex_(board_game).

[16] D. Gale, “The Game of Hex and the Brouwer Fixed-Point Theorem,” *The American Mathematical Monthly*, vol. 86, no. 10, pp. 818–827, 1979.

[17] C. Strategy, *Strategy vs. tactics*. [Online]. Available: `https://www.clearpointstrategy.com/strategy-vs-tactics/`.

[18] C. Browne, “Yavalath,” in *Evolutionary Game Design*. London: Springer, 2011, pp. 75–85.

[19] WebAssembly Community Group, “WebAssembly Specification,” Tech. Rep., Mar. 2021. [Online]. Available: `https://webassembly.github.io/spec/core/_download/WebAssembly.pdf`.

[20] “Asterius: Bringing haskell to webassembly,” (ICFP, Sep. 23, 2018), Mar. 2021. [Online]. Available: `https://icfp18.sigplan.org/details/hiw-2018-papers/6/Lightning-talk-Asterius-Bringing-Haskell-to-WebAssembly`.

[21] A. Gill, “Domain-Specific Languages and Code Synthesis Using Haskell: Looking at Embedded DSLs,” *ACM Queue*, vol. 12, no. 4, pp. 30–43, Apr. 2014.

[22] J. Beck, *Combinatorial Games: Tic-Tac-Toe Theory*. Cambridge University Press, 2008.

[23] Wikipedia, *Strategy-stealing argument*. [Online]. Available: `https://en.wikipedia.org/wiki/Strategy-stealing_argument`.

[24] D. Hefetz, M. Krivelevich, M. Stojaković, and T. Szabó, “Biased games,” in *Positional Games*. Basel: Springer Basel, 2014, pp. 27–42.

A

Appendix

A.1 Cross Code

```
module Cross where

import Data.List (intersect)
import Prelude hiding (lookup)

import Data.Map (
    Map
  , elems
  , keys
  , lookup
  , member
  , adjust
  )

import Boardgame (
    Player(..)
  , Position(..)
  , PositionalGame(..)
  , mapPosition
  , isOccupied
  , takeEmptyMakeMove
  , nextPlayer
  , drawIf
  , criteria
  , symmetric
  , unless
  , player1LosesWhen
  , player1WinsWhen
  )

import Boardgame.ColoredGraph (
    ColoredGraph
  , values
  , mapValues
  , anyConnections
  , filterValues
  , filterG
  , hexHexGraph
  , missingDirections
  , hexDirections
  )

#ifdef WASM
import Data.Aeson (ToJSON(..))
#endif

-------------------------------------------------------------------------------
-- * Cross
-------------------------------------------------------------------------------

newtype Cross = Cross (ColoredGraph (Int, Int) Position (Int, Int))
```

```haskell
instance Show Cross where
  show (Cross b) = show b

#ifdef WASM
instance ToJSON Cross where
  toJSON (Cross b) = toJSON b
#endif

instance PositionalGame Cross (Int, Int) where
  positions   (Cross b)     = values b
  getPosition (Cross b) c   = fst <$> lookup c b
  setPosition (Cross b) c p = if member c b
    then Just $ Cross $ adjust (\(_, xs) -> (p, xs)) c b
    else Nothing
  makeMove = takeEmptyMakeMove

  gameOver (Cross b) = criterion b
    where
      criterion =
        drawIf (all isOccupied . values) `unless` -- It's a draw if all tiles are owned.
        -- Here we say that in any position where one player wins,
        -- the other player would win instead if the pieces were swapped.
        symmetric (mapValues $ mapPosition nextPlayer)
        (criteria (player1LosesWhen <$> -- you lose if you have connected 2 opposite sides.
          [ anyConnections (==2) [side1, side4] . filterValues (== Occupied Player1)
          , anyConnections (==2) [side2, side5] . filterValues (== Occupied Player1)
          , anyConnections (==2) [side3, side6] . filterValues (== Occupied Player1)
          ]) `unless`
        criteria (player1WinsWhen <$> -- you win if you have connected 3 non-adjacent sides.
          [ anyConnections (==3) [side1, side3, side5] . filterValues (== Occupied Player1)
          , anyConnections (==3) [side2, side4, side6] . filterValues (== Occupied Player1)
          ]))

      -- A list of coordinates for every side based on which neighboring tiles are empty.
      [side1, side2, side3, side4, side5, side6] =
        missingDirections b <$> [[hexDirections !! i, hexDirections !! ((i+1) `mod` 6)] | i <- [0..5]]

emptyCross :: Int -> Cross
emptyCross = Cross . hexHexGraph
```