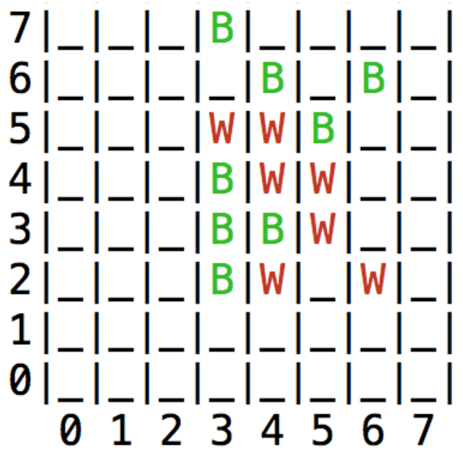
Spira: A language for prototyping grid board games with linear logic in Haskell

DATX02-20-26

Nicke Carlsson Simon Edvardsson Oskar Grekula
Erik Ljungdahl Jennie Zhou

2020-06-05



```
7|_|_|_|B|_|_|_|_|
6|_|_|_|_|B|_|B|_|
5|_|_|_|W|W|B|_|_|
4|_|_|_|B|W|W|_|_|
3|_|_|_|B|B|W|_|_|
2|_|_|_|B|W|_|W|_|
1|_|_|_|_|_|_|_|_|
0|_|_|_|_|_|_|_|_|
  0 1 2 3 4 5 6 7
```

https://github.com/ErikLjungdahl/spira

Abstract

This project explores the process of creating a board game model and introduces a new high level domain-specific language called Spira that specialises in prototyping *grid board games*. Spira is a DSL, embedded in Haskell, that generates Ceptre code. Ceptre is a language deeply rooted in the principles of linear logic, and it possesses qualities that allows board game functionality to be simplified and implementation minimised. To demonstrate Spira's capability of bringing Ceptre's inherent functionality into Haskell we constructed a case study, comparing two common board game implementations in Spira to Ceptre.

Sammandrag

Detta projekt undersöker processen i skapandet av en brädspelmodell och introducerar ett nytt domänspecifikt språk på hög nivå vid namn Spira som är specialiserat för prototyping av *rutnäts-brädspel*. Spira är en DSL, inbäddad i Haskell, som genererar Ceptre-kod från användarinmatning. Ceptre är ett språk djupt förankrat i principerna för linjär logik och den besitter kvaliteter som kan förenkla brädspelens funktionalitet och minimera dess implementation. För att demonstrera Spiras förmåga att föra Ceptres innehållande funktionalitet till Haskell, konstruerade vi en fallstudie där vi jämför två vanliga brädspelimplementeringar i Spira med Ceptre.

Acknowledgements

We would like to thank Anton Ekblad for being a great supervisor, and for all the help he has provided us with throughout this project. We are also very grateful for his help with starting us off in the right direction and his valuable guidance and feedback along the way.

In addition we would like to thank Anna Norrström for providing us with useful information regarding writing and oral presentation techniques. We are also thankful to the other students who have given us feedback on our report.

Contents

1 Introduction

This project describes the development of a new language for prototyping board games. Starting with the introduction, the cornerstones of this project and the thought process behind each step is explained.

1.1 Purpose

The purpose is to create a domain-specific language (DSL), for prototyping grid board games, that is easier to use and requires less written code than previous interpretations. It should result in a DSL, embedded in Haskell, that uses Ceptre as back-end.

1.2 Background

An important part of programming is choosing what language to work in, and depending on that choice the implementation process will look completely different. A task that in one language feels intuitive and self-explanatory could in another be filled with complications. Programming a board game can either be done in a general-purpose language or in a domain-specific language (DSL) that is designed towards game creation.

General-purpose languages, such as C, Java and Haskell, give the creator free control over how they want their games to work since there is little predefined game structure. It is also likely that someone intending to program a board game already knows one or more general-purpose languages. This because they are more likely to be widely distributed for being used in a multitude of different fields. The downside to general-purpose languages would be their lack of an optimised framework for board game creation, meaning certain aspects of board game functionality could end up becoming unnecessarily complex to implement.

Domain-specific languages meant for game creation, such as Multigame [1], PuzzleScript [2] and Ceptre [3], have the upside of an inherent structure for game creation that includes straightforward and powerful tools within that domain. Working in a DSL can improve the understanding of its problem area through the help of its various supporting operations. Furthermore, learning it is easier considering the limited domain. A potential downside of DSLs is that moving outside of its intended domain can be difficult, since the tools are usually limited to that domain. Because of this, it is required to put more effort into ensuring that the domain encompasses everything you want to achieve, compared to choosing a general-purpose language to work in.

The DSL constructed in this project is called Spira. To still gain some of the benefits a general-purpose language possesses, Spira is implemented as an embedded DSL in Haskell. This means that Spira has access to the syntax of the general-purpose language alongside its custom operations and game board optimised structure. Furthermore, Spira compiles to Ceptre code and uses Ceptre's logic structure to create the finalised game.

1.3 Delimitations

Limiting the scope to a more distinct board game type is sensible considering the large quantity of different behaviour within these games. Comparing two games such as Chess and Monopoly for example, while they both revolve around a board and game pieces, Monopoly requires completely separate functionality for the managing of capital and houses. The limited scope would make it easier to ensure that a specific game type is fully supported with helper methods.

To find a board game categorisation to work with, we looked at the largest and most regularly updated website regarding board games, Board Game Geek (BGG) [4]. BGG has two ways to categorise their games, either from a user perspective, with keywords such as 'memory', 'negotiation' or 'children's game' [5]; or by breaking down the games' mechanics into descriptions, such as 'dice rolling', 'race' or 'storytelling' [6]. A game mechanic that was relevant for our desired limitation was one called *grid movement*. A board game with grid movement has its pieces moving on the grid, which is usually a square or hexagonal board, in many directions [7]. A lot of classic board games fall under this description, for example Checkers and Chess. However, there are other games not being encompassed that share a lot of similarities with for example Chess, which we would also like to include in our scope. These are games that do not have moving pieces, but where the win condition is dependent on descriptions of relations on a grid, for example Tic-tac-toe and Connect Four where pieces once placed are unable to be moved. Since none of the categories or mechanics properly satisfy this condition, we decided to define a category of our own: *grid board games.*

Grid board games are games that can be played on a grid and function primarily based on the relations between its pieces and board. Examples of such functionality are the movement of pieces in Chess, games whose states depend on grid descriptions and the three-in-a-row win condition in Tic-tac-toe. This means any game with a board that can be represented as a type of grid and does not include separate components, such as dices and cards. Therefore, games such as Ludo, also known as "Fia med knuff", and Monopoly are excluded. Only allowing the use of grids as boards could seem like a strict limitation, but the truth is that a lot of different board shapes can be expressed through grids. For example, Ludo's plus-shaped board could quite easily be modelled as seen in Figure 1. It is made by creating a quadratic grid and having certain tiles be unavailable, shown as grey in Figure 1. By giving the grey tiles a value of for example 'null' and the white tiles 'free' and then only operating on 'free' tiles, you have in essence created a Ludo board.

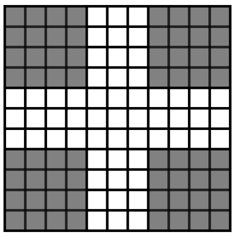


Figure 1: *Ludo board represented as a grid, where the grey tiles are unused.*

While this category has been the main focus when creating Spira's helper operations, it is still possible to create games outside of the category. Although Spira does not have explicit support for things like dices or cards, it is relatively simple to implement such things using Spira's basic components. Therefore, Ludo and Monopoly are games that can be implemented using Spira, if the user is willing to put in a bit more effort.

Limiting this project to focus on grid board games was mostly due to two reasons. Firstly, we wanted to create a language for a less explored area of Ceptre, and grid board games happen to be one of those areas due to the lack of inherent support for geometric properties. Secondly, many of the easier board games to model that we could think of, Tic-tac-toe and Connect Four for example, consist almost entirely of grid descriptions. This made it easier to start working by making prototypes of these games in our front- and back-end. Aside from deepening our understanding of board game structure, looking for similarities in these prototypes would later come to help us write the pseudocode for our language which can be read about in 3.3 *Modelling board games.*

1.4 Domain-specific languages

A domain-specific language (DSL) is a language with the sole purpose of modelling a particular problem. Two examples of DSLs are Structured Query Language (SQL), a language for managing databases used to some degree by most of the databases in the world [8]; and MATLAB, a matrix-based language made for mathematical computations used by millions of engineers and scientists around the world [9]. When a DSL is written in a general-purpose language, using that language as host in order to inherit its syntax, it is called an embedded domain-specific language (eDSL). The benefit of an eDSL is that it gives you access to the constructs of the host language when constructing helper methods. Furthermore, users experienced with the host language will also find it easier to use the DSL, since its syntax is carried over from a language they are familiar with. The difference between an eDSL and a normal library is that an eDSL, being its own domain-specific language, is designed for being primarily used to solve a problem domain while a library can be practically anything importable, regardless of having a clear purpose or not. Two examples of eDSLs are Haxl, a DSL written in the general-purpose language Haskell that simplifies remote data access [10]; and Feldspar, a language that enables high-level and platform-independent descriptions of digital signal processing algorithms also written in Haskell but that generates C code [11].

Similarly to how eDSLs are a subdivision of DSLs, eDSLs can be broken down further into something called deep and shallow embedding. Choosing what type of embedding to use decides how operations in the DSL are implemented. Deep embedding builds an abstract syntax tree from the operation, a tree that can be transformed and traversed in order to become more optimised, before translating it to the target language. Shallow embedding is more direct, translating the operation to the target language immediately. An eDSL often contains a mix of deep and shallow embedding depending on what operations are being managed [12].

The eDSL created in this project, Spira, is embedded and similarly to Feldspar it generates code in another language. Worth mentioning is that Spira will be referred to as a DSL rather than eDSL in a majority of the report. This is done because DSL is the general term for languages of this type with 'embedded' more describing its implementation.

1.5 Front- and back-end

The structure used for Spira is called front- and back-end, where the language the programmer writes in is called front-end and the language the programmer never directly interacts with is called back-end.

The front-end used for Spira is Haskell, a functional programming language. Haskell possesses a powerful type system that supports algebraic data types. These are types that let you specify the shape of each element. It is, for example, possible to pair elements together with a tag to be used as a single structure. These structures can be used similarly to how a single element would, you can for example pair two structures together with a tag to form a new structure. This is useful for creating trees since a structure can contain two other ones, each in turn containing two of their own and so on, accurately representing the branches of a tree [13]. Having a way of easily constructing trees is good for eDSLs since deep embedding uses abstract syntax trees. Haskell also possesses lazy evaluation, which means arguments are not evaluated before being passed to a function. This makes creating combinators, a term often used when referring to functions of an eDSL, easier, since it allows for the handling of undefined values and infinite loops. Another good aspect of Haskell is its monad-support. Monads in Haskell are used to describe composable computations and grants features such as I/O, common environment and updatable state. Updatable state, also known as state monad, is especially interesting in this case since it allows for the use of a global state alongside do-notation. This makes it possible to always change things inside of the state — something very reminiscent of a board game [14].

The back-end language used for Spira is Ceptre, a language built on the principles of linear logic [3]. The advantages of using Ceptre as the back-end is thoroughly explained in section 2.2 *Ceptre*. Examples of languages or engines that could have been used as back-ends instead of Ceptre are Unity [15], PuzzleScript [2] and Multigame [1]. The reason for not choosing Unity, an engine where one can create objects in three-dimensional space to control and apply functionality to, is the difference in problem area compared to this project. Using a tool meant to generalise game creation as a whole for the small task of creating board games would result in too much work when translating the code from our front-end. PuzzleScript is a scripting language made for creating certain types of puzzle games. Similarly to this projects delimitation, it is designed for working with grids. The reason PuzzleScript is not a valid option for creating board games comes down to some of its weaknesses. On PuzzleScripts official website the creator has listed a number of things difficult to do in PuzzleScript [16]. Written on that list is, among others, counting and diagonal movement. Counting is an important aspect of many board games win conditions, such as the number of adjacent markings in Tic-tac-toe and Connect Four; and diagonal movement is something we see in games such as Chess, where the bishop cannot move in any other direction. Multigame is optimised for board games and has a similar domain to the one used in this project. It is also a high-level language meaning it is easy to learn and understand. However, one important detail in the DSLs rules was problematic for us: it only supports the creation of games with up to two players. Since we wanted to allow games with a larger amount players it was not compatible.

1.6 Related work

This project and its resulting language, Spira, is made possible by the use of knowledge from similar projects. This section provides short summaries of these projects as both acknowledgement of their contribution, and to supply readers with more material within the subject.

The logic used for Spira is something researched by logician Girard and thoroughly described in his article *Linear logic: its syntax and semantics* [17]. The examples used in our article to describe the principles of linear logic are heavily influenced by similar examples found in his paper. Girard explains not only the rules of linear logic, but also presents proofs, theorems and the complete theory behind what sets linear logic apart from usual logic and what they have in common. Our explanation of linear logic is found in 2.1 *Linear logic*.

The back-end used for Spira, Ceptre, is a language written by PhD graduate Martens for their thesis [18] and is thoroughly described in *Ceptre: A Language for Modeling Generative Interactive Systems* [3]. The paper presents Ceptre as a means to administrate operational logics in ways not previously possible with other languages. It also covers the advantages of basing your language on linear logic and compares Ceptre with other programming languages. A large part of the paper consists of case studies meant to show compatibility with many different game types; one of which is board games which correlates to this project's use case. It also mentions that Ceptre was designed to be suitable as a back-end for other languages.

The article *Multigame — A Very High Level Language for Describing Board Games* [1] introduces Multigame, a language for describing board game rules on a high level. The paper brings up the problem of lower-level languages and the advantages a high-level language such as Multigame offers. It gives examples of a multitude of board games that can be created using Multigame, while still mentioning some of its limitations, such as only supporting games with up to two players — the reason it was not used for this project.

PuzzleScript [2] is a scripting language written by game developer Lavelle. Lavelle considers PuzzleScript neither a general-purpose game making tool nor a general-purpose puzzle game making tool, but rather a tool that *"might prove handy/enabling for a number of people, if one is accepting of its limitations"* as he words it. Regardless, PuzzleScript was originally created as a way of simplifying puzzle-game creation. The engine offers a movement system that is both flexible and powerful; something like getting a player character to push a box can be done in a single line of code. The idea of using grids to represent the board in this project was influenced by PuzzleScripts use of a grid with interactive tiles to perform a wide range of different behaviours; along with the ability to create many different levels by altering the shape trough limiting the tiles that can be stepped on. If one is interested further in PuzzleScript, the official webpage [19] allows you to work with it yourself and offers a large number of tutorials and games created by others.

2 Theory

In this section, we describe the core theory needed for our project: board game theory and the programming language *Ceptre* together with its related theory on *linear logic*. We also provide some concepts regarding regular expressions that are used for simplification of the user interface during run-time of a game.

2.1 Linear logic

Linear logic was introduced by Girard [17] and here is an explanation as to why it is suitable for this task. In classical logic an atom represents a *fact*, and an implication represents drawing a conclusion based on some other knowledge. There is however another kind of implication in real life, the implication of *state* or *resources*. The difference is that when you draw a conclusion based on some fact, you do not lose the previous knowledge, but rather add to it. For resources however, exchanging some resource for another, or moving from one state to another state, usually means the previous situation is changed. This is expressed in linear logic with the binary operator for linear implication: $\multimap$ (lollipop) where $p \multimap q$ means exchanging 'p' for 'q' — losing 'p' in the process. Two more operators of linear logic are used in this project, linear conjunction ("and"), $\otimes$, which works as classic conjunction but in the world of linear logic, as well as the unary operator !, which marks facts and therefore allows us to retain classical implication as well.

2.1.1 Weakening and contraction

The core idea in linear logic is that everything is resources instead of facts [20]. In order to support this kind of reasoning, linear logic (partly) discards two central rules in classical logic: weakening and contraction [21].

The weakening rule says that we do not lose anything from adding more hypotheses, which makes sense when dealing with facts. For example, if we know that it is raining, then we can conclude that it is wet outside, regardless of if we know what day it is.

$$Rain \implies Wet \tag{1}$$

$$Rain \land Tuesday \implies Wet \tag{2}$$

We can visualise this with a proof tree, note how the knowledge of 'Tuesday' is not needed: we have acquired 'Wet' already in the top left sub-tree.

$$\dfrac{\dfrac{Rain}{Wet}\,(1) \qquad Tuesday}{Wet}\,(2)$$

However this does not work when dealing with resources: if we have one coin and use it to buy a piece of candy, we end up with only a piece of candy and no coin. If we use weakening to add another coin, and then use one of the coins to buy a piece of candy, we no longer end up with just a piece of candy, but instead with a piece of candy **and** one coin. You could say that the resulting state is {coin : 1, candy : 1} instead of {candy : 1}. We could also use the other coin to end up in a state with {candy : 2}. Visualised in Figure 2.

$$\dfrac{\dfrac{\dfrac{Coin}{Candy} \qquad Coin}{Candy \otimes Coin}}{Candy \otimes Candy}$$

Figure 2: Proof tree of buying candy in linear logic.

Similarly, contraction, removing duplicate hypotheses, is not allowed either since from two coins we could derive two pieces of candy, but from only one coin we cannot possibly derive two pieces of candy — thus they cannot be equivalent.

Note that facts are not completely removed in linear logic, instead they are marked explicitly by the exponentials ? and !. Furthermore, weakening and contraction are allowed as logical rules on such facts, rather than as structural rules applying to the whole logic [22].

2.1.2 Proof search as computation

In order to do computations with logic, one can use *logic programming*, which means first stating known facts and formulas, before having the program perform a proof search. Martens [18] makes a point in distinguishing proof search from proof construction. In proof search you want to know only whether there is a proof (and what it is) or not, whereas in proof construction we are interested in the proof itself and how it is constructed. The reason we are interested in the construction instead is because it corresponds to the playing of the game, making one move at a time. A proof in itself, without caring about the construction, would instead correspond to a *finished* game, and could be used to answer questions similar to *"Is X a possible outcome in game Y under the conditions Z?"*. There is also another similar approach to the connection between programs and proofs, the Curry-Howard isomorphism. While in logic programming, the running of the program is the process of making a proof, the Curry-Howard isomorphism instead states that the program itself can be seen as a proof, and the types used can be seen as propositions [23].

2.1.3 Forward- and backward-chaining

There are two different approaches for proof searching in logic, forward- and backward-chaining. In classical logic forward-chaining starts from known facts and matches all possible premises on all available inference rules. Whenever a match is found, the rule's conclusion is added to the set of known facts. This process is then repeated until either no more premises match (no proof), or the desired conclusion has been found (proof obtained). Backward-chaining does essentially the same thing, but in reverse. That is, it starts from the desired conclusion and matches it with conclusions of available inference rules. Whenever it finds a match it will add the premise of that rule to the set of goals. It continues this way until it has no more goals (proof obtained), or it cannot match any more conclusions with the available goals (no proof). The two approaches can also be combined, you could for example start with forward-chaining and then continue with backward-chaining, but add the already proven facts as premises. In Ceptre we primarily use forward-chaining because it corresponds to playing the game, and use backward-chaining for facts [18].

It is worth noting that this works a little differently in linear logic compared to classical logic. The difference is that while in classical logic you dont lose facts as you use inference rules, in linear logic those can be lost. For example, consider the following formula in linear logic where $\otimes$ (tensor) is the linear *and*, and $\multimap$ (lollipop) is the linear implication:

$$p \vdash q \otimes u \tag{3}$$

given the inference rules:

$$p \multimap q \tag{4}$$

$$p \multimap t \tag{5}$$

$$t \multimap q \otimes u \tag{6}$$

We can see that a valid proof is the application of first (5) and then (6):

$$\dfrac{\dfrac{p}{t}\,(5)}{q \otimes u}\,(6)$$

resulting in the state of both 'q' and 'u'. However we note that if we were to first apply (4), then we would obtain 'q' only, with no more premises match, failing to find a proof:

$$\dfrac{p}{q}\,(4)$$

2.1.4 Linear logic for state

Board games typically consist of a board and some rules that dictate what operations are allowed and when someone wins, based on the game state. The game state usually includes things such as: the board and position of pieces, whose turn it is, if the game has ended with some winner and other game-specific information. For example, in Tic-tac-toe you are allowed to place your marker on some square if it is your turn and that square is free.

Linear logic is suitable for board games since it is designed to describe state change [24]. Other options to consider would be for example *linear temporal logic* or *computation tree logic*. These however have a notion of time, in the form of a sequence of events, rather than just changes in state. Such sequences are built by specifying, for example, if a predicate holds in the present, or if it will come true sometime in the future [25]. This makes it more complicated, without any clear benefit for describing game rules [18]. They are more suited for questions such as those discussed in 5.2.4 *Game-solutions* and 5.2.3 *Sanity checking*. As an example of linear logic, we can represent making a move in Tic-tac-toe as:

$$free(Square) \multimap occupied(Player, Square), \tag{7}$$

where 'Square' and 'Player' are variables over the squares on the board and the players playing, respectively.

2.2 Ceptre

First, what is Ceptre? Quoting Martens [3], Ceptre is a *"rule specification language* [...] *intended to enable rapid prototyping for experimental game mechanics, especially in domains that depend on procedural generation and multi-agent simulation."*

More explicitly, Ceptre is a *logic programming language*, which means it is primarily based on formal logic — in this case a fragment of linear logic. The main concepts used are *linear conjunction*, *linear implication* and an *exponential* for marking facts. In Ceptre these are represented by `*`, `-o` and `!` respectively, corresponding to $\otimes$, $\multimap$ and ! in linear logic.

From Martens [18] there are three main points to using Ceptre which are important to us:

- Mathematical foundation through linear logic.
- Powerful enough for most components you would want in board games.
- Built to enable front-end tools based on it.

Our goal is to allow rapid prototyping of board games, which is exactly the kind of task Ceptre is intended to support. In addition, many board games are multi-agent, i.e. we often have two or more players interacting, and sometimes rules are automatically applied.

2.2.1 Using Ceptre

The control flow logic of programs written in Ceptre is primarily built around *stages*. The program will be in exactly one stage at a time, and can transition between stages by user-provided stage transition rules. A stage contains a collection of linear implications, and the program may only use those in the current stage.

In order to write the rules, one can introduce *predicates*, which can have zero or more arguments, each of some *type* that the programmer also introduces. There can also be instances of a type. For example we can introduce the predicates `win` and `lose` taking zero arguments:

```
win : pred.
lose : pred.
```

a type `player` and an instance:

```
 player : type.
 alice  : player.
```

and a predicate for having a move, taking a player as an argument:

```
move player : pred.
```

We can then create a stage that lets the player make a move, in this case the player only chooses to win or to lose. Note that `Player` is a free variable of the type `player` (they don't have to have the same name), and it works like pattern matching. That means when the program runs, Ceptre will try to assign each free variable a value such that the rule can be fired. If we were to have another instance of `Player` in the conclusion of the implication, for example `win Player` instead, then that instance would be bound to the first instance.

```
stage play = {
move1
    : move Player
    -o win.
move2
    : move Player
    -o lose.
} #interactive play.
```

Note the *interactive* at the end, this allows the user input to choose among possible rule applications at run time. We can choose to omit this to make the program automatically choose rule application, as Ceptre will choose amongst the possible rules at random — which is useful to provide randomness such as dices [18].

To finally make the program runnable we need to provide Ceptre with a starting point as follows:

```
context init = {
move alice
}

#trace _ play init.
```

In the context we provide a starting state, consisting of applied predicates separated by a comma. The *trace* provides the maximum number of steps, here unbounded, as well as the starting stage and initial state — `play` and `init`.

2.2.2 Numbers in Ceptre

Ceptre does not have numbers as we normally know them in most programming languages, no integers, floats nor doubles. Instead it is most natural to use what is known as Peano arithmetic [26]. This means defining the natural numbers inductively as follows:

- $z \in \mathbb{N}$
- $\forall n \in \mathbb{N}.s(n) \in \mathbb{N}$

Essentially what we are saying with this definition is that 'z' is a natural number (zero), and then s(z), corresponding to 0+1, is also a natural number; and therefore so is s(s(z)) and so on, continuing indefinitely.

One way to make the previous dummy game presented in 2.2.1 *Using Ceptre* slightly more meaningful would be to introduce randomness in the form of, for example, a 2-sided dice. First we provide some types and predicates that are needed. A dice token to consume in order to roll the dice, natural numbers and the result of the dice roll.

```
nat      : type.
z        : nat.
s nat    : nat.
dice/token  player     : pred.
dice/result player nat : pred.
```

Then we provide the dice rolling stage, which we could use to replace the previous play-stage. We need one rule for each result, as we want the dice equally weighted. We could provide two rules for the same outcome to skew the probability.

```
stage dice = {
roll1
    : dice/token P
    -o dice/result P (s z)
roll2
    : dice/token P
    -o dice/result P (s (s z))
}
```

Note the lack of interactive statement, `#interactive dice`, we do not want the stage to be interactive. It is also worth noting that all rules have the same condition, otherwise we risk a situation where only a subset of the rules can fire, skewing the result of the dice.

2.2.3 Quiescence in Ceptre

We have seen how to use basic stages, so next we will take a look at stage transitions. Stage transitions in combination with *quiescence* allows Ceptre to become more powerful [18].

Quiescence, meaning *the state of being temporarily quiet and not active*, is represented by the built-in predicate `qui` that is introduced at run-time when no more rules can be fired within the current stage [18].

A particularly powerful example of using quiescence is negation, which is currently not possible in Ceptre right away [18], although there has been research on how to include negation in logic programming [27].

Here is an example implementation of negation, the purpose is to be able to conclude `not_exist` if the predicate `target` is not present. First we define the predicates to use:

```
target    : pred.
exist     : pred.
not_exist : pred.
```

The goal is to show that `not_exist` is equivalent to `target` not being present when checking. We initialise with `not_exist`, and the idea is to attempt to consume both `not_exist` and `target` at the same time in some rule. We do this in the check-stage:

```
stage check = {
exists
    : not_exist * target
    -o exist.
}
```

Now we have one part of the equivalence, this rule can only fire if `target` does exist, therefore we now know that `exist` implies that `target` existed at the time of checking. The second part of the equivalence is to ensure that `not_exist` implies that `target` did not exist. We can do this by utilising the quiescence predicate `qui` in the following way in a transition rule between stages:

```
go/not_exist
    : qui * stage check * not_exist
    -o stage not_exist.
```

`qui` will be introduced if no rule in the check-stage can fire. This means that since we initialised with `not_exist`, if we have both `qui` and `not_exist`, then `target` does not exist. Thus we now have the second piece of the equivalence, and can conclude that `not_exist` is equivalent to the target not being present when checking, and vice versa for `exist`. We can also provide a transition for the `exist` case, allowing us to branch the program based on the existence of `target`.

```
go/exist
    : qui * stage check * exist
    -o stage exist.
```

In addition to negation, Martens [18] presents some more design patterns for Ceptre: rule ordering, finite set comprehension and more.

2.3 Regular expression

Regular expression [28] is a sequence of characters specialising in searching for a specific pattern, these are usually found in some programming languages find/replace functions like Java's `replaceAll(String regex, String repl)` [29]. Because regular expression are an efficient and simple way of finding sequences, they suit this project perfectly when it comes to making Ceptre's output more readable for the user. We will show some functions and most commonly used patterns that regular expression have in its library.

Three of the functions that are highly useful are *search*, *match* and *sub* presented in Table 1. With these you could accomplish most of the possible modifications and searches you would want for a String.

Table 1: Functions implemented in regular expression.

Function	**Parameters**	**Description**
Search	Pattern String Flag	Searches for the first instance of a matching pattern. It returns a match object but can be converted with its method group().
Match	Pattern String Flag	Searches for of a matching pattern in the beginning of a String. It returns a match object but can be converted with its method group().
Sub	Pattern Replacement String Max	Replaces all instances of the matching pattern with the replacement string if no max is defined, otherwise it takes the max number of instances. Returns a new string.

When using regular expression functions, all of them have a parameter called 'Pattern'. A pattern is a way of expressing what to search for and you can use multiple smaller patterns for creating a more complex one. We list some basic ingredients for the creation of a pattern in Table 2.

Table 2: Some patterns for matching a sequence.

pattern	**description**
`^`	Matches beginning of the string
`$`	Matches end of the string
`[...]`	Matches any character of the ones inside the bracket
`*/+/?`	Matches 0 or more/1 or more/ 0 or 1
`\d / \D`	Matches digits / does not match digits.

These are just a few of the patterns that exist, by combining them we can create a lot of complex matches. For example, if we have a String: **`"Regular expression is a nice tool to have"`**, lets call it '**`str`**'. If we want to find the first word **`"Regular"`** we use `re.match(r"(Regular)", str)`. If we want to replace the first five characters of **`"e"`** in the string to **`"a"`** we use the sub-function like this, `re.sub(r"e", "a", str, 5)`.

3 Method

This section covers the different methods that have been used throughout this project. In general, the design of Spira is based on a study of several board games, some of which will be presented in 4.3 *Case studies*. We describe two approaches performed in the means of developing the *core*, which was the main goal of this process. Furthermore, we describe the partitioning of the workload and workflow. Additionally, towards the end of the section, we proceed with an explanation of the modelling board games part in more detail, describe prototypes in Ceptre and the implementation in Haskell.

3.1 The core

We have applied two different approaches to a specific node to clarify the description of our method, presented in Figure 3. The goal was for the core to perform as a node that connects the programmer of Spira with the code of Ceptre. The first approach was a broad perspective from the programmer to the core, whereas the core contains the data types and functions that facilitates the use of Spira. The second approach focused on how to compile the code from Spira to Ceptre.

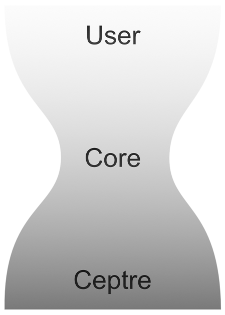


Figure 3: *The core of Spira.*

3.2 Workload

The general workload was divided into four parts that were dependent on each other. Each part required support from another during the work process.

- Ceptre
- Modelling board games
- Implementation in Haskell
- Parse Ceptre-output

The first part was Ceptre, the back-end chosen for our project. In this part the main focus was studying the language and specialising in how to implement grid board games in Ceptre. Since Ceptre is posing as a sort of intermediary between linear logic and programming code as described in 2.2 *Ceptre*, it was important to figure out how we could implement board games in it efficiently — in particular finding ways to represent grid properties in linear logic. Moreover, we had to find an approach to generalise our implementations in Ceptre to be able to generate suitable code from our Haskell module. This was based on the second approach in Figure 3, as mentioned in 3.1 *The core.* More about this in 3.4 *Ceptre prototypes.*

In order to facilitate the implementation in Haskell, it was necessary to introduce a second part, which was modelling board games. In this part, we had to consider how to effectively model board games, for instance focus on what assumptions were possible to make and where the programmer would want more control in the module. Modelling board games is further discussed in 3.3 *Modelling board games.*

The third part of the workload was the implementation of Spira in Haskell, to provide tools to further aid in prototyping board games for users of Ceptre. In particular, the focus was to implement an eDSL as earlier mentioned in 1.2 *Background*, by compiling Ceptre code. Once we got the idea of what both the DSL interface and the output code should look like, we had to start implementing the compilation from our DSL to Ceptre. Based on the input to our model, we had to take into consideration how to interpret it and translate that code into the corresponding Ceptre code. In the implementation we also had to take into account the expectations from the user of our language, as presented as the first approach in 3.1 *The core*, so that Spira contains enough information for it to be compiled into Ceptre.

The output from Ceptre contained representations of internal data structures, which would be difficult for the user of Spira to read and understand. Therefore, we introduced a fourth part, which was to parse the Ceptre-output in order to facilitate the readability for the user as much as possible. More information about this in 4.2 *User interface with Ceptre.*

We found inspiration regarding agile methods from the Agile Manifesto created in 2001 by the agile manifesto authors, which promotes collaborating agile teams to produce better software with faster delivery and higher value [30]. Since agile ways of working have been proven to help deliver products of high quality to the customers, we determined to adopt an agile workflow for the project. Furthermore, in order to retain structure in the work, we decided that sprint planning would be helpful for our project and used Trello, a flexible tool for organising projects into boards [31]. Trello has helped us keep track of the project, since it contains options for displaying a multitude of tasks, that can show you which people are working on what tasks and how far a task has come in its completion.

3.3 Modelling board games

The modelling process was divided into two tasks. Analysis, where we picked out a number of board games to compare and find similarities between, and the creation of a pseudo-code model where we used the knowledge from the analysis to write a basic type-system that was used as reference when creating the Haskell core for Spira.

3.3.1 Analysis

The first game we analysed was Rock-paper-scissors. Although not originally a board game, it could be implemented as one with the use of a two-tiled board and pieces representing rock, paper or scissors. Other games that were included in the analysis are Tic-tac-toe, Connect Four, 21-sticks and Chess. By roughly breaking down each game into its basic functionality and general components, as presented below, we found several things they had in common. For example, Tic-tac-toe, Connect Four and Chess all required the ability to occupy board tiles with pieces and these pieces in turn needed parameters to represent which player they belonged to. This information was then used to determine what data-types were needed when writing the pseudo-code model.

Breakdown of Tic-tac-toe:

- 2 players.
- Board of 9 empty squares in a 3x3 pattern.
- Players take turns marking empty squares.
- Every turn after a player has marked a square the game checks for a winner, if the win condition is satisfied the game ends.
- Win conditions: Three markings belonging to the same player, neighbouring in a diagonal, vertical or horizontal line.
- Every turn, after checking for a winner, if there is no winner and all board. spaces are marked/ no squares are empty its a draw and the game ends

3.3.2 Pseudo-code model

Since the primary objective of the pseudo-code model was to be used as reference for the Haskell core that we later implemented, the first things we included in the model were the core components: player, board and pieces. Thus, a composition of data types, roughly meant to imitate a Haskell implementation of a general board game, was made:

```
data Game a = Game
    { players :: [Player]
    , turn    :: Player
    , board   :: a
    , rules   :: Rule a
    }
```

Game is a data type containing a number of variables that themselves are data types. We refrained from giving board a type to let the programmer freely decide how they want it to be implemented. This decision was made to support a wider range of board shapes since different boards could require different approaches of implementation. The other types are defined as follows:

```
data Player = Player { name :: String, id :: Int }
```

The Player type contains a name and an id for the game to differentiate players in case multiple players share the same name.

```
data Rule a = Move (Game a -> Game a)
            | WinCond (Game a -> Player)
```

Rule has two possible types of arbitrary operations, Move and WinCond. The details of the operations are defined by the programmer, but their purpose correlates with their names. Move is intended for moving a piece on the board and WinCond is intended for detecting if any player has won.

```
data Tile = X | O | Nil

board :: Board [[Tile]]
board = Board [[Nil | i <- [0..2]] | j <- [0..2]]

alice :: Player
alice = Player { "Alice", 0 }
bob :: Player
bob = Player { "Bob", 1 }
players :: [Player]
players = [alice, bob]

changeTileAt :: Int -> Int -> Tile -> Board -> Board
changeTileAt x y t b = ...

threeInRow :: Board -> Player
threeInRow b = ...

ticTacToe :: Game [[Tile]]
ticTacToe = Game  { players = players, turn = alice,
        board = board, rules = rules }
```

Figure 4: *Tic-tac-toe in pseudo-code.*

To see if the data types defined in the pseudo-code model were enough to be able to create the structure of a board game, we used them to build a pseudo-code prototype of Tic-tac-toe. The result can be seen in Figure 4. For the board we used a list of `Tiles` inside another list to represent the columns and rows of a rectangular board. `Tile` is a data type that can either be `Nil`, in other words empty, or `X` or `Y` depending on what player occupies that tile. `Alice` and `Bob` are the names of the two players, with `0` and `1` being used respectively as their player IDs. `changeTileAt` is an operation that is supposed to change the value of a tile on the board, where the two integers would be used to specify the chosen coordinates of the tile. `changeTileAt` does not have any functionality implemented since the point of the example was not to make a complete board game, but rather test if the pseudo-code model was adequate for introducing all the components needed in a game. `threeInRow` similarly is meant to operate on the board, but unlike `changeTileAt` should not change the board state itself, but rather check the board state for a possible winner. `threeInRow` has not been implemented for the same reason as `changeTileAt`. Finally, `ticTacToe` takes all these values and assigns them to the variables making up the game.

3.4 Ceptre prototypes

In this section we first show an example of a prototype implementation of Tic-tac-toe in Ceptre, which you can find in full in Appendix A.

3.4.1 Tic-tac-toe in Ceptre

First we needed an idea of how to represent Tic-tac-toe. We considered the game objects *player*, *board* and *squares* — which the board is built of — as well as a 'turn system' which allows the players to make a move, automatically checks the win conditions and changes whose turn it is.

In Ceptre we can choose to represent 'objects' in two different ways, either as instances of a *type* or as instances of a *predicate*. Using predicates is usually simpler, but they cannot take other predicates as arguments. Therefore, we used type for some of the objects instead.

We represented players using the player-type, and introduced two arbitrary players, as follows:

```
player : type.
alice  : player.
bob    : player.
```

We represented squares of the board as either `free` or `occupied` (by some player), and two natural numbers corresponding to its xy-coordinate. These "loose" squares then made up the board, without any extra structure.

```
free            nat nat : pred.
occupied player nat nat : pred.
```

Here we used the Peano definition of natural numbers, explained further in 2.2.2 *Numbers in Ceptre.*

```
nat     : type.
z       : nat.
s   nat : nat.
```

At this point we had still not actually made any squares, but had made all the tools needed. Below is an instantiation of the bottom row in Tic-tac-toe.

```
free    z           z,          % (0,0)
free    (s z)       z,          % (1,0)
free    (s (s z))   z,          % (2,0)
```

The next step was to get the game to actually work, to make things happen. For this we needed three different concepts: *stages*, *linear implications* and *stage transitions.* See 2.2.1 *Using Ceptre* for more details.

3.4.2 The play-stage

We started by defining basic play, and then moved on to defining win conditions and the more subtle case of drawing. Consider first what a move is in Tic-tac-toe; it is some player's turn, that player chooses a free square and puts their marker there — making it now occupied by that player. We made this stage interactive, so that the program user can control where to place their marker.

```
stage play = {
play
    : turn A
    * free X Y
    -o occupied A X Y.
} #interactive play.
```

This almost worked, but there was one detail missing: we had to consider the opponent's turn as well. However, we did not want them to act immediately, since we also had to check win conditions between turns. Therefore we introduced two new predicates, one of them backward-chaining — meaning it represents a fact, rather than the usual linear resource.

```
token   player        : pred.
opp     player player : bwd.
opp     alice  bob.
opp     bob    alice.
```

The `token` predicate represents whose turn it is next, without actually giving them their turn. `opp` represents the fact that two players are opponents, and we instantiated it to `alice` being the opponent of `bob`, and vice versa. This let us make the following change to our play-stage, resulting in a fully working stage:

```
stage play = {
play
    : turn A
    * opp A B
    * free X Y
    -o occupied A X Y
    * token B.
} #interactive play.
```

3.4.3 Win conditions

Let us first look at the two transition rules from the play-stage. After someone made their turn, we had a transition to check win conditions. We knew the turn was over if we had a `token` instead of a `turn`, leaving us with the following transition rule:

```
go/win
    : qui * stage play
    * $token A
    -o stage win.
```

Similarly, if the `turn` was not consumed, it meant there were no more free squares and so the game should draw.

```
go/draw
    : qui * stage play
    * $turn A
    -o stage draw.
```

In these two transitions there are two main things to point out. Firstly `qui` ensures that no more rules can fire, this is what ensures that the draw-transition actually works — if the play-rule could have fired, it would have, consuming the `turn` in the process. The second key part is that we should only have one `turn` or `token` predicate at any given time, otherwise we would not be able to ensure there are none left after consuming one. The '`$`' here is syntactic sugar for also having the predicate in the conclusion of the implication, which allowed us to retain the invariant of exactly one `turn` or `token` predicate.

Next we show the very simple draw-stage. It had not necessarily needed to be a stage on its own, it could be included in the transition rule. However for consistency, we chose to keep as much logic as possible within the stages.

```
stage draw = {
draw
    : turn A
    -o draw.
}
```

For the win conditions we ended up with four different cases; one case for rows, one for columns and then one for each diagonal. We expressed a row in the following way: a square at any position (x, y) and the square at (x+1, y) and the square at (x+2, y). In Ceptre we wrote this as:

```
stage win = {
win/row
    : token B
    * occupied A X Y             % some square
    * occupied A (s X) Y         % square to the right
    * occupied A (s (s X)) Y     % another square to the right
    -o win A.
... }
```

The remaining row and diagonal rules were left out, but they follow in a similar fashion. The full details are shown in Appendix A.

The final step was to let the game loop if no player has won. We did this with the following transition rule from the win-stage to the play-stage:

```
go/play
    : qui * stage win
    * token A
    -o stage play
    * turn A.
```

Note that each win condition rule consumes a token. Combined with the `qui` this ensures that the game will only continue if none has yet won.

At this point the game was almost completely finished, and we just needed a few more details. Ceptre uses the **trace** command to acquire its starting point in the following way:

```
#trace _ play {init, all_free}.
```

We simply provided an unbounded number of steps, play as our initial stage and a context consisting of two sub-contexts; `init` and `all_free`. Init provided the starting player by giving them a turn:

```
context init =
{turn alice}
```

Figure 5 shows the context that provided our starting board state by creating a `free` predicate for each possible square on the board.

It is worth noting how easy it would be to make changes to the initial state in this case. For example, it would be possible to change the board shape here or start the game in some specific state — both of which can be very useful for prototyping.

```
context all_free = {              % (x,y)
free    z          z,             % (0,0)
free    z          (s z),         % (0,1)
free    z          (s (s z)),     % (0,2)
free    (s z)      z,             % (1,0)
free    (s z)      (s z),         % (1,1)
free    (s z)      (s (s z)),     % (1,2)
free    (s (s z))  z,             % (2,0)
free    (s (s z))  (s z),         % (2,1)
free    (s (s z))  (s (s z))      % (2,2)
}
```

Figure 5: Initialisation of the Tic-tac-toe board, setting all squares to free.

4 Results

In this section, we will first present how our implementation of Spira works. Then we take a look at how we enhanced Ceptre's user-interface. Finally, we will take on a couple of case studies, as examples of how Spira can be used to prototype board games.

4.1 Implementation

Spira closely matches the semantics of Ceptre, meaning we still have stages and stage transitions as described in 2.2.1 *Using Ceptre*, and the logic is described as linear implications. What we have done is making it easier to express these things for board games. We made the design choice that the programmer mainly puts the logic inside of stages, and then there are only two different transitions to glue the stages together.

4.1.1 Transitions between stages

For each stage we automatically add a pre-token and a post-token to every implication. Every time a player chooses an option within a stage, it consumes the pre-token and gives the player a post-token.

The two different stage transitions use these pre- and post-tokens to go from one stage to another. The first transition is `fromStageToStage`: when a player is given a post-token the transition consumes it and goes to the next stage, giving the player a corresponding pre-token. The second transition is `fromFailedStageToStage`, which applies when a player does not meet the requirements for any of the options. This is checked by whether or not the player still has its pre-token after being in the stage; if it does, the transition consumes that pre-token and goes to the next stage as previously stated. To make sure that any possibility in the stage has been chosen before transitioning, we add the key-predicate 'qui' in the transition, read more about it in 2.2.3 *Quiescence in Ceptre.*

The main philosophy in a stage is that only one option is chosen, in some cases you might want to apply all possible choices though. An example of this is in the board game Othello, where multiple rows/columns/diagonals can be flipped when placing a single piece. To accomplish this one could write the following piece of code:

```
stage_flip `fromStageToStage`       stage_flip
stage_flip `fromFailedStageToStage` stage_next
```

This makes the program recursively go through `stage_flip` until it fails, before moving to `stage_next`.

Having a game where the user gets multiple moves per turn is not as straightforward, but it can be achieved. An example of such a game is Takenoko[1] where you get 2 moves per turn. One solution would be to add identical stages, for example a `stage_move2` after the first `stage_move`:

```
stage_move  `fromStageToStage` stage_move2
stage_move2 `fromStageToStage` stage_next
```

A more general solution, that works for any number of moves per turn, would be to have a counter indicating the number of options a player is allowed to take, and decreasing that counter for each post-token. Once the counter reaches zero it moves on to the next stage. This logic would be in `stage_counter` in this example:

```
stage_move    `fromStageToStage`       stage_counter
stage_counter `fromStageToStage`       stage_move
stage_counter `fromFailedStageToStage` stage_next
```

4.1.2 Stages, types and constructors

Now you have seen how to do transitions between stages, but how are the stages actually built? Let us look at the type-signature

```
stage :: Interactive -> Var -> [Implication] -> M StageIdentifier
data Interactive = Interactive | Noninteractive
```

First we consider the return type, `M StageIdentifier`, the `M` is simply a type-synonym

```
type M a = State St a
```

We use a State-monad to keep track of everything inside our DSL, and it allows the programmer to use do-syntax, which we use in code examples throughout the report. The return-value `StageIdentifier` contains the pre- and post-tokens of that stage along with the stage-name, which is actually what the transitions use:

```
fromStageToStage       :: StageIdentifier -> StageIdentifier -> M ()
fromFailedStageToStage :: StageIdentifier -> StageIdentifier -> M ()
```

Then we look at the first argument of `stage`, it can either be `Interactive` or `Noninteractive`. If it is interactive the players get to choose one of the options, otherwise Ceptre randomly chooses between the allowed options.

To make sense of the rest of the arguments, we have to start from the bottom. First we need to explicitly create types with the function

```
newType :: Name -> M Type
type Name = String
```

[1]https://boardgamegeek.com/boardgame/70919/takenoko

Types are used in all constructor functions:

```
newPredConstructor :: Name -> [Type] ->         M ([Var] -> Pred)
newFactConstructor :: Name -> [Type] ->         M ([Var] -> Pred)
newConstructor     :: Name -> [Type] -> Type -> M ([Var] -> Var )
```

To create the type-signature of the constructur functions, they all take in a list of types. The `newConstructor` also needs a return type, as the Pred- and Fact-constructors already have their return-types in Ceptre: `pred` and `bwd` respectively, which we explain in 2.2.1 *Using Ceptre*. The type-signature is used in the function they return to type-check the variables `[Var]` upon applying them. `newConstructor` returns a `Var`, meaning it can be used recursively, while the other simply return a `Pred`.

We can use these `Preds` to actually create some logic, using the lollipop operator `-@` to create a linear implication

```
(-@) :: [Pred] -> [Pred] -> Implication
```

The operator simply takes in two lists of predicates, consumes the predicates in the first list and produces the predicates in the second list.

Predicates needs variables, and another way to create a `Var` is with

```
newBinding :: Type -> Var
```

A binding is used to bind variables between predicates within a linear implication, bindings are explained more in 2.2.1 *Using Ceptre*

We have now looked at the basics and can take another look at the rest of the stage

```
stage :: Interactive -> Var -> [Implication] -> M StageIdentifier
```

The next argument is a `Var`, which should be a player-binding we can apply the pre- and post-tokens to. This should in most cases be the same player-binding as you use in the `[Implication]` since that makes it operate on the current player, otherwise you can give it a non-bound player-binding which allows the stage to operate on any player.

The `[Implication]` is simply a list of linear implications of all the different possible choices/rules in this stage.

Now we have all tools to create the logic of a game, to make it complete we just need to create the initial state of the game which we can do with these functions:

```
initialStageAndPlayer :: StageIdentifier -> Var -> M ()
addPredsToInit :: [Pred] -> M ()
```

Finally, to compile the whole game into Ceptre-code we use

```
compileGame :: M () -> FilePath -> IO ()
```

4.1.3 Summary of implementation

We have given the programmer a set of tools for creating board games in Spira. The ones described thus far are the basic building blocks, with them you should be able to create about any board game. These might not be the most user-friendly though, for this reason we also added composite functions for common patterns in board games, such as managing players and having a rectangular board. We also added functions to compensate for Ceptre's limits. Such as having no inherent support for numbers or comparators like less-than (<) or not-equal (/=). We will make use of the primary composite functions in 4.3 *Case studies*.

The implementation is partly a shallow-embedding with the logic of the stages and transitions being handled in the exposed functions. On the other hand the application of predicates, constructors and variables are handled when compiling the Ceptre code, making those parts deep-embedded. The result is a mix of deep- and shallow-embedding which is quite normal when creating a DSL.

4.2 User interface with Ceptre

When a Ceptre program is executed there is a lot of information that gets printed. At first Ceptre prints a summary of the program it is about to run and after that all the possible actions that can be chosen by the player. Finally, it shows the final state and the trace. Most of this is unnecessary information for programmers intending to prototype board games. By simplifying what is shown on the screen, we can make the user experience quite a lot smoother.

Python is a well-suited tool for this type of task because it is easy to use for taking input and output. By using the python package *subprocess* [32] we can create a filter for some of the lines that Ceptre puts out, which we will explain in more detail in this section, along with how Ceptre's *log.txt* file was used to create a visual board.

4.2.1 Filtering Ceptre output

First of, to run a program with the Python filter on top of it you first need to run the Spira code to generate a Ceptre file containing the game. Furthermore, you use the Python script and give it the 'ceptre-bin' file for executing a Ceptre file and the file path to the game like `UI.py <filepath_ceptre> <filepath_game>`.

Ceptre alternates sending information to the output and receiving a chosen input option for the interactive play, see 2.2.1 *Using Ceptre*. The output contains all information from the summary to what options the user can choose from, in the form of a string separated by newline characters. We catch this output with a Python script's subprocess library and modify it.

Firstly, by removing some parts we can make it more straightforward when the game is played. This is done by applying a match with regular expression, as presented in 2.3 *Regular expression.* For instance, in the 'won' removal of Ceptre's summary, it is invariably that after the string `"#trace ..."` we go over to the options for our players. By exploiting this property it is easy to throw away the previous rows. The same can be done for every other string that we need to hide from the user.

4.2.2 Modifying the view of decisions

Apropos of the line containing the decisions for the user, there are some bigger difficulties explained in this section. Firstly, Ceptre does not have support for using integers, instead, it is represented in the form of successor functions. Secondly, the language does not display simple information on what is what or whose turn it is. The original unfiltered output looks like the following

```
0: (quiesce)
1: (play/1 oskar (coord z z))
2: (play/1 oskar (coord z (s z)))
3: (play/1 oskar (coord z (s (s z))))
4: (play/1 oskar (coord (s z) z))
5: (play/1 oskar (coord (s z) (s z)))
6: (play/1 oskar (coord (s z) (s (s z))))
7: (play/1 oskar (coord (s (s z)) z))
8: (play/1 oskar (coord (s (s z)) (s z)))
9: (play/1 oskar (coord (s (s z)) (s (s z))))
```

Figure 6: Unfiltered output from running Tic-tac-toe with Ceptre.

For the successor functions, the trivial solution was to interpret it as numbers. This was done by taking each line and applying regex sub function in Table 1 for the match `"\([sz].*?\)"`, which represents finding the first parenthesis and one or more `s` or `z`, ended by closing parenthesis. For each of these matches, the script replaces it by the number of letters `s` that we found or with 0 if we found the representation of only a `z`.

To simplify the interface for the user, we created a choice to use labels for each word in the row of options. There is plenty of useless information, for example, the `"(play/1"` that could be removed and some others that could need an explanation. This is done by creating a line in the Ceptre file with the function `outputNames`. For the example above, it would look like this:

```
%% play/1 Turn Col/Row
```

The double percent signs are generated automatically to create an easy way for finding the row that decides all labels. `"play/1"`, which is the first word, explains what stage the user is supposed to get labels for. The following words match the number of free variables in the Ceptre output and are parsed with the label chosen to explain what it is. We also introduced a label `Col/Row` that creates an easier way of seeing what coordinates you can choose from. So when everything is labeled and simplified we get the result seen in Figure 7.

```
Choose a move
---------------------
1: Turn: oskar  Col/Row: 0/0
2: Turn: oskar  Col/Row: 0/1
3: Turn: oskar  Col/Row: 0/2
4: Turn: oskar  Col/Row: 1/0
5: Turn: oskar  Col/Row: 1/1
6: Turn: oskar  Col/Row: 1/2
7: Turn: oskar  Col/Row: 2/0
8: Turn: oskar  Col/Row: 2/1
9: Turn: oskar  Col/Row: 2/2
```

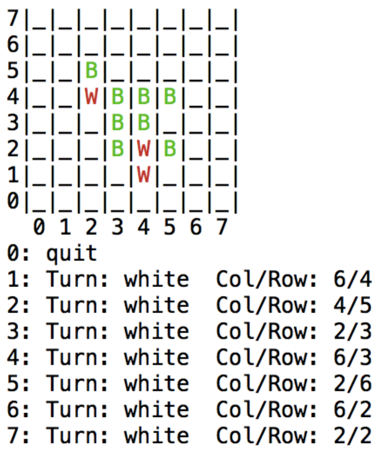
Figure 7: Filtered output from running Tic-tac-toe from the UI script.

4.2.3 Presenting the board

Another issue with Ceptre is that we have no visual representation of what moves that have already happened. It is fine for smaller games like Tic-tac-toe where we only have a three by three board and 9 different positions to remember. But with, for example, Othello we have an eight by eight board and we end up with 64 tiles and three states (free, black, white) for each. Ceptre can log all free variables for each stage that we have gone through in the file 'log.txt', this can then be used to find out which state the board is in at the moment. When you start a Ceptre program it directly starts to write to 'log.txt'. An example of a logged state looks like following:

```
---- {(stage play),
(pretoken_play xena),
(tile oskar (coord z (s z))),
(tile free (coord (s (s z)) (s (s z)))),
(tile free (coord (s (s z)) (s z))),
(tile free (coord (s (s z)) z)),
(tile free (coord (s z) (s (s z)))),
(tile free (coord (s z) (s z))),
(tile free (coord (s z) z)),
(tile free (coord z (s (s z)))),
(tile free (coord z z))}
```

It looks similar to the unfiltered Ceptre output for choosing a move, but we have the predicates for those stages instead. By first removing the unused rows (`"stage play"` and `"pretoken_play"`) we can use the same simplification on the successor functions that we have already implemented earlier, see 4.2.2 *Modifying the view of decisions.* Furthermore, we create a dictionary for each player where they have a list of their positions with a value. This can then be converted into a board representation.



```
7|_|_|_|_|_|_|_|_|
6|_|_|_|_|_|_|_|_|
5|_|_|B|_|_|_|_|_|
4|_|_|W|B|B|B|_|_|
3|_|_|_|B|B|_|_|_|
2|_|_|_|B|W|B|_|_|
1|_|_|_|_|W|_|_|_|
0|_|_|_|_|_|_|_|_|
  0 1 2 3 4 5 6 7
0: quit
1: Turn: white  Col/Row: 6/4
2: Turn: white  Col/Row: 4/5
3: Turn: white  Col/Row: 2/3
4: Turn: white  Col/Row: 6/3
5: Turn: white  Col/Row: 2/6
6: Turn: white  Col/Row: 6/2
7: Turn: white  Col/Row: 2/2
```

Figure 8: *The filtered version of how the board and options looks like for the game Othello.*

To create a board more pleasant to look at we created specific markers for each player, colours, board simplifications and a nicer way to present the game. To create a marker for a person we just used the first letter in their name and made it upper case to have an easy way to remember which marker is yours, after that the colours are generated dependent on what player appears first (only work for Linux and macOS systems.). To make it easier to see what is new we clear the output after each option, then repaint the board with its updated changes before displaying the options the player has. After all the changes, the output will look like Figure 8.

4.3 Case studies

This section describes two case studies of our project, Tic-tac-toe and Othello.

4.3.1 Tic-tac-toe

In 3.4.1 *Tic-tac-toe in Ceptre* we went through how to make Tic-tac-toe in Ceptre, and now we will also make it in Spira. The whole source code can be found in Appendix B.

To create the game Tic-tac-toe in Spira, we first start by designing the board which is a three by three grid with all tiles are set to free.

```
board <- initSimpleBoard 3 3
let (coordType, coord) = coord_t_c board
let free = free_v board
let tile = tile_p board
```

We then proceed to initiate players to the game, and distinguish the players by assigning them names. We also get the already initiated player-type with `gets playerType` for later use.

```
    (playernames,stage_next_player,opp)<- players["oskar","xena"]
    player <- gets playerType
```

When everything with the board and players is finished we can move on to finding a way to place a piece. By creating an implication that finds all free tiles, we use the position of one tile and convert it into a tile that is owned by the player. We then create this stage with the interactive play tag.

```
    pos <- newBinding coordType
    p <- newBinding player
    let impl = [tile [free, pos]] -@ [tile [p,pos]]
    stage_play<- stage "play" Interactive p [impl]
```

For win conditions, we simply use Spiras built-in `inALine` function that creates three cases: to win by horizontal, vertical or diagonal positions, all three require that the same player 'p' controls those tiles. We then add them to the non-interactive stage to always check for a winner:

```
    rules <- inALine 3 (p)
    stage_win <- stage "win" Noninteractive p (rules)
```

After creating the stages required for playing the game we now need to bind them together. First, we need to check for win conditions each time a player chooses a move which is done with `stage_win`, if a player can not choose a move we go to `stage_draw` instead. After the check for a win condition, there are two outputs. Either one of the players has won, or we need to carry on with the game, which sends us to `stage_next_player`. Furthermore, when we are in `stage_next_player`, a non-interactive stage, we can simply choose whose turn it is and go on to the interactive play stage `stage_play`. All this is done by the following four rows of code.

```
    stage_play `fromStageToStage` stage_win
    stage_play `fromFailedStageToStage` stage_draw
    stage_win `fromFailedStageToStage` stage_next_player
    stage_next_player`fromStageToStage` stage_play
```

Lastly, to create the initial states and what stage to start with we use the function `initialStageAndPlayer`.

```
    initialStageAndPlayer stage_play (head playernames)
```

Tic-tac-toe is a basic game with logic that is straightforward to implement. When looking at the code written in Spira we end up with a bit less than 50 rows. When comparing this to Ceptre, which needs 125 rows, it's clear a lot of it has been simplified with Spira's methods. For example, the `initDrawStage` which creates 5 rows in Ceptre. To further see the difference between Spira and Ceptre, we will look at a more complicated game.

4.3.2 Othello

This part will explain the creation of Othello[2] in Spira. We implement the version where the game ends when a single player cannot make a move, instead of both. The source code of Othello in Spira can be found in Appendix D.

When implementing the board game Othello, we start off by initialising the board with the function `initSimpleBoard` which sets all the tiles to free

```
board <- initSimpleBoard 8 8
let (coordType, coord) = coord_t_c board
let free = free_v board
let tile = tile_p board
```

In comparison to Spira, creating a board in Ceptre can be done in two different ways. The first one is to write out each tile representing the board. In other words hard-coding the board, which becomes more problematic the bigger the board gets. For instance, initialising the board for Tic-tac-toe shown in Figure 5, required 9 lines of code in Ceptre. Perhaps you could imagine how many lines of code it would take in Ceptre to initialise a board for Othello which contains 64 tiles. Moreover, Spira compiles to this version of Ceptre-code and is therefore a facilitating tool for the programmer.

The second and easier way to generate large boards in Ceptre is using nested loops to generate the board. An example of this is shown in `Stage initialization` in Appendix E. Although, this becomes more complex when some tiles start as occupied and others as free. In Haskell, it is simple to keep track of every unique tile by using a map, which in turn can easily be translated to predicates in an initial context.

After we have created the board in Spira we initialise the players. The `players`-function gives us three things: player-predicates, a stage that changes whose turn it is, and a `factConstructor` 'opp', which given a player says who the opponent is.

```
(playernames,stage_next_player,opp)<-players ["black","white"]
let black = head playernames
let white = last playernames
```

For instance, the player's disc (disc referring to a game piece) that is being placed, must be adjacent to an opponent's one. All pieces belonging to the opponent, that are positioned between the newly placed piece and the player's other pieces, are considered as being 'flanked' by the new piece. All flanked pieces are seen as captured by the player and will later be flipped to match their color.

To place a disc we first need a free tile in the `startPosition` presented in Figure 9, then with `opp` we check that the opponent's discs are in the `middlePositions`. Furthermore, the current player needs to be on the `endPosition`.

[2]Othello rules: `https://www.worldothello.org/about/about-othello/othello-rules`

```
p <- newBinding player
opponent <- newBinding player

let place (startPosition:pos) =
     let middlePositions = init pos
         endPosition = last pos
         in
            [ tile [free, startPosition]
            , opp  [p, opponent]
            ]
            ++
            map (\middlePos ->
               makePersistent (tile [opponent, middlePos])
                ) middlePositions
            ++
            [ makePersistent $ tile [p, endPosition]
            ] -@
            [ tile       [p, startPosition]
            , lastPlaced [p, startPosition]
            ]
```

Figure 9: The function `place` in Spira.

In the playing stage we map this place-function on `allPositionPermutations`, which consists of all line permutations containing a `startPosition`, 1-6 `middlePositions` and an `endPosition` — a line being vertical, horisontal or diagonal.

```
stage_play <- stage "play" Interactive p
                      (map place allPositionPermutations)
```

Since placing a disc can flip more than one line of the opponent's discs, we added another stage for doing all the flipping. In that case we need to remember which disc we just placed, that is why we added `lastPlaced` in the `place`-function which was constructed as follows:

```
lastPlaced<-newPredConstructor "lastPlaced" [player,coordType]
```

The pieces can be flipped vertically, horizontally or diagonally on the same move and there is no option for not flipping a disc. It cannot become the next player's turn, unless at least one of the opponent's piece has been flipped.

The flipping-stage presented in Figure 10 is similar to the playing-stage, but instead of looking for a free-tile it uses `lastPlaced`. On the `middlePositions` it changes the owner of the discs from the opponent to the current player.

```
let flip (startPosition:pos) =
     let middlePositions = init pos
         endPosition = last pos
         in
            [ makePersistent $ lastPlaced [p, startPosition]
            ,                  opp        [p, opponent] ]
            ++
            map (\middlePos ->
                               tile       [opponent, middlePos]
                ) middlePositions
            ++
            [ makePersistent $ tile       [p, endPosition] ]
            -@
            map (\middlePos ->
                               tile       [p, middlePos]
                ) middlePositions

stage_flip <- stage "flip" Noninteractive p
                   (map flip allPossiblePositions)
```

Figure 10: The function `flip` and its stage in Spira.

We need to keep `lastPlaced` since we want `stage_flip` to execute recursively until all discs that should be flipped have been flipped, then we can remove it in `stage_remove`

```
stage_remove <- stage "remove_last_placed" Noninteractive p
                      [[lastPlaced [p, coord [x,y]]] -@ []]
```

With the stages constructed thus far, we can construct the game loop. Starting with the current-player placing a tile, then flipping, and finally moving on to the next player:

```
stage_play `fromStageToStage` stage_flip
stage_flip `fromStageToStage` stage_flip
stage_flip `fromFailedStageToStage` stage_remove
stage_remove `fromStageToStage` stage_next_player
stage_next_player `fromStageToStage` stage_play
```

Once `stage_play` fails, meaning a player is out of moves, the player with most discs wins. First we count the number of discs each player has by adding 1 to the `points`-predicate:

```
x <- newBinding nat
xp1 <- x<+1
stage_count <- stage "count_tiles" Noninteractive whoever
    [ [ tile [p, whatever]
      , points [p, x]
      ] -@
      [ points [p, xp1] ]
    ]
```

Then we can present the winner by comparing their score, using the less-than-operator (LT):

```
lt <- initLT
p2 <- newBinding player
stage_winner <- stage "winner" Noninteractive whoever
    [ [ lt [x, y]      -- y is greater
      , points [p, x]
      , points [p2, y] -- y binds with p2
      ] -@
      [ win [p2]]      -- p2 wins
    ]
```

When both players have the same amount of discs on the board, it is a tie, handled by:

```
stage_draw <- initDrawStage
```

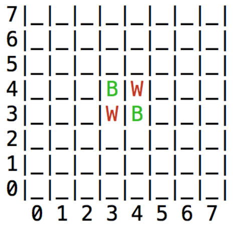
The initial state is when the black and white discs are placed in specific positions of the board, as shown in Figure 11. To actually place the discs in the right positions we introduce the function `addToInitialBoard`. It overwrites the existing tiles to make sure there are no duplicates.

```
mapM addToInitialBoard [(tile [ black, coord [three,four ] ])
                       ,(tile [ black, coord [four ,three] ])
                       ,(tile [ white, coord [three,three] ])
                       ,(tile [ white, coord [four ,four ] ])
                       ]
```

We set the starting stage and player by assigning the player with black discs the role of 'first player' with `initialStageAndPlayer stage_play black`, since the rules state that black always starts.

When Ceptre later on runs, it will output a list with all possible moves that the player can choose from, as shown in Figure 8.

In Spira, our Othello implementation consists of approximately 100 lines of Haskell-code, whereas the generated code in Ceptre contains around 1500 lines of Ceptre-code — a significant difference.



```
7|_|_|_|_|_|_|_|_|
6|_|_|_|_|_|_|_|_|
5|_|_|_|_|_|_|_|_|
4|_|_|_|B|W|_|_|_|
3|_|_|_|W|B|_|_|_|
2|_|_|_|_|_|_|_|_|
1|_|_|_|_|_|_|_|_|
0|_|_|_|_|_|_|_|_|
  0 1 2 3 4 5 6 7
```

Figure 11: Initial state of the Othello board. B and W stand for the black and white player, respectively.

5 Discussion

In this section, we show different perspectives on the result of our project. First we discuss the use of Ceptre as our back-end, followed by future work in Spira and social aspects regarding the project.

5.1 Ceptre as back-end

In this section we discuss our experience with using Ceptre as a back-end, including some problems we came across, wishes for additions in Ceptre and ending with some general remarks on the experience as a whole.

To begin with, some design patterns could be very helpful to have included in the language. Martens [18] presents for example syntactic sugar for rule ordering, which to our knowledge is not included — at least not yet. In addition to that, negation could also be useful to have included, as it is very commonly used, and oftentimes a bit tricky to implement.

The next thing that would be nice is some form of “higher order predicates” that can take other predicates as arguments. See for example the setup-stage of the battleships implementation in Appendix E. Currently it needs repetition for each different boat size, but instead it would be preferable to only have two rules, which each get passed a size argument.

Similarly, some data structures could be helpful to have built-in, to allow some predefined predicates, for example lists with quantifiers (‘exists’ and ‘for all’) or natural numbers with some basic arithmetic. Once again, see the battleships example, or this other example of a dummy list implementation[3].

About the basic arithmetic for natural numbers, we had a problem with occasional combinatorial explosions. For example we were not able to run a multiplication example[4] of 5 times 3 because it tried to allocate too much memory. Possibly related, our first more intuitive implementation of Nim[5] in Ceptre was very impractical to run, because it let us choose every different combination of taking one, two or three sticks — even if the sticks are identical. Solving this issue could potentially lower the need for natural numbers, as that was what we used to resolve this issue.

However, in general most things were fairly intuitive and straightforward to implement, with the exception of the few issues discussed above.

[3] `https://github.com/ErikLjungdahl/spira/blob/master/ceptre/list.cep`

[4] `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/arith.cep`

[5] `https://github.com/ErikLjungdahl/spira/blob/master/ceptre/nim_bad.cep`

5.2 Future work

Here we present some ideas for the future that could improve Spira in various ways.

5.2.1 Create non-rectangular boards

Spira currently only offers helper methods for the creation of rectangular boards. While hexagonal or other shapes are still possible to create, operations to make these shapes just as easy to create is something Spira could contain in the future.

5.2.2 Improving user interface

The visuals we have currently implemented for Spira to present its games, while functional, could see some improvements as they are not well supported by Ceptre. Since our DSL ultimately translates Haskell code into Ceptre code, having a separate application programming interface (API) designed to be used by Ceptre is definitely something that would be appropriate. The API could keep track of the state and possible choices available to the user and display these as a reference to make the designing of the interface easier.

5.2.3 Sanity checking

Since the games are constructed as stages and transitions between those stages, it could be useful to construct a stage diagram as a visual aid for the programmer when creating a board game. On top of that our DSL could give warnings when there are unreachable stages or dead-ends in stages that are not final.

5.2.4 Game-solutions

A completely different approach to the entirety of the project would be to use Ceptre's logic for *game-solutions*. In other words analysing games to come up with ways of solving them, for example finding out the optimal way of playing Tic-tac-toe such that every game ends in either a win or draw. This approach would require a different purpose and would have little basis to be had from this article. Nevertheless, it is as far as we are aware of a relatively unexplored approach to using Ceptre and it would be interesting to see how Ceptre and linear logic in general handles game-solutions. If game-solutions is something you would be interested in exploring, using Cepre's proof-search could come in handy.

5.3 Social aspects

Since our DSL facilitates the production of board games, we believe that our project can be beneficial for society in several ways that will be presented in this section.

There exist several studies that show positive effects of board games. One study examined how playing number games affects the development of number knowledge and early arithmetic. The study shows that mathematical board games such as linear number board games can help 5-year-old children's mathematical development [33].

Other studies also show that board games are effective pedagogical tools in learning; for instance board games can be used as simulations to teach evolution [34]. One example is AntibioGame, which is a game used by medical students that is designed for improving medical training in antibiotics [35]. Furthermore, in the article *Teaching Introductory Programming with Popular Board Games* [36] written by computer scientist Kelvin Sung and mathematical scientist Peter Drake they argue that board games are a *"particularly engaging and relevant topic for programming assignments in introductory programming courses"*. They also suggest that board game characteristics have *"potentials for attracting and engaging female and other traditionally unrepresented groups"*. From these statements we can asses that Spira, a tool for simplifying the creation of board games would similarly be useful in the education of this type; and likewise has the potential of engaging traditionally unrepresented groups in programming.

By introducing Spira as an open-source project, available on GitHub[6], it could reduce the production costs for other developers since the DSL already separates the essential from incidental complexity in Ceptre. For a developer using Spira it will be more efficient, since it will not be necessary for them to do the corresponding work. For instance, a Haskell programmer would find it easier using Spira than learning Ceptre. Similarly, many applications use different open-source libraries to avoid unnecessary productions costs.

[6]Spira source code: `https://github.com/ErikLjungdahl/spira`

6 Conclusion

We have created Spira, a domain-specific language for board games embedded in Haskell, that compiles to Ceptre. In accordance to our purpose, we made Spira easier to use by cleaning up Ceptre's interactive output and added some visualisation. We also showed through case studies that Spira prototypes can require less written code than the equivalent in Ceptre. Spira is open-source, available on GitHub[7]. There you can find working versions of Tic-tac-toe, Connect Four and Othello programmed in Spira to prove its usefulness.

[7]Spira source code: `https://github.com/ErikLjungdahl/spira`

References

[1] J. Romein, H. Bal, and D. Grune, “Multigame - a very high level language for describing board games”, 1995.

[2] S. Lavelle, *About (puzzlescript).* [Online]. Available: `https://www.puzzlescript.net/Documentation/about.html` (visited on 04/27/2020).

[3] C. Martens, “Ceptre: A language for modeling generative interactive systems”, in *Proceedings of the Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2015, November 14-18, 2015, University of California, Santa Cruz, CA, USA*, A. Jhala and N. Sturtevant, Eds., AAAI Press, 2015, pp. 51–57. [Online]. Available: `http://www.aaai.org/ocs/index.php/AIIDE/AIIDE15/paper/view/11536`.

[4] BoardGameGeek staff, *Welcome to boardgamegeek.* [Online]. Available: `https://boardgamegeek.com/wiki/page/Welcome_to_BoardGameGeek` (visited on 03/04/2020).

[5] ——, *Board game categories.* [Online]. Available: `https://boardgamegeek.com/browse/boardgamecategory` (visited on 03/04/2020).

[6] ——, *Board game mechanics.* [Online]. Available: `https://boardgamegeek.com/browse/boardgamemechanic` (visited on 03/04/2020).

[7] ——, *Grid movement.* [Online]. Available: `https://boardgamegeek.com/boardgamemechanic/2676/grid-movement` (visited on 03/04/2020).

[8] *What is sql?*, QuinStreet, Inc. [Online]. Available: `http://www.sqlcourse.com/intro.html` (visited on 04/10/2020).

[9] *What is matlab?*, The MathWorks, Inc. [Online]. Available: `https://se.mathworks.com/discovery/what-is-matlab.html` (visited on 04/10/2020).

[10] (2019). Haxl: A haskell library for effecient, concurrent, and concise data access, Facebook, Inc, [Online]. Available: `https://hackage.haskell.org/package/haxl` (visited on 04/11/2020).

[11] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda, “Feldspar: A domain specific language for digital signal processing algorithms”, 2010. [Online]. Available: `http://www.cse.chalmers.se/~josefs/publications/FeldsparMEMOCODE2010.pdf`.

[12] J. Gibbons, “Folding domain-specific languages: Deep and shallow embeddings”, 2013.

[13] *Algebraic data type*, HaskellWiki. [Online]. Available: `https://wiki.haskell.org/Algebraic_data_type` (visited on 05/12/2020).

[14] *Monad*, HaskellWiki. [Online]. Available: `https://wiki.haskell.org/Monad` (visited on 05/13/2020).

[15] Unity staff, *Unity core platform*, Unity Technologies. [Online]. Available: `https://unity.com/products/core-platform` (visited on 04/29/2020).

[16] S. Lavelle, *Rules (puzzlescript).* [Online]. Available: `https://www.puzzlescript.net/Documentation/rules.html` (visited on 05/07/2020).

[17] J.-Y. Girard, “Linear logic: Its syntax and semantics”, *London Mathematical Society Lecture Note Series*, pp. 1–42, 1995.

[18] C. Martens, “Programming interactive worlds with linear logic”, vol. 136, p. 229, Jan. 2015. [Online]. Available: `http://www.cs.cmu.edu/~cmartens/thesis/thesis.pdf`.

[19] S. Lavelle, *Puzzlescript!* [Online]. Available: https://www.puzzlescript.net/index.html (visited on 05/11/2020).

[20] T. Braüner, *Introduction to linear logic.* Computer Science Department, 1996.

[21] B. Jacobs, "Semantics of weakening and contraction", *Annals of pure and applied logic*, vol. 69, no. 1, pp. 73–106, 1994.

[22] J.-Y. Girard, Y. Lafont, and L. Regnier, "Advances in linear logic", 1995.

[23] T. Sheard, "Putting curry-howard to work", in *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, 2005, pp. 74–85.

[24] F. Pfenning. (2012). Linear logic lecture 01, Youtube, [Online]. Available: https://www.youtube.com/watch?v=Bx2_CGzeMT4 (visited on 04/06/2020).

[25] E. A. Emerson, "Temporal and modal logic", in *Formal Models and Semantics*, Elsevier, 1990, pp. 995–1072.

[26] R. Kaye, "Models of peano arithmetic", 1991.

[27] R. J. Simmons and B. Toninho, "Constructive provability logic", *arXiv preprint arXiv:1205.6402*, 2012.

[28] G. van Rossum. (2020). Python, Python Software Foundation, [Online]. Available: https://docs.python.org/3/library/re.html (visited on 04/16/2020).

[29] *Java documentation.* [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#replaceAll(java.lang.String,%5C%20java.lang.String) (visited on 05/11/2020).

[30] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, *Manifesto for Agile Software Development.* 2001. [Online]. Available: http://www.agilemanifesto.org/.

[31] Atlassian. (2011). Trello, Trello, Inc, [Online]. Available: https://trello.com/ (visited on 04/15/2020).

[32] Python, "Documentation", [Online]. Available: https://docs.python.org/3/library/subprocess.html#.

[33] J. Elofsson, S. Gustafson, J. Samuelsson, and U. Träff, "Playing number board games supports 5-year-old childrens early mathematical development", *The Journal of Mathematical Behavior*, vol. 43, pp. 134–147, Sep. 2016. DOI: 10.1016/j.jmathb.2016.07.003.

[34] M. R. Muell, W. X. Guillory, A. Kellerman, A. O. Rubio, A. Scott-Elliston, O. Morales, K. Eckhoff, D. Barfknecht, J. A. Hartsock, J. J. Weber, and J. L. Brown, "Gaming natural selection: Using board games as simulations to teach evolution", *Evolution*, Jan. 2020.

[35] R. Tsopra, M. Courtine, K. Sedki, D. Eap, M. Cabal, S. Cohen, O. Bouchaud, F. Mecha?, and J. B. Lamy, "AntibioGameő: A serious game for teaching medical students about antibiotic use", *Int J Med Inform*, vol. 136, Jan. 2020.

[36] P. Drake and K. Sung, "Teaching introductory programming with popular board games", 2011. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1953163.1953338.

Appendices

A Tic-tac-toe in Ceptre

```
% Tic-tac-toe, using nats & 'draw on stalemate'
% Comments:
% Diagonals could be done using the specific values, instead of general form.

% Natural numbers
nat     : type.
z       : nat.
s   nat : nat.

% Player/Game related
draw            : pred.
player          : type.
turn    player  : pred.
token   player  : pred.        % Token means this player will have the next turn.
win     player  : pred.
opp     player  player  : bwd. % Backwards chaining predicate.

% Square/board representation
free            nat nat : pred.
occupied player nat nat : pred.

% Stages, main game
stage play = {
play
    : turn A
    * opp A B
    * free X Y
    -o occupied A X Y
    * token B.          % Token means this player will have the next turn.
}
#interactive play.
%% play player _ row col

% We got a token, so someone made their turn.
go/win
    : qui * stage play
    * $token A
    -o stage win.

% 'turn' is still available, so there was no free square and it's a draw.
% (this is essentially for checking "no free square)
% win conditions have already been checked after the last turn.
go/draw
    : qui * stage play
    * $turn A
    -o stage draw.

stage draw = {
draw
    : turn A
    -o draw.
}
```

```
stage win = {       % Consume the token so it doesn't go/play.
win/row
    : token B
    * occupied A X Y            % some square
    * occupied A (s X) Y        % square to the right
    * occupied A (s (s X)) Y    % another square to the right
    -o win A.
win/column
    : token B
    * occupied A X Y            % some square
    * occupied A X (s Y)        % square above
    * occupied A X (s (s Y))    % another square above
    -o win A.
win/diagonal/up
    : token B
    * occupied A X Y
    * occupied A (s X) (s Y)
    * occupied A (s (s X)) (s (s Y))
    -o win A.
win/diagonal/down
    : token B
    * occupied A X (s (s Y))
    * occupied A (s X) (s Y)
    * occupied A (s (s X)) Y
    -o win A.
}

% Next turn
go/play
    : qui * stage win
    * token A
    -o stage play
    * turn A.

% "Objects" / terms
alice : player.
bob : player.
opp alice bob.
opp bob alice.

% Init board to free
context all_free = {            % (x,y)
free    z          z,           % (0,0)
free    z          (s z),       % (0,1)
free    z          (s (s z)),   % (0,2)
free    (s z)      z,           % (1,0)
free    (s z)      (s z),       % (1,1)
free    (s z)      (s (s z)),   % (1,2)
free    (s (s z))  z,           % (2,0)
free    (s (s z))  (s z),       % (2,1)
free    (s (s z))  (s (s z))    % (2,2)
}

% Initial context
context init =
{turn alice}

#trace _ play {init, all_free}.
```

B Tic-tac-toe in Spira

```
module TicTacToe where
import Game

main :: IO ()
main = runGame ticTacToe "game.cep"

ticTacToe :: M ()
ticTacToe = do
    board <- initSimpleBoard 3 3
    let (coordType, coord) = coord_t_c board
    let free = free_v board
    let tile = tile_p board

    nat <- gets numberType
    player <- gets playerType

    (playernames, stage_next_player, opp) <- players ["Oscar","X-ray"]

    -- Pick a free tile and make it occupied by the player
    pos <- newBinding coordType
    p <- newBinding player
    let impl = [tile [free, pos]] -@ [tile [p,pos]]
    stage_play<- stage "play" Interactive p [impl]

    -- A player wins if they have 3 occupied tiles in a row/colum/diagnal
    rules <- inALine 3 (p)
    stage_win <- stage "win" Noninteractive p (rules)


    stage_draw <- initDrawStage

    -- After play we check win condition
    stage_play `fromStageToStage` stage_win
    -- If we can't play, all tiles are filled and it is a Draw
    stage_play `fromFailedStageToStage` stage_draw
    -- If noone has won, we go to the next player
    stage_win `fromFailedStageToStage`stage_next_player
    -- And then the next player gets to play
    stage_next_player`fromStageToStage` stage_play


    initialStageAndPlayer stage_play (head playernames)
```

C Tic-tac-toe compiled to Ceptre

```
% TYPES
player : type.
nat : type.
coordType : type.

% CONSTRUCTORS
z  : nat.
s nat : nat.
coord nat nat : coordType.
free  : player.
Oscar  : player.
X-ray  : player.

% PREDS AND BWDS
tile player coordType : pred.
opp player player : bwd.
opp Oscar X-ray.
opp X-ray Oscar.
pretoken_next_player player : pred.
postoken_next_player player : pred.
pretoken_play player : pred.
postoken_play player : pred.
pretoken_win player : pred.
postoken_win player : pred.
draw  : pred.
pretoken_draw player : pred.
postoken_draw player : pred.

% STAGES AND TRANSITIONS
stage next_player = {
next_player/1
    : pretoken_next_player A
    * opp A B
    -o postoken_next_player B.
}

stage play = {
%% play/1 Turn Col/Row
play/1
    : pretoken_play D
    * tile free C
    -o postoken_play D
    * tile D C.
}
#interactive play.

stage win = {
win/1
    : pretoken_win D
    * tile D (coord E F)
    * tile D (coord (s E) F)
    * tile D (coord (s (s E)) F)
    -o postoken_win D.
win/2
    : pretoken_win D
    * tile D (coord G H)
    * tile D (coord G (s H))
    * tile D (coord G (s (s H)))
    -o postoken_win D.
win/3
    : pretoken_win D
    * tile D (coord I J)
    * tile D (coord (s I) (s J))
    * tile D (coord (s (s I)) (s (s J)))
    -o postoken_win D.
win/4
    : pretoken_win D
    * tile D (coord K (s (s L)))
    * tile D (coord (s K) (s L))
    * tile D (coord (s (s K)) L)
    -o postoken_win D.
}

stage draw = {
draw/1
    : pretoken_draw M
    -o postoken_draw M
    * draw .
}

play_to_win
    : qui
    * stage play
    * postoken_play N
    -o pretoken_win N
    * stage win.

play_failed_to_draw
    : qui
    * pretoken_play O
    * stage play
    -o pretoken_draw O
    * stage draw.

win_failed_to_next_player
    : qui
    * pretoken_win P
    * stage win
    -o pretoken_next_player P
    * stage next_player.

next_player_to_play
    : qui
    * stage next_player
    * postoken_next_player Q
    -o pretoken_play Q
    * stage play.


% INITIAL
#trace _ play
    { tile free (coord z z)
    , tile free (coord z (s z))
    , tile free (coord z (s (s z)))
    , tile free (coord (s z) z)
    , tile free (coord (s z) (s z))
    , tile free (coord (s z) (s (s z)))
    , tile free (coord (s (s z)) z)
    , tile free (coord (s (s z)) (s z))
    , tile free (coord (s (s z)) (s (s z)))
    , pretoken_play Oscar}.
```

D Othello in Spira

```
module Othello where
import Game

main :: IO ()
main = runGame othello "game.cep"

othello :: M ()
othello = do
    (nat,suc,zero) <- gets nats
    player <- gets playerType
    (playernames, stage_next_player, opp) <- players ["black","white"]

    board <- initSimpleBoard 8 8
    let (coordType, coord) = coord_t_c board
    let free = free_v board
    let tile = tile_p board
    tile `outputNames` ["Turn","_/_"]

    coord_eq <- initCoordEQ

    lastPlaced <- newPred "lastPlaced" [player, coordType]

    let black = head playernames
    let white = last playernames

    x <- newBinding nat
    y <- newBinding nat
    output <- newBinding coordType
    p <- newBinding player
    p2 <- newBinding player

    let place (startPosition:pos) =
         let middlePositions = init pos
             endPosition = last pos
             in
                [ opp  [p, p2]
                , tile[free, startPosition]
                ]
                ++
                map (\middlePos ->
                   makePersistent (tile [p2, middlePos])
                    ) middlePositions
                ++
                [ makePersistent $ tile [p, endPosition]
                , coord_eq [startPosition, output] -- For output
                ] -@
                [ tile       [p, startPosition]
                , lastPlaced [p, startPosition]
                ]
    let coordinates = half ++ map reverse half
            where half = concat
                         [[ take n (zip cols rows)
                          | (cols,rows) <-
                                 [ ([0..]    , repeat 0)
                                 , (repeat 0 , [0..]   )
                                 , ([0..]    , [0..]   )
```

```
                            , (reverse [0..n-1]   , [0..n-1]  )
                            ]
                     ] | n <- [3..8]
                    ]

allPossiblePositions <-
    mapM (\positions ->
        mapM (\pos -> do
            x' <- x <+ fst pos
            y' <- y <+ snd pos
            return $ coord [x',y']
        ) positions
    ) coordinates

let impls_play = map place (allPossiblePositions)
stage_play <- stage "play" Interactive p impls_play


let flip' (startPosition:pos) =
     let middlePositions = init pos
         endPosition = last pos
         in
            [                  opp        [p, p2]
            , makePersistent $ lastPlaced [p, startPosition ]
            ]
            ++
            map (\middlePos ->
                               tile       [p2, middlePos]
                ) middlePositions
            ++
            [ makePersistent $ tile       [p, endPosition ]
            ] -@
            map (\middlePos ->
                               tile       [p, middlePos]
            ) middlePositions

let impls_flip = map flip' allPossiblePositions
stage_flip <- stage "flip" Noninteractive p impls_flip


stage_remove <- stage "remove_last_player" Noninteractive p
                      [[lastPlaced [p, coord [x,y]]] -@ []]

points <- newPred "points" [player, nat]

whatever <- newBinding coordType
whoever <- newBinding player
xp1 <- x<+1
stage_count <- stage "count_tiles" Noninteractive whoever -- p and p2 don't have to match
    [ [ tile [p, whatever]
      , points [p, x]
      ] -@
      [ points [p, xp1] ]
    ]

lt <- initLT
win <- newPred "win" [player]
stage_winner <- stage "winner" Noninteractive whoever
```

```
    [ [ points [p, x]
      , points [p2, y]
      , lt [x, y]
      ] -@
      [ win [p2]]
    ]
stage_draw <- initDrawStage

stage_play `fromStageToStage` stage_flip
stage_flip `fromStageToStage` stage_flip -- Flip as much as we can
stage_flip `fromFailedStageToStage` stage_remove -- When we no longer can flip,
                                                 -- we remove last_placed pred
stage_remove `fromStageToStage` stage_next_player
stage_next_player `fromStageToStage` stage_play

stage_play `fromFailedStageToStage` stage_count
stage_count `fromStageToStage` stage_count --  Keep counting
stage_count `fromFailedStageToStage` stage_winner -- When done counting,
                                                  -- see who the winner is
stage_winner `fromFailedStageToStage` stage_draw -- If noone wins, it's a draw.

three <- zero <+ 3
four <- zero <+ 4
mapM addToInitialBoard [(tile [ black, coord [three,four ] ])
                       ,(tile [ black, coord [four ,three] ])
                       ,(tile [ white, coord [three,three] ])
                       ,(tile [ white, coord [four ,four ] ])
                       ]
mapM_ (\player -> addPredToInit (points [player, zero])) playernames

initialStageAndPlayer stage_play black
```

E Battleships in Ceptre

```
% Battle ships
% Notes:
% With this approach, lots of repetition.
% Possible alternative:
% Generate boats by using rollback (choose one boat/square at a time).

% Could probably ignore ship geometry past setup phase...
% as long as they're setup as proper ships, it doesn't really matter.
% But want to preserve geometry.

% Nats
nat     : type.
z       : nat.
s nat   : nat.

% Player
player      : type.
alice       : player.
bob         : player.
opp player  player : bwd.
opp alice   bob.
opp bob     alice.

% Board
max/x nat : pred.       % Board width
max/y nat : pred.       % Board height
gen/y nat nat : pred.   % Token for initialization

% Operations on the board.
% "player" means that player's board.
% For example: a guess by alice will be "guess bob X Y"
hit     player nat nat : pred.
miss    player nat nat : pred.
guess   player nat nat : pred.  % A guess was made here, to be processed
open    player nat nat : pred.  % No boat placed here (for setup)
free    player nat nat : pred.  % No miss/hit here (for game)

% Boats
% Boat is a list of "positions",
% each represented by the coordinate X Y
boat    : type.
nil     : boat.             % Empty list
cons nat nat boat : boat.   % Recursive case

% Misc
owns        player boat : pred. % Player owns some boat
token/boat  player nat  : pred. % Token for placing a boat
turn        player : pred.
token/turn  player : pred.      % This player will get a turn next
maybe/boat  player : pred.      % To check if player has any boats left
has/boat    player : pred.      % Player did have a boat
no/boat     player : pred.      % Player did not have any boat
win         player : pred.

% Sinking a boat
sunk player boat : bwd.
```

```
sunk/Z : sunk P nil.             % The empty boat is sunk
sunk/S : sunk P (cons X Y Bs)    % Sink one piece, and the rest of it (if hit).
       <- hit P X Y
       <- sunk P Bs.

% Guess on 'player' 'x' 'y' given 'boat'
% Was the boat hit?
% The boat was hit if the guess was on the same square as any of the boat pieces.
was/hit player nat nat boat : bwd.
was/hit/R  : was/hit  P X Y (cons X' Y' Bs)   % Recursive case
             <- was/hit P X Y Bs.
was/hit/W  : was/hit P X Y (cons X Y Bs).     % Found witness

% Stage initialization based on "max" value.
% Basically a nested loop over X and then over Y.
% For i=0 to X:
%   put token: gen i Y
% Then for each such token,
% recursively (over Y) generate:
%   free alice x Y_j
%   open alice x Y_j
%   free bob x Y_j
%   open bob x Y_j
stage init = {
initfree/XZ
    : max/x z
    * $max/y Y
    -o gen/y z Y.
initfree/XS
    : max/x (s X)
    * $max/y Y
    -o gen/y (s X) Y
    * max/x X.
initfree/YZ
    : gen/y X z
    -o free alice X z
    * open alice X z
    * free bob X z
    * open bob X z.
initfree/YS
    : gen/y X (s Y)
    -o free alice X (s Y)
    * open alice X (s Y)
    * free bob X (s Y)
    * open bob X (s Y)
    * gen/y X Y.
}

init_to_setup
    : qui * stage init
    -o stage setup.

% Game setup, players concurrently place their boats.
% Consume 'open' to ensure boats do not overlap.
% This could possibly be done with some "rollback" approach instead.
% Note: lots of repeated code. Could this be done with functions/vectors instead somehow?
stage setup = {
fill/Z
```

```
    : token/boat P z
    * open P X Y
    -o owns P
    (cons X Y nil).
fill/two/horizontal
    : token/boat P (s z)
    * open P X Y
    * open P (s X) Y
    -o owns P
    (cons (s X) Y
    (cons X Y nil)).
fill/two/vertical
    : token/boat P (s z)
    * open P X Y
    * open P X (s Y)
    -o owns P
    (cons X (s Y)
    (cons X Y nil)).
fill/three/horizontal
    : token/boat P (s (s z))
    * open P X Y
    * open P (s X) Y
    * open P (s (s X)) Y
    -o owns P
    (cons (s (s X)) Y
    (cons (s X) Y
    (cons X Y nil))).
fill/three/vertical
    : token/boat P (s (s z))
    * open P X Y
    * open P X (s Y)
    * open P X (s (s Y))
    -o owns P
    (cons X (s (s Y))
    (cons X (s Y)
    (cons X Y nil))).
}
#interactive setup.

setup_to_game
    : qui * stage setup
    -o stage game.

% Main game, simply make a guess.
% Note: we consume "free" so we can't guess on this position again.
stage game = {
do/guess
    : turn P
    * opp P P'
    * free P' X Y
    -o guess P' X Y
    * token/turn P'.
}
#interactive game.

game_to_checks
    : qui * stage game
    -o stage check_stage.
```

```
stage check_stage = {
% Confirm hit
check/hit
    : $owns P Bs
    * guess P X Y
    * was/hit P X Y Bs
    -o hit P X Y.
% Consume sunk ships
sink
    : owns P Bs
    * sunk P Bs
    -o ().
}

checks_to_boat_stage
    : qui * stage check_stage
    -o stage boat_stage.

% Check that both players have at least one boat.
% (negation for win condition: "if !boat, then win opp")
stage boat_stage = {
check/boat
    : maybe/boat P
    * $owns P Bs
    -o has/boat P.
% If 'guess' wasn't removed in last stage, it was a miss
cleanup
    : guess P X Y
    -o miss P X Y.
}

% A 'maybe/boat' token was not consumed,
% so that player didn't have a boat.
% This works because of "qui", introduced at run-time when no more rules can fire.
no_boat
    : qui * stage boat_stage
    * maybe/boat P
    -o no/boat P
    * stage win.

% Both players had a boat, go again.
go_next_round
    : qui * stage boat_stage
    * has/boat P * has/boat P'
    * token/turn P
    -o maybe/boat P * maybe/boat P'
    * turn P
    * stage game.

stage win = {
do/win
    : opp P P'
    * no/boat P'
    -o win P.
}

context init = {
```

```
max/x (s (s (s z)))
,max/y (s (s (s z)))
,token/boat alice (s z)
,token/boat bob (s z)
,token/boat alice z
,token/boat bob z
,turn alice
,maybe/boat alice
,maybe/boat bob
}

#trace _ init init.
```