A DSL for Puzzles

Patrick Aldis

07/9/23

Contents

Abstract

The aim of the project was to create a framework for expressing and solving simple grid puzzles. The framework includes a domain specific language written in Haskell for stating puzzles, infrastructure to solve these puzzles, and methods to present them to the user.

1 Introduction

This project aims to create a framework for **defining**, **solving** and **inspecting** a wide range of puzzles. The puzzles tackled are numerical puzzles typically found in a newspaper, that enforce a set of constraints on a grid. Such problems are referred to as *constraint satisfaction problems*. They are covered in more detail in section 3.2, however at a high level they involve a set of variables (the cells) and a set of rules the variables must adhere to (the rules of the puzzle). If one is to try and solve a puzzle of this form programatically, there are currenly two options, outlined in section 3.2.3:

1. Describe the problem in a general purpose programming language using a constraint library. This allows a developer to express constraint problems in a language they are familiar with. The program can then be complied and run to produce a solution.

2. Describe the problem in a general constraint programming language. This can then be executed by the language's solver to give a complete solution.

Currently both of these options have drawbacks, explained in section 3.2.3. At its core, the project introduces a domain specific language for describing grid puzzles. This has many advantages over the options above. Firstly, having control over the language syntax means that we can design an efficient and comprehensible way of expressing these problems. Secondly, we can write tools specific to puzzles expressed in the DSL. This means that not only can we write a solver to produce solutions, but we could write a UI to view and inspect them – and that this applies to any puzzle expressed in this language. The structure of the project is outlined below.

1.1 Project Components

The project consists of multiple components, but the two most significant are:

- A *Domain Specific Language* (DSL), that can express the rules for a puzzle in Haskell source code.
- A *Solver*, that can complete a partially filled puzzle expressed in the DSL.

In my research proposal I outlined a further two goals:

- A *File Format* for writing the DSL in plain text
- A *Front-End UI* for inspecting puzzles

These were considered stretch goals at the time, however I feel that given the progress I have made they are within reach of the final project.

Part I

Background

2 Project Management

I have used several methods to try and ensure that development of the project runs smoothly, including:

- A Github-hosted source repository, providing **Continuous Integration** (CI)
- The project being built using the **Nix** package manager, providing system-agnostic builds
- Frequent note-taking in markdown using the **Obsidian** note taking app.

Currently, when committing to the Github repo, a Github-hosted Nix builder will build the project and return a success or failure CI status, depending on whether the project compiles. Having continuous integration means that I can easily find breaking changes simply by looking through the commit history.

2.1 Nix

Nix is a *purely functional package manager*, originally created by Eelco Dolstra for his PhD thesis [1]. It consists of both the Nix language, allowing you to write packages, and the Nix package manager, allowing you to build packages. One feature of Nix is that it builds all packages, from dependencies up, in isolation, assuming nothing else is installed [2]. All dependencies have to be explicitly described before building using the Nix language. This means that the build process for a package is system agnostic and makes it ideal for a remote builder to evaluate the project.

It does have its downsides. The learning curve for the language is very steep – the documentation is inaccessible, with no clear entry point for the language. This has meant it has taken a large amount of time setting up the repository, however once set up, it requires very little maintenance. I settled on using IOHK's `haskell.nix` [3] infrastructure, as it allowed the most flexibility.

Nix stores any packages or build artifacts that are created during a package build in a separate `\nix` directory at the system root, referred to as the Nix store [2]. During a build, if a requried dependency is already in the store, the store version is used. This means that building a Nix project a second time round will be much quicker, as the system already has all the required dependencies. I've used a platform called **Cachix** [4] to host a Nix store on a remote server. Whenever the project is built by the remote builder it attempts to build the project using the Cachix Nix store, meaning the previous build artifacts are used.

2.2 Obsidian

I have found that note-taking has been an invaluable resource before and while coding. The Haskell language encourages the workflow of defining types and structures before defining procedures, and hence having some space to jot ideas about my project's types has been incredibly useful. The note-taking app **Obsidian**[5] was the best fit for my use case as the notes are written in markdown syntax, and the app provides many features to visualise notes.

3 Background and Research

Initially in the Project Specification, I claimed that the project would try and solve newspaper puzzles. Given the huge variety of puzzles found in newspapers, it was necessary to restrict my focus to some subset that:

- Covers a wide range of use cases
- Can be abstracted easily

To this end, I decided to focus on puzzles like *sudoku*, that concern filling in a partially completed grid, subject to some constraints. This seemed to be a good fit for the project because:

- There are a large number of examples available (see section 3.1 below).
- Such puzzles can be phrased as a constraint satisfaction problem (see section 3.2.1)

3.1 Example Puzzles

The following resources have been invaluable for finding examples of grid puzzles:

- **nikoli.co.jp** [6]- Homepage for the Japanese puzzle company Nikoli, responsible for creating *sudoku*, *bridges*, *slitherlink* and other similar puzzles.

  The website lists some of their most renowned puzzles and accompanying rules.

- **braingle.com** [7] - The website's *Puzzlepedia* section, described as:

  > A encyclopedic collection of various puzzles, mind games, brain teasers and resources for people who like challenging problems.

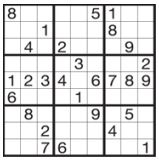
  braingle.com contains a puzzle catalogue with accompanying links and information. The grid puzzles subsection lists 54 grid puzzles, explaining the rules behind every puzzle, and in many cases includes links to pages where you can play each one.

- **puzzle-nurikabe.com** [8] - Part of a series of puzzle websites for various grid-puzzles explaining the rules of, and allowing you to play, *nurikabe* in your browser.

The website provides links to other websites in the series, where you can browse the rules for other grid-based puzzles.

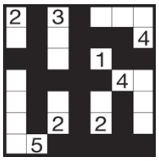
Using the previous resources, I selected the following puzzles to serve as canonical examples:

sudoku [9]



- Cells must contain elements from $\{1, \ldots, 9\}$.
- **Rows** must have unique entries.
- **Columns** must have unique entries.
- **Blocks** must have unique entries.

nurikabe [10]



Colour cells **Black** such that:

- You cannot fill cells containing numbers.
- A number describes how many **White** cells are in a continuous block. Each area of **White** cells contains only one number.
- The **Black** cells must form one connected component.
- The **Black** component cannot contain a 2×2 filled square.

hashiwokakero [11]

- **Cells** can be either an **Island** or a **Bridge**.
- Each **Island** has a marked capacity, n, indicating that the island must have exactly n adjacent **Bridges**.
- **Bridges** must connect islands, and can only go in straight lines, connecting to bridges in the same direction.
- A **Bridge** cell can double up to become two bridges, granted that it is connected to other two-bridge cells.

hitori [12]

1	1	2	4	3	5
1	1	5	4	4	6
4	6	6	2	1	1
6	3	3	3	5	4
2	3	4	1	6	5
2	5	4	6	2	5

Colour cells **Black** such that:

- No numbers are repeated in any **Row** or **Column**.
- **Black** cells never touch.
- **Non-Black** cells form one connected component.

skyscrapers [13]

	4	1	2	2	2	
2	1	5	3	4	2	3
3	2	4	5	3	1	3
2	4	3	2	1	5	1
1	5	1	4	2	3	3
2	3	2	1	5	4	2
	2	4	3	1	2	

- **Cells** represent skyscrapers in a city block. Each cell can take values in $\{1, \ldots, 5\}$, indicating the height of the skyscraper on this cell.
- The edges of the puzzle indicate how many skyscrapers are visible along that **Row**/**Column**. Taller skyscrapers obscure smaller skyscrapers.

3.2 Theory of Constraint Satisfaction

As mentioned previously, the problem can be phrased as a constraint satisfaction problem (CSP). Rossi's Handbook of Constraint Programming [14] defines this as:

Definition

Definition 3.1. A CSP, $\mathcal{P}$, is a triple $\langle X, D, C\rangle$ where:

- $X = \langle x_1, x_2, \ldots, x_n\rangle$ is and n-tuple of **variables**
- $D = \langle D_1, D_2, \ldots, D_n\rangle$ is the corresponding tuple of **domains**: $x_i \in D_i$
- $C = \langle C_1, C_2, \ldots, C_n\rangle$ is a t-tuple of **constraints**.

Where each **constraint** is defined as a pair $\langle R_{S_j}, S_j\rangle$, where R_{S_j} is a relation on the variables S_j. Here S_j is referred to as the scope of the constraint. This relation defines the set of acceptable values for the variables S_j under this constraint.

3.2.1 Puzzles as a CSP

We can phrase a grid puzzles in this way.

- **Variables** (X)

  The variables correspond to the values of the puzzle cells, and are arranged in an $n \times m$ grid:

  $$\begin{bmatrix} x_{11} & \ldots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \ldots & x_{nm} \end{bmatrix}$$

  Giving a total of $n \cdot m$ variables.

  $$X = \langle x_{11}, x_{12}, \ \ldots \ , x_{mn}\rangle$$

- **Domains** (D)

  Groups of cells in puzzles typically take the same ranges of values:

  - All the entries in a *sudoku* take values in $\{1, \ldots, 9\}$.
  - Cells in *hashiwokakero* are either **Islands**, taking values in $\{1, \ldots, 8\}$, or **Bridges**, taking values in $\{Single, Double\}$

  I have decided to frame this in the following form:

  There exists a set of possible cell *types*, T, which lists the different kinds of cells present. We can then, for each $x_{ij} \in X$, write it's type t_{ij} and form the matrix S (I've referred to this as the structure matrix).

$$S = \begin{bmatrix} t_{11} & \dots & t_{1m} \\ \vdots & \ddots & \vdots \\ t_{n1} & \dots & t_{nm} \end{bmatrix} \quad \text{where } t_{ij} \in T$$

For simplicity, we will assume that the domain of each $t \in T$ is some finite subset of $\mathbb{N}$ – that is that x_{ij} takes a finite number of integer values. If $\mathcal{D} : T \to \mathcal{P}(\mathbb{N})$ recovers the domain of a type, then $x_{ij} \in \mathcal{D}(t_{ij})$. Using this notation we can write the set of variable domains, D, as follows:

$$D = \langle \mathcal{D}(t_{11}), \mathcal{D}(t_{12}), \ \dots \ , \mathcal{D}(t_{nm}) \rangle$$

- **Constraints** (C)

  Puzzles can contain constraints of arbitrary generality, hence we can take the definition from above:

  $$C = \langle C_{11}, C_{12}, \ \dots \ , C_{nm} \rangle$$

  One thing to note when designing the DSL was that often in puzzles constraints are parametrised over cell types – i.e. a constraint will apply to all cells of type t_1. These constraints can be introduced in the DSL with the `ForAll` constructor. Notice this is different from C_{ij} as this applies only to one singular variable.

Notice that our puzzle is defined by the following objects:

- T – The types of cells involved
- S – The structure matrix
- C – The constraints

Assuming that the defintion of T is included in the definition of S, we can think of a puzzle as the tuple (S, C).

3.2.2 Constraint Solvers

There are known algorithms to solve problems framed as a CSP. If your problem involves only boolean variables, it can be solved by a class of solver called a **SAT Solver**, designed to solve the **Boolean Satisfiability Problem** [15]:

Definition

Definition 3.2. Given a boolean expression E in conjunctive normal form as a conjunction of n clauses, where each clause is a disjunction of literals, the **boolean satisfiability problem** is to find an variable assignment such that E is true.

The reality is that the majority of problems involve more complex variables than just `true` or `false`, in which case a different class of solver is needed. An **SMT Solver** (Satisfiability Modulo Theories Solver) extends the capabilities of a **SAT Solver** to allow for reasoning in terms of types other than just booleans. These other types are encapsulated by the idea of a Σ-theory. This is explained in [16]:

Definition

Definition 3.3. Let T be a Σ-theory. A Σ-formula P is T-satisfiable if there *exists a model* M such that $M \models A$ and $M \models P$

An SMT solver generates a satisyfying assignment to a Σ-formula. Known SMT solvers include Microsoft's `z3` [17], or the open source project `CVC5` [18].

3.2.3 Constraint Programming Languages

Constraint Programming is a programming paradigm for describing constraint problems. Programs written in a constraint programming language are run to produce a satisfying output. This is exactly what the DSL portion of this project hopes to accomplish, however in the specific domain of solving puzzles as defined above. Current examples of CP languages are:

- `ECLiPse` [19] : First conceptualized in the 90s as an approach to easily specify constraint problems. The language is implemented in `prolog`.
- `Gecode` [20] : An open source `C++` constraint toolkit with an emphasis on performance.
- `MiniZinc` [21] : Created at Monash University, a modelling language that compiles to `FlatZinc` – a solver independent language for expressing constraint problems. This is the most recent of all the languages above, formalized in 2007.

All of these languages are general-purpose constraint languages and allow you to express general constraint problems. However, for solving simple puzzles the expressiveness of these languages are not needed.

3.3 Haskell

I decided to write the DSL in Haskell for the following reasons:

- **Type System** : The language supports recursive data types, making it perfect for expressing syntax trees, where expressions are nested inside expressions. Having a static type system means that programs can be type-checked in real time, detecting errors before compilation, with the help of the Haskell Language Server (HLS) [22]. Additionally, as the language is purely functional, almost every part of a program has a type and can be type checked, meaning that the majority of errors can be resolved before compilation.
- **Reliability** : The type system allows me to write reliable code. The Haskell compiler and HLS can detect whether a function is partial or not. This is incredibly helpful for validation, both for user inputs and from a development persective to ensure different parts of the project will interact correctly. If all functions are total, it is less likely a breaking error will occur internally as the language can predict what type of data it will receive. As this project has a lot of moving parts (see section 4), this is very important.
- **Purity** : The pure nature of the language lends itself very well to writing DSLs – syntax can be built up from smaller pieces and stitched together with combinators. Hence I can very easily isolate sections of code for testing during development, as each function is a black box, depending only on its inputs. Its purity also means that we can reason about a program mathematically, using equational reasoning.
- **History** : Haskell has a great reputation for writing DSLs. The process is well documented, and many good resources are available [23][24][25].
- **Libraries** : The language has a rich ecosystem of packages, specifically in the areas of this project (see section 3.4).
- **Testing Suite** : Haskell provides a wide array of powerful testing libraries [26][27][28], enabling you to phrase tests that check properties of functions. This will be invaluable for checking that properties of the language hold (see section 11).

3.4 Libraries

Haskell provides an extensive collection of libraries, indexed on **Hackage**, offering a wide array of packages for *constraint satisfaction* and *text parsing*.

3.4.1 Constraint Satisfaction

In the initial project specification, several options were considered for constraint packages:

- `monadiccp` [29] : Its accompanying website explains how the project is solver-independent, meaning constraints are solved without using an external SMT solver such as those listed in section 3.2.2.
- `mad-props` [30] : A simple constraint solver using the principle of constraint propagation. Like `monadiccp`, this library also doesn't need an external SMT solver.
- `sbv` [31] : A framework for expressing general constraint problems, to then be solved by an external solver.

I eventually settled on `sbv`, as the documentation for the package was very thorough, providing a wealth of examples in comparison to the other two options. I also decided it was advantageous to use a known solver – such solvers are well optimised and offer the best performance.

3.4.2 Text Parsing

`parsec`[32] advertises itself as "an industrial strength parser library", offering "extensive libraries, good error messages and is fast". It allows you to define text parsers and provides a framework to combine and manipulate parsers. Many relatives of `parsec` exist, including `attoparsec`[33] and `megaparsec`[34], each offering a different set of features. The maintainer of `megaparsec`'s blog [35] provides a good comparison between them:

> - `parsec` has been the "default" parsing library in Haskell for a long time. The library is said to be focused on quality of error messages. It however does not have good test coverage and is currently in maintenance mode.
> - `attoparsec` is a robust, fast parsing library with focus on performance. It is the only library from this list that has full support for incremental parsing. Its downsides are poor quality of error messages, inability to be used as a monad transformer, and limited set of types that can be used as input stream.
> - `megaparsec` is a fork of parsec that has been actively developed in the last few years. The current version tries to find a nice balance between speed, flexibility, and quality of parse errors. As an unofficial successor of parsec, it stays conventional and immediately familiar for users who have used that library or who have read parsec tutorials.

As the text parsing in this project will be limited to very small text files, quality of error messages takes a priority over speed. Hence, `megaparsec` or `parsec` are the best options.

3.5 User Interface

It is important that the project is very user friendly. Solving grid puzzles can be done in any constraint programming language, however it is not neccessarily

easy to do so. The DSL and project should streamline this process and make solving puzzles accessible to everyone. As the language is specific to puzzles, it's important that we leverage this and create a set of tools that are convenient for a puzzle-solving audience. This means having a UI that can:

- **Input Puzzles** : Having a way to input puzzles graphically makes this tool much more user friendly. Grid puzzles are graphical in nature; being able to edit a cell directly rather than entering it in a comma separated text file is invaluable.
- **Display Solutions** : Solutions should be displayed graphically in a way that is readable, for example differentiating between cells that have been entered by the user and those produced by the solver.
- **Highlight Errors** : A potential future feature would be for the solver to be able to decide which cells violate constraints, and highlight them. This would help with identification of impossible puzzles.

3.5.1 Front-end Technologies

Options I considered for a front-end included:

- **Terminal** : The terminal can provide a UI with minimal work. However, it lacks the user-friendliness of graphical front-ends. It is a good choice for a provisional interface, however not for the finished product. Having a more intelligent terminal UI library like `ncurses`[36] might alleviate this concern however; the Haskell library `brick`[37] could be a good option.
- **HTML/TypeScript** : HTML offers flexibility when structuring an interface, while TypeScript provides a type-safe scripting of the behaviour. TypeScript synergises particularly well with Haskell with its type system and functional programming support. This option offers the best versatility, and arguably the most visually pleasing final product. Unfortunately I also believe it will be the most work for myself; I have very little experience in web development and would be programming in TypeScript for the very first time.

In the project objectives (see section 4 below), the UI will be completed last. This means that I can assess, based on the time remaining, which option is the best fit.

Part II

Objectives

4 Current Objectives

I have formalised the following objectives for the project:

1. A DSL that can describe simple puzzles
   Requirements:
   - The DSL should be **expressive** enough to describe a wide range of puzzles. A good measure of this is whether it can represent a range of puzzles from the resources listed in section 3.1.
   - The DSL should be **human readable**. This means clarity of syntax should take priority over having an efficient representation, and it should be easy to discern the rules of a puzzle from looking at the code.

2. Infrastructure to provide solutions to the DSL
   Requirements:
   - A **DSL parser** that converts Haskell code into a CSP that can be solved by the `sbv` library.
   - **Convenience functions** that will generate a solution to DSL code using the parser. Prior to objective *(4.)* this should be rendered in a useful form to the user in the terminal.

3. A file format to make the language more accessible **[Stretch Goal]**
   Requirements:
   - A **specification** for a file format that describes a puzzle
   - A **text parser** that will convert such a file into DSL code, using one of the libraries mentioned in section 3.4.2.

4. A user interface to render the puzzles **[Stretch Goal]**
   Requirements:
   - A **user interface** for visualizing solutions to problems - A **specification** for how to render different types of puzzles

Part III

Implementing the DSL

Recalling the objectives laid out in section 4:

1. A DSL that can describe simple puzzles
   Requirements:
   - The DSL should be **expressive** enough to describe a wide range of puzzles. A good measure of this is whether it can represent a range of puzzles from the resources listed in section 3.1.
   - The DSL should be **human readable**. This means clarity of syntax should take priority over having an efficient representation, and it should be easy to discern the rules of a puzzle from looking at the code.

5 Prototyping

Before starting on development of the main project, I thought it was necessary to create a small prototype constraint solver. This still posed some issues. I have no previous experience with writing a DSL, and despite the amount of resources available I still struggled. Before trying to create the final DSL I created a proof of concept that described and solved simple integer constraint problems, such as:

$$\forall x \in \{1, \ldots, 9\} : \forall y \in \{1, \ldots, 9\} : x \neq y$$

This was showcased at my project's video presentation.

5.1 Transition from the Prototype

I found that making the jump from the proof of concept above to a DSL that could solve puzzles was quite large. The prototype language above had no sense of structure between the variables – making statements like:

"x and y share the same column"

just didn't make sense. This meant that I had to formalise types to describe the structure of the puzzle, so that we could reason about where the variables are located. These types came about after several iterations, with a couple unsuccessful attempts. I found that the best approach was to write notes about the intended purpose and formalise types on paper or in a markdown file, rather than in the IDE.

6 Core Data Types

When describing a puzzle as a CSP in section 3.2.1, I stated that a single puzzle can be thought of as a tuple consisting of two objects:

- C – The constraints
- S – The structure of the puzzle

In the DSL, S and C are separated, so that the rules of a puzzle aren't specific to one particular arrangement of cells. Describing a puzzle in the language consists of two Haskell data types:

- `PuzzleClass` : Defines the rules for a given **class** of puzzles (e.g. *sudoku*, *hitori*). This is analgous to C, without S.
- `PuzzleInstance` : Corresponds to a particular instance of a **class**. Here the structure of the puzzle, and any known values are defined. This is analagous to (C, S)

A user will express the rules for a **class** of puzzles using the DSL, and then solve for a particular instance.

6.1 Phrasing Rules

`Rule` is the primary object in the DSL and is used to either introduce variables or constrain cells. It is recursive and has two constructors:

```
data Rule
  = ForAll CellType (CellVar -> [Rule])
  | Constrain (Expression Bool)
```

Where:

- `ForAll t f` is used to introduce a variable. It iterates over all cells of type `t`, applying `f :: CellVar -> [Rule]` to each one.
- `Constrain expr` is the terminal case of `Rule` and indicates that `expr` should always be true.

I decided to represent the syntax tree using a Higher Order Abstract Syntax representation (or HOAS representation). This means that lambda functions are implemented using the Haskell language's implementation, rather than being implemented from scratch. This means that the handling of variables is much simpler, as we can write expressions such as:

```
ForAll t1 (x ->
  ForAll t2 (y ->
    x .== y
  )
)
```

without having to worry about misspelling variables. It also means that expressions in the language are simply Haskell expressions and can still be type-checked

by the Haskell Language Server, meaning that DSL expressions can be checked for type errors. It has had some significant downsides however – It has meant that many DSL constructs now require lambda expressions that otherwise wouldn't, and has made parsing significantly more difficult. The alternative, First Order Abstract Syntax (or FOAS), is that we implement these features as constructors in the DSL datatype. The function that parses DSL expressions would then have to deal with variable scoping and substitution.

6.2 Creating Expressions

The `Expression` type represent a single value that can be produced from the variables in scope. The type is parameterised by the result produced. The terminal case is `SBV`'s symbolic type:

```
data Expression a
  = Exp (SBV a)
  | If (Expression Bool) (Expression a)
  | Count (CellRule Bool) (SWord8 -> Expression a)
  | Sum (CellRule Word8) (SWord8 -> Expression a)
  | ConnectedBy (CellRule Bool) CellVar CellVar (SBool -> Expression a)
```

I elected to use `SBV`'s symbolic types, as it meant that I could use the library's tools for expressing propositional logic, rather than coding my own. However, this has had some drawbacks in terms of parsing (see section 13).

6.3 Properties

Each cell in a puzzle may have multiple attributes. For example, cells in *akari* need the following information:

Attribute	Description	Value Type
Capacity	Should the cell be black, the number written on the cell indicating the number of adjacenct bulbs	`Int`
Colour	Whether the cell is shaded black or white	`Bool`
Bulb	Whether the cell should contain a lightbulb	`Bool`

To accomodate this, a puzzle is split into several layers:

Layer	Description	Examples from *akari*
Value	A single layer of *Unknown* information. This is the objective of the problem.	**Bulb**
Property	Layers of additional *known* information that are used to determine the **value** layer	**Capacity**, **Colour**

These layers are determined by the types of cells in the puzzle. In the definition for `CellType` we can see that it has field labels `value` and `properties`, corresponding to the allowed values and properties:

```
data CellType = CellType
  { typeName :: String,
    value :: CellEntrySet,
    properties :: Map String CellPropertySet
  }
```

7 Features

After implementing `ForAll` bindings and `Expressions`, which seemed essential to describing any puzzle, I had to stop and reevaluate my strategy going forward. I shortlisted several features that could be implemented in the language. These are detailed in sections 7.1 to 7.5.

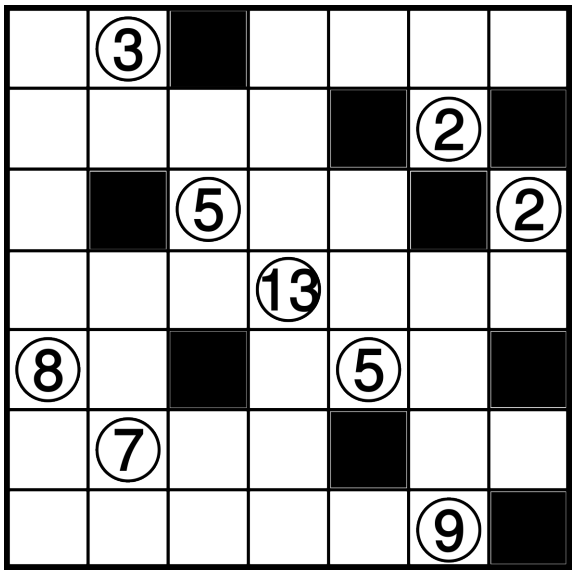


Figure 1: A solution to the puzzle *kurodoko*

7.1 Count

One possible feature is having the ability to count the number of cells that satisfy a boolean condition. Given a function f : `Cell` $\rightarrow$ `Bool`, a `Count` statement would determine the number of cells that return `True`. It would allow us to state rules like:

```
ForAll cell (\x -> (Count <cond>) .== val "Bridges" x)
```

We could then enforce conditions like:

1. A cell numbered k must be adjacent to exactly k white cells (*nurikabe*)
2. Each row must have exactly n black squares, where n is indicated at the the start of the row (*tilepaint*)
3. A cell with a labelled number, m, must have a total of m white squares in horizontal and vertical directions before reaching a black square (*kurodoko*, see figure 1)

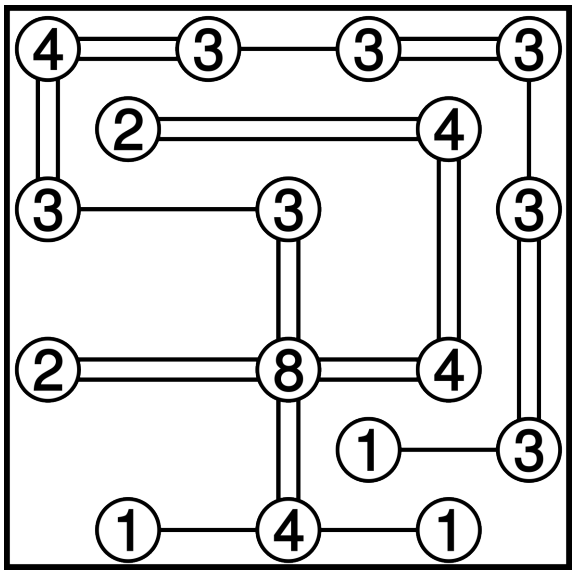


Figure 2: A solution to the puzzle *hashi*

7.2 Sum

Another possible feature is having a method to iterate over all cells with a function $f : \texttt{Cell} \to \texttt{Int}$ and sum the result. It would allow making statements like:

```
ForAll cell (\x -> (Sum <func>) .== 10)
```

So that we can enforce conditions such as:

1. All rows, columns and diagonals must sum to the same number (*magic square*)
2. An island must have exactly i bridges adjacent, where each cell can have up to 2 bridges (*hashi*, see figure 2)
3. Given a numbered cell with value j in the border of the puzzle, consecutive white cells directly to the right or downwards must sum to j (*kakuro*)

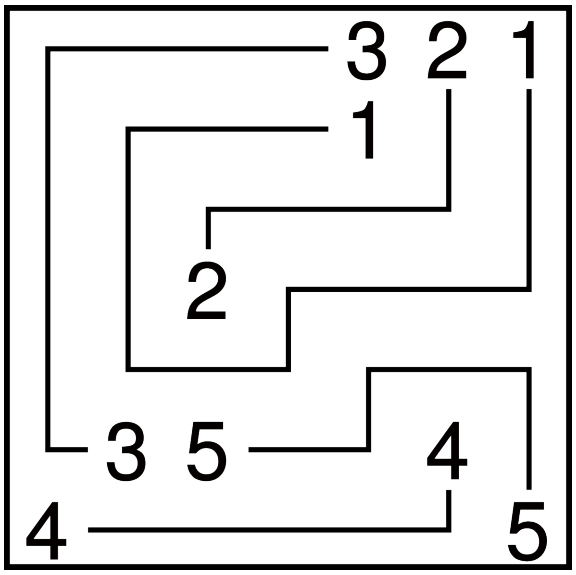


Figure 3: A solution to the puzzle *Numberlink*

7.3 Connectedness

After labelling every cell with some $f : \texttt{Cell} \to \texttt{Int}$, can we decide if cell x lies in the same connected component as cell y? Doing so would allow us to enforce conditions such as:

1. Black cells form one connected component – i.e. every black cell is connected to every other black cell (*nurikabe*)
2. Given two distinct cells x, y with the same value, there exists a path from x to y (*numberlink*, see figure 3)
3. A light bulb illuminates all squares in its row and column, until a black square or the edge of the puzzle is reached (*akari*)

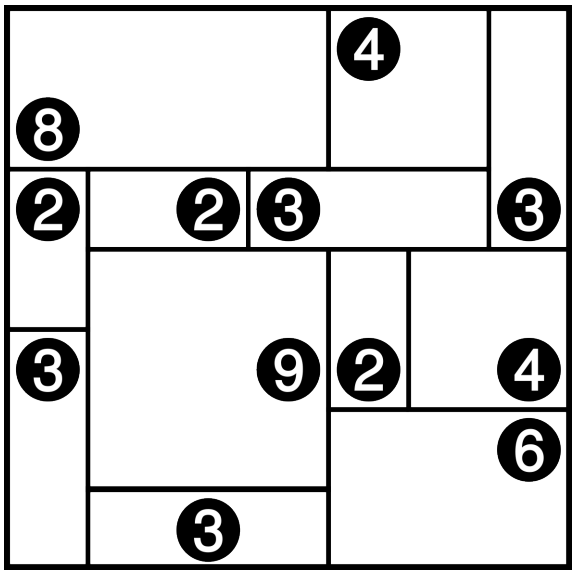


Figure 4: A solution to the puzzle *shikaku*

7.4 Rectangular Regions

Having some way of testing to see if a connected component is rectangular would be another useful feature. Many puzzles listed in the resources provided in section 3.1 have entries grouped in rectangular regions, where inside these regions the cells must abide by certain rules.

It would allow us to enforce conditions like:

1. Each rectangle must contain only one number cell (*shikaku*, see figure 4)
2. If a rectangle contains a number, the region must contain that many black cells (*heyawake*)
3. Cells marked with a pencil tip must form the tip of a *pencil* – a $1 \times n$ rectangular region ending in a pencil tip (*pencils*)

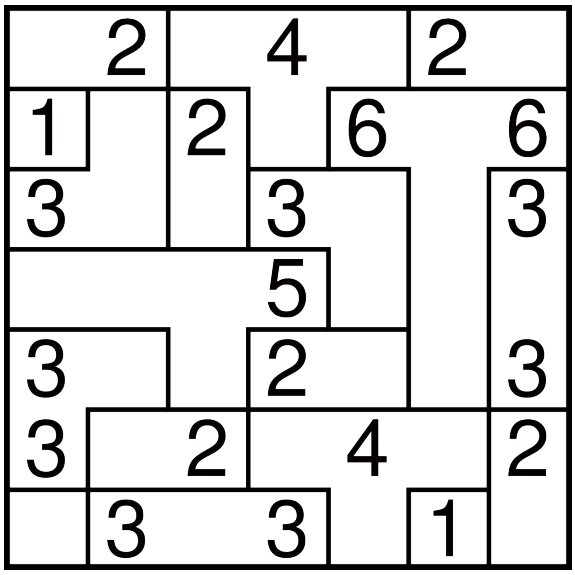


Figure 5: A solution to the puzzle *fillomino*

7.5 Arbitrary Regions

Having some way to define and reason about arbitrary regions would unlock other puzzle types. In addition to rectangular regions, many of the resources listed in section 3.1 describe puzzles made up of not only rectangular regions, but of regions defined by polyominos [1].

1. Each arbitrary region contains exactly 2 black squares (*norinori*)
2. Given a cell x marked with a number k, x must belong to a region consisting of exactly k cells (*fillomino*, see figure 5)

[1] A polyomino of size n is a collection of n squares of equal size arranged with coincident sides [38].

7.6 Implementation

Before implementing any features, I observed that some features could be considered subsets of others. Noticably:

1. **Count** is a subset **Sum**. Recall that:

   - **Count** counts the size of the subset of cells.png that satisfies some boolean condition $b : \texttt{Cell} \to \texttt{Bool}$:

     $$\text{Count}(b) := \left| \{c : b(c) = \texttt{True}\} \right|$$

   - **Sum** iterates over every cell with a function $f : \texttt{Cell} \to \texttt{Int}$ and sums the resulting values:

     $$\text{Sum}(f) := \sum_c f(c)$$

   However, given any boolean function $b : \texttt{Cell} \to \texttt{Bool}$, we can find a function, $B : \texttt{Cell} \to \texttt{Int}$ such that Count(b) = Sum(B). Simply take:

   $$B(c) = \begin{cases} 1 & b(c) = \texttt{True} \\ 0 & b(c) \neq \texttt{True} \end{cases}$$

   So by implementing **Sum**, we get **Count** for.png free.

2. Producing **Arbitrary Regions** requires a notion of **Connectedness**.

   Although some puzzles (e.g *fillomino*, *makaro*) come with regions already annotated, others require that the solver produces these regions as the solution of the puzzle (e.g *five-cells*, *double-choco*). To do this, the solver will need some notion of **Connectivity** to avoid mistakes such as categorising two opposite corners of the puzzle as one region. Hence categorising regions requires **Connectivity**

7.7 Methodology for Choosing Features

Implementing all of these features was not possible given the time frame. Hence, it was necessary to compile a list of puzzles from [6], along with the features they required (see Tables 3 and 4 below). As discussed previously, any puzzle requiring the **Count** feature can also be solved using **Sum**, and thus these two categories have been combined:

Table 3: A selection of puzzles from [6] and the features required to implement them

Puzzle	**Con**	**Sum**	**Rect**	**Arbi**
Hashi		True		
Hitori	True			
Akari	True	True		
Kakuro	True	True		
Kurodoko	True	True		
Norinori				True
Nurimeizu	True	True		
Fillomino				True
Makaro				True
Moon-or-Sun				True
Numberlink	True			
Nurikabe	True	True		
Pencils	True	True	True	
Ripple Effect				True
Satogaeri	True			True
Shikaku			True	
Tilepaint		True		
Usowan		True		
Yosenabe				True
Heyawake	True	True	True	

Table 4: The frequency of subsets of features

	Count
{**Sum**}	3
{**Con**}	2
{**Rect**}	1
{**Arbi**}	5
{**Sum**, **Con**}	5
{**Con**, **Arbi**}	1
{**Sum**, **Con**, **Rect**}	2

8 Sum

By my phrasing of the `Sum` feature in section 7, a `Sum` statement requries defining a function f : `Cell` $\rightarrow$ `Int` that associates a number with every cell. The main obstacle to coding this was that in its current state, the DSL had no way of phrasing such a function. To this end, I created a new datatype, `CellRule`, that would encode functions of the form `Cell` $\rightarrow$ `a`:

```
type CellRule a = [CellTypeRule a]

data CellTypeRule a
  = For CellType (CellVar -> Expression a)
```

Whereas the `Rule` type introduces multiple variables into an `Expression`, `CellRule` only introduces one – the current cell. The `For` constructor acts as a branch of a piecewise function, where `For <t> <r>` returns `r` when the input is of type `t`. Writing functions in this way using the exisiting `Expression` type meant that I could leverage the already existing syntax, so that a `CellRule` could be written in terms of any of the language features.

Implementing the `Count` feature afterwards really confirmed that Haskell was the right language choice for this project. Using the language's pattern matching, I was able to write a function that would convert a `Bool` cell rule into one that returns an `Integer` (in the fashion described in section 7.6).

9 Connectedness

I realised there were two possible approaches to this problem:

- Deal with connectivity between variables, on a more local scale. We would have some equivalence relation $x \sim y$ that would decide whether x and y lie in the same connected component. For the sake of this document, I will refer to this as **local connectivity**.
- Pre-compute all connected components in the grid. Deciding if x and y lie in the same component reduces to looking up their component labels, and seeing if they agree. I will refer to this as **global connectivity**.

I struggled immensely with both approaches, and failed to produce a way of measuring connectivity. My progress was hindered in part by the black-box nature of the `sbv` library. The library's `allSat` function, the backbone of the solving portion of this project, has type signature:

```
allSat :: Satisfiable a => a -> IO AllSatResult
```

Where the typeclass `Satisfiable` represents "an expression with constraints, potentially returning a boolean" [39] – our list of cell rules. This function acts as a bridge between `sbv`'s symbolic types and the external solvers, to produce a result.

9.1 Local Connectivity

9.1.1 Depth-First Search

An intuitive approach would be to view connectivity as a graph search problem. We see each cell in B as a node in a graph, G, which is connected to up to four neighbours depending on whether the value of the neighbours agree with the value of the cell. The question:

> "is x connected to y"

now reduces to whether there exists a path from x to y in G. The elementary imperative algorithm for exploring this is the **Depth First Search**, outlined in [40]. Rather than re-phrasing the algorithm in a functional manner, I initially tried to use mutable data structures [2] and the `ST` monad to preserve its imperative nature, however this proved troublesome. The algorithm requires checking the current value of a cell. As cells have type `SBool` rather than `Bool`, traditional `if <cond> then <exp> else <exp>` statements don't make sense, and the symbolic equivalent, `ite` (if then else), has to be used. However the type signature of `ite`:

```
ite :: Mergeable a => SBool -> a -> a -> a
```

Only allows for results from the typeclass `Mergeable`, of which `Mergeable a => ST a` is not a member.

9.1.2 Path Connected Relation

An alternative method I tried was inspired by the mathematical definition of **path-connectivity**. The definition given in [41] states:

Definition

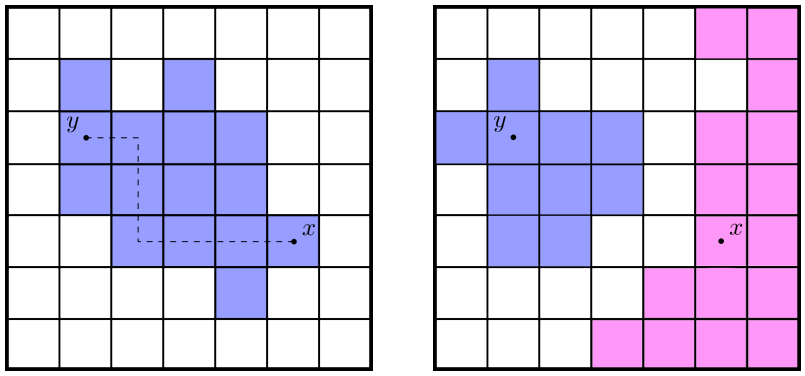
Definition 9.1. A *topological space* X is **path-connected** if any two points of X can be joined by a *path* in X

However, in the case of our grid of cells, a path is a sequence of cells (x_i) all having the same value, where x_k is a neighbor of x_{k+1}. This is illustrated in figure 6. We can phrase this as a recursive equivalence relation:

Define the relation $(\sim)$ as follows:
We say $x \sim y$ if *all* of the following are true:

- x and y have the same value
- x and y are neighbors, or there exists z, a neighbor of x, such that $z \sim y$

[2] As Haskell prides itself on being a pure language, almost all data structures are immutable by default. To use mutable data structures one has to use the State (or `ST`) monad, which keeps track of the mutations made to variables.



(a) x and y are *path connected*, as a path exists between them

(b) x and y are not *path connected*, as no such path exists

Figure 6: An illustration of path connectivity

My code for this (see appendix B) faced an issue where the solver would search indefinitely for a solution, but never terminate. I suspected that this might be due to the solver interpreting the relation as having an infinite search space, however it is difficult to be sure.

9.2 Global Connectivity

I found two main approaches for doing this, given the binarized cells, B:

- An imperative-style flood fill. This consisted of repeated application of the *depth-first search* algorithm described above, until all components of the grid are explored. This meant that we could label components with a unique number, given to all cells in that component. The entire procedure can be thought of as a function $f : \mathbb{M}_{n\times m}(\texttt{SBool}) \to \mathbb{M}_{n\times m}(\texttt{SWord8})$, behaving as follows:

$$f\left(\begin{bmatrix} T, T, F \\ T, F, F \\ F, F, T \end{bmatrix}\right) = \begin{bmatrix} 1, 1, 0 \\ 1, 0, 0 \\ 0, 0, 2 \end{bmatrix}$$

  We can then deduce if x is connected to y by computing $f(B)$ and looking up their respective component numbers to see if they agree.

- Using `sbv` to find a series of sets that are a partition of B into its connected components. This involved writing constraints to express what it means for a group of sets to a partition another. I later found an inherent flaw in one of the constraints, mentioned in section 9.2.1.

9.2.1 Partitioning Sets

This idea was coined by the fact that `sbv` has datatypes for working with symbolic sequences and sets. This meant that an SMT solver could produce a set of arbitrary length that satisfies a given number of conditions. This felt like the natural way to phrase connectivity, as a list of sets that partition B but also correspond to the connected components. I started with the following definition of a partition:

> A list of sets, $P = [X_1, \ldots, X_n]$ is a **partition** of X if:
> 1. The union of all X_i is X
> 2. For every $i \neq j$, X_i and X_j are disjoint

I then expressed constraints in Haskell to find a list of sets subject to the following:

> 1. P $= [B_1, \ldots, B_n]$ is a *partition* of B
> 2. For each $b_1, b_2 \in B_i$, $b_1 = b_2$
> 3. Each B_i is *connected*

Where I took the following definition of **connected**:

> $X \subset C$ is called **connected** if for every $x \in X$, $\exists x' \in X$, where x' is a neighbor of x

However, this produced incorrect partitions, labelling all `True` values as one connected component despite having two distinct regions. This is due to a mistake in my definition of what it means for a set to be **connected**. The definition disallows singleton sets, however not much else. It is more akin to the mathematical definition of a set being **open** [42] :

Definition

Definition 9.2. Let (X, d) be a *metric space*, and $U \subseteq X$. We say that U is **open** in X if for every $x \in U$ there exists $\epsilon_x > 0$ such that $B_{\epsilon_x} \subseteq U$

10 Puzzles Expressed

With the current feature set, the following puzzles from section 7.7 can be described (and solved):

- *sudoku*
- *hashi*
- *tilepaint*
- *usowan*
- *yosenabe*
- *norinori*
- *ripple effect*

11 Testing Suite

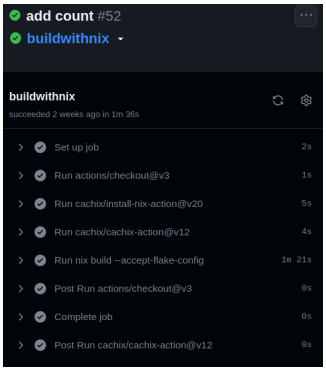
As outlined in section 2.1, I have used the functional package manager Nix to package and build the project. Although the language definitely has many downsides, one of the advantages is that its sandboxed nature lends itself very well to testing and remote building. Github's Actions [43] feature allows exectution of arbitrary jobs when code is uploaded to the repository. I have used this to build and run my repository's testing suite with Nix every time new code is pushed. This has proved an invaluable resource, allowing me to easily identify breaking code and consistently track my progress by adding tests every time I implement a new feature.

For implementing the testing suite, I decided to use the `hspec` library. It provides a framework to use the Haskell testing libraries `QuickCheck`, `SmallCheck` and `HUnit`, so that a variety of types of tests can be used. Of these libaries, I mainly used the unit testing provided by `HUnit`. It allowed me to write black-box tests for module functions during development, ensuring that they behaved as expected. In addition to this, as new language features were implemented, I made sure to write an example puzzle that could be solved with these new features. This ensured that as the feature set of the language increased, none of the previously solvable puzzles were affected.

This workflow of writing tests before writing code helped immensely during development. It meant that I could easily identify which function was not behaving as expected before more code was written. This synergises particularly well with functional languages like Haskell that encourage breaking up functions into smaller functions for reusability, reinforcing my belief that Haskell was the correct choice of language for this project.

```
Language Features
  Bools [✔]
  Properties [✔]
  Summation
    Module Functions
      cellSum [✔]
    Puzzle Tests
      Sum Puzzle [✔]
  Counting
    Puzzle Tests
      Count Puzzle [✔]
  Connectivity
    Module Functions
      slookup (literals) [✔]
      neighbors (literals) [✔]
      dfs (literals) [✔]
      slookup (unknowns) [✔]
      neighbors (unknowns) [✔]
Puzzles
  Sudoku
    Easy [✔]
```

(a) The result of running the test suite in the command line



(b) Continuous integration

12 Solving

The objective outlined in section 4 was:

> 2. Infrastructure to provide solutions to the DSL
> Requirements:
> - A **DSL parser** that converts Haskell code into a CSP that can be solved by the `sbv` library.
> - **Convenience functions** that will generate a solution to DSL code using the parser. Prior to objective *(4.)* this should be rendered in a useful form to the user in the terminal.

12.1 Solving Overview

The procedure for solving puzzles is as follows:

```
do
  emptyBoard >>= writeLiterals >>= applyConstraints
```

Most solving computations rely on monads[3].

This is because, during symbolic computations, the `sbv` library keeps track of context and variable constraints using a `Symbolic` monad. In the computation above, the result of each stage of the computation is being passed into the next stage, where more constraints are enforced. The computation reads:

1. `emptyBoard` : Generate an $m \times n$ board of symbolic variables, with their domains restricted such that they can only take values as specified by their cell types. Pass the resulting symbolic variables to the next computation.
2. `writeLiterals` : Given that this whole computation aims to finish a

[3]Monads are a type that allows computations to be sequenced and combined, while keeping functions pure. For a more comprehensive description, see here.

partially complete puzzle, we must at some point enter the values we know. This part of the computation takes the inputted variables, and sets them to their known values where neccessary. It then passes these variables into the next computation.

3. `applyConstraints` : Finally, the constraints outlined in the DSL are applied.

12.2 Struggles

12.2.1 Familiarity with the `sbv` library

Initially, I found starting the solving backend difficult as I had never worked with a constraint solving library previously. In addition to this, the `sbv` library is such a diverse library that although there is a wealth of examples, many of them are for different types of constraint problems. Finding the correct constructs in such a large library is also quite intimidating, however luckily Haskell's type system helps narrow the list down.

12.2.2 `ForAll` Quantifier

In tandem with the change in structure, I had to change the behaviour of the `ForAll` binding. In the prototype the binding iterates over all possible values of a variable, and tests to see if constraints are satisfied. This really serves the same purpose as the constrains themselves – the desired functionality is for the quantifier to parameterize constraints over the *cells*. This way constraints can apply to groups of cells as described in section 3.2.1.

In addition to the objects being iterated over, I had to slightly alter the intended behaviour of the `ForAll` binding. To illustrate this, I will take an an example from *sudoku*. I wanted to describe the idea that entries are all unique. Using predicate logic, we might assume it would be phrased as follows, given the set of variables X:

$$\forall x \in X : \forall y \in X : \qquad \text{value}(x) \neq \text{value}(y)$$

However, we have to account for the fact that x can equal y:

$$\forall x \in X : \forall y \in X : \qquad x \neq y \implies \text{value}(x) \neq \text{value}(y)$$

This mistake caused constraints that can never be satisfied: a variable must always equal itself. It also highlighted a recurring pattern when writing constraints – when a new variable is introduced in a `ForAll` binding, we almost always assume it is distinct from the other variables in scope. The constraints of sudoku, written in a form closest to the DSL, are:

$\forall x \in Cells : \forall y \in Cells :$

$$
\begin{array}{rcll}
x \neq y & \Longrightarrow & (x, y \text{ same } \textbf{col} & \Longrightarrow \text{ value}(x) \neq \text{value}(y)) \quad \text{[Col constraint]} \\
\wedge\, x \neq y & \Longrightarrow & (x, y \text{ same } \textbf{row} & \Longrightarrow \text{ value}(x) \neq \text{value}(y)) \quad \text{[Row constraint]} \\
\wedge\, x \neq y & \Longrightarrow & (x, y \text{ same } \textbf{box} & \Longrightarrow \text{ value}(x) \neq \text{value}(y)) \quad \text{[Box constraint]}
\end{array}
$$

This is not particularly intuitive. For this reason I decided that a `ForAll` binding should exclude variables that are already in scope. This meant keeping track of all variables in scope, and blacklisting what is introduced by the bindings.

12.2.3 Data Format

When a constraint problem is solved in `sbv`, the default `Show` instance displays a long list of the variable names and their values:

```
X0:0 = 1 :: Word8     X1:0 = 9 :: Word8     X2:0 = 5 :: Word8
X0:1 = 7 :: Word8     X1:1 = 2 :: Word8     X2:1 = 4 :: Word8
X0:2 = 3 :: Word8     X1:2 = 6 :: Word8     X2:2 = 8 :: Word8
X0:3 = 2 :: Word8     X1:3 = 5 :: Word8     X2:3 = 7 :: Word8
X0:4 = 6 :: Word8     X1:4 = 4 :: Word8     X2:4 = 1 :: Word8
X0:5 = 9 :: Word8     X1:5 = 8 :: Word8     X2:5 = 3 :: Word8
X0:6 = 5 :: Word8     X1:6 = 3 :: Word8     X2:6 = 9 :: Word8
X0:7 = 8 :: Word8     X1:7 = 1 :: Word8     X2:7 = 2 :: Word8
X0:8 = 4 :: Word8     X1:8 = 7 :: Word8     X2:8 = 6 :: Word8
```

Not only was this not clear for an end user, but it highlighted a problem that had to be solved internally – when the variables and constraints are sent by `sbv` to the solver, the structure of the problem is lost. The solver only views this as satisfying constraints on 81 named variables, and will return a long list of 81 named variables as the result.

To solve this, I wrote functions that would extract the correct values for each cell, and arrange them in an array. Having a grid structure to the output meant that I could correctly draw the solutions, first in the console, then graphically, as follows:

```
Initial Problem:    Solution 1:
1 ? 3 ? ? ? ? 8 ?   1 7 3 2 6 9 5 8 4
? ? 6 ? 4 8 ? ? ?   9 2 6 5 4 8 3 1 7
? 4 ? ? ? ? ? ? ?   5 4 8 7 1 3 9 2 6
2 ? ? ? 9 6 1 ? ?   2 8 7 4 9 6 1 3 5
? 9 ? 8 ? 1 ? 4 ?   3 9 5 8 7 1 6 4 2
? ? 4 3 2 ? ? ? 8   6 1 4 3 2 5 7 9 8
? ? ? ? ? ? ? 7 ?   4 5 9 6 3 2 8 7 1
? ? ? 1 5 ? 4 ? ?   8 3 2 1 5 7 4 6 9
? 6 ? ? ? ? 2 ? 3   7 6 1 9 8 4 2 5 3
```

Part IV

Stretch Goals

13 File Format

The goals set at the start of the project were:

1. A DSL that can describe simple puzzles
   Requirements:
   - The DSL should be **expressive** enough to describe a wide range of puzzles. A good measure of this is whether it can represent a range of puzzles from the resources listed in section 3.1.
   - The DSL should be **human readable**. This means clarity of syntax should take priority over having an efficient representation, and it should be easy to discern the rules of a puzzle from looking at the code.
2. A file format to make the language more accessible **[Stretch Goal]**
   Requirements:
   - A **specification** for a file format that describes a puzzle
   - A **text parser** that will convert such a file into DSL code, using one of the libraries mentioned in section 3.4.2.

In its current state, the embedded DSL wouldn't satisfy the second requirement of **human readability**. When designing the text format I aimed to create a straightforward format that satisified this, that would allow end users without Haskell proficiency to express puzzles. Unfortunately, I had significant difficulties doing this, as mentioned in section 13.2.

I identified `parsec` and `megaparsec` in section 3.4.2 as candidates for parsing libraries. I settled on `megaparsec`, as it's `ParseError` type is much richer than `parsec`'s error type.

13.1 Ideal Textual Format

As explained in section 6, describing a puzzle consists of two objects:

- A `PuzzleClass` object, describing the rules for a given *class* of puzzles.
- A `PuzzleInstance` object, used to solve a particular *instance* of a *class*. Here the arrangement of the cells is defined.

The text format should allow a user to both write rules for a *class* of puzzles, and solve for a particular *instance*. Below I settled on an idea for an ideal text format for describing a *class* of puzzles. A `PuzzleClass` contains the following

information:

```
data PuzzleClass = PuzzleClass
  { name :: String,
    rules :: [Rule],
    types :: [CellType]
  }
```

In the ideal text format that I created, I decided to split this data into two sections as follows:

```
---
<< YAML HEADER >>
---

<< RULES >>
```

Where the YAML header will include:

- The **Puzzle Name**
- The **Cell Types**

And the **Rules** will be written underneath.

13.1.1 YAML Header

I settled on the following format:

Generic Format	Example

`---`
`name: <puzzleName>`
`cells:`
`  <cellName>:`
`  Value: <celltype>`
`  <propertyName>: <celltype>`
`  .`
`  .`
`  .`
`  <cellName>:`
`  .`
`  .`
`  .`
`---`
	`---`
	`name: PuzzleNameHere`
	`cells:`
	`  NumericType1:`
	`  Value: Int`
	`  Prop1: Bool`
	`  Prop2: Int`
	`  NumericType2:`
	`  Value: Int`
	`  Prop1: Bool`
	`  BoolType1:`
	`  Value: Bool`
	`  Prop2: Int`
	`---`

The advantage to using a known format like YAML is that it is familiar to many, and an existing parser [44][45] can be used to cut down on development.

13.1.2 Rules

It was important to make sure that phrasing rules met the requirements outlined in section 4, specifically:

> - The DSL should be **human readable**. This means clarity of syntax should take priority over having an efficient representation, and it should be easy to discern the rules of a puzzle from looking at the code.

The following sections outline how certain language features could be written in an ideal text format:

13.1.2.1 ForAll Bindings When written in the DSL, it is common for the puzzles listed in section 3.1 to contain nested `ForAll` expressions. It was important that it was easy to represent this in the text format. Hence, the following syntax was planned:

```
ForAll x, y CellType:
  <rules>
```

This represents a nested `ForAll` expression, and can be expanded to:

```
ForAll x CellType:
  ForAll y CellType:
    <rules>
```

13.1.2.2 Values It was a priority that it was easy to reference the value of the cell. Currently in the Haskell eDSL, retreiving a cell value is quite cumbersome. I think notationally it makes sense to reference the value of a cell with just the cell variable name. Using this notation, enforcing that the value of cell `x` is 1 simply requires writing `x == 1`.

13.1.2.3 Properties In addition to values being easy to retrieve, it was also important that it be simple to state expressions involving cell properties. For this reason I planned to write properties using the format `<property> <variableName>`. For example:

```
col x > col m && row x == row m
```

13.2 Problems with Parsing

As mentioned in section 6.1, the DSL uses a Higher Order Abstract Syntax tree (HOAS), rather than a First Order Abstract Syntax tree (FOAS). Unfortunately, this meant that I was unable to create a working parser. I've outlined the problems with using a HOAS representation below :

13.2.1 FOAS vs HOAS

To reiterate the difference between FOAS and HOAS, I have created two different definitions of the language's `ForAll` constructor, one implemented with higher-order syntax and one with first-order syntax.

	HOAS	FOAS
Data Type	`ForAll CellType (CellVar -> [Rule])`	`ForAll CellType String [Rule]`

Haskell Code	`ForAll t ( x ->`
	`  [Constrain x .== 1]`
	`)`
		`ForAll t "x" (`
		`  [Constrain (Var "x") .== 1]`
		`)`

Variables	• Variables are handled using Haskell's lambda functions. When translating to an `SBV` expression, `x` takes a value by applying the function to an input.	• When translating to an `SBV` expression, variables are handled internally. During the translation the code must keep track of which variables are in scope and their values, such that the correct values can be substituted

13.2.2 Parsing Higher Order Syntax

A problem with HOAS arises when parsing: It becomes very difficult to programmatically create higher-order expressions. In this representation, writing variables can only be done using Haskell lambda syntax. When parsing a text file it becomes increasingly difficult to express things in this way. Take the following simple example:

```
ForAll x Cell:
  If x < 2:
    x == 0
```

Writing a parser to convert this into a higher-order expression involves creating a function that, when an input is applied, propagates the value to fill the `x`'s. One method of doing this is creating a **context** vector containing the Haskell variables, and their matching strings. The **context** is then propagated deeper into the expression, so that whenever the parser encounters a matching string, it can substitute the value of the variable.

This is exactly the process of variable substution required for a FOAS representation, however it is done at a parsing level.

13.3 Expressable Concepts

This section showcases various concepts that could be expressed using the text format, assuming that all language features mentioned in section 7 are implemented. This gives an insight into the potential of the final product.

13.3.1 Existence and Uniqueness

If I wanted to express that a cell must exist satisfying some condition, this can be done using the `Count` feature. Given a condition f : `Cell` $\rightarrow$ `Bool`, to require **existence** all we need to enforce is that *at least* one such cell exists. That is:

```
0 < Count (For x Cell: f x)
```

And similarly to require **unique existence** we can enforce that *exactly* one such cell exists:

```
1 == Count (For x Cell: f x)
```

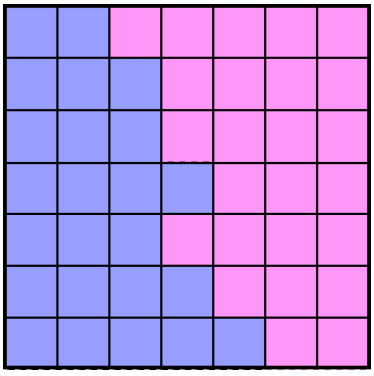
In the puzzle *norinori*, it is required that black squares come in pairs, placed like dominos. This could be phrased as a unique existence constraint in the following way:

```
ForAll x Cell:
  If x:
    1 == Count (For y Cell: y neighbor x && y)
```

If x takes on the value `True` (i.e. is a black square), it must have exactly one neighbour that is also `True` (is also a black square)

13.3.2 Regions

Regions can be expressed in the language by giving the cells of a puzzle one additional property: an integer representing the current region it belongs to. Cells in the same region have the same region number.



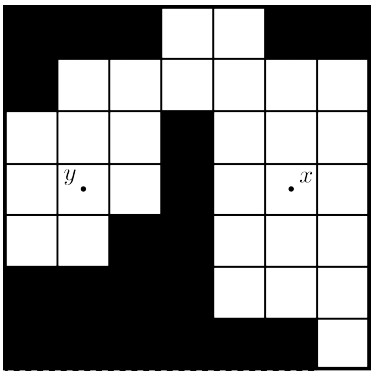
(a) A puzzle with two regions

1	1	2	2	2	2	2
1	1	1	2	2	2	2
1	1	1	2	2	2	2
1	1	1	1	2	2	2
1	1	1	2	2	2	2
1	1	1	1	2	2	2
1	1	1	1	1	2	2

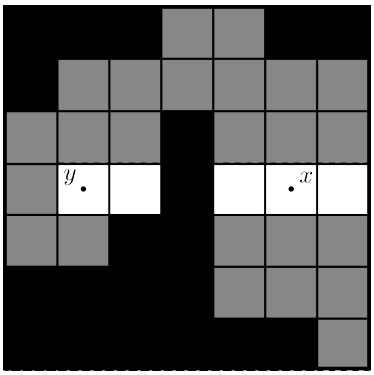
(b) Numerical representation

13.3.3 Light Rays (*akari*)

In the puzzle *akari*, the objective is to illuminate the white cells on the board by placing lights. These lights must be placed conforming to constraints. Each bulb illuminates all cells in the same column and row that are in the eyesight of a person standing in a cell (i.e. they aren't obscured by a black cell).



(a) A puzzle may require determining if two cells x and y are within eyesight of each other.



(b) This reduces to determining if x is connected to y after applying f, described below.

This idea of illuminating cells occurs quite frequently in the puzzles listed on [6] or [7], and require the puzzler to determine whether one cell is in eyesight of another. This could be expresssed in the DSL using connectivity. Given a board of black and white cells, with row and column given by $\texttt{row}, \texttt{col} : \texttt{Cell} \to \texttt{Int}$ and value given by $\texttt{val} : \texttt{Cell} \to \texttt{Bool}$. One cell c_2 is in eyesight from another, c_1, if c_1 is connected to c_2 after binarising the board with the function:

$$f(c) = \begin{cases} \texttt{val}(c) & \texttt{row}(c) = \texttt{row}(c_1) \vee \texttt{col}(c) = \texttt{col}(c_1) \\ \texttt{False} & \text{otherwise} \end{cases}$$

This concept also occurs in the puzzle *kakuro*, where numbers on the edge of the puzzle indicate the total sum of the contiguous white cells immediately to the right. This element of the puzzle could be phrased as follows:

```
ForAll m Marked:
  m = Sum (
    For x Unmarked:
      If (Connected m x Via (
        For y Unmarked: y && col x > col m && row x = row m
        For y Marked: False
      )
      Then: x
    )
  )
```

13.3.4 Nurikabe

Nurikabe could be expressed in the following form:

```
---
name: Nurikabe
cells:
  Marked:
    Value: Bool
    Capacity: Int
  Unmarked:
    Value: Bool
---

ForAll x Marked:
  x = False
  Capacity x == Count (
    For y Marked: True
    For y Unmarked: Connected x y Via (
      For z Marked: not z
      For z Unmarked: not z
    )
  )

ForAll x, y Unmarked:
  Connected x y Via (
    For z Marked: z
    For z Unmarked: z
  )
```

Part V

Appraisal

14 Evaluation Against Project Objectives

14.1 Objective 1

1. A DSL that can describe simple puzzles
   Requirements:
   - The DSL should be **expressive** enough to describe a wide range of puzzles. A good measure of this is whether it can represent a range of puzzles from the resources listed in section 3.1.
   - The DSL should be **human readable**. This means clarity of syntax should take priority over having an efficient representation, and it should be easy to discern the rules of a puzzle from looking at the code.

I think I was very effective at managing my time to maximise the number of puzzles I could **express**. Classifying the 20 puzzles in section 7.7 by the features required to implement them proved a great time-saving decision. It allowed me to prioritise the features that would lead to the most applications. Although attempting to implement the connectivity feature (see section 7.3) ultimately failed and took a significant amount of time, I think attempting to do so was justified based on the number of puzzles that require it. In addition, I theorised various potential approaches that will help any future contributors should they choose to continue implementing this feature.

I do think that the **readability** of the eDSL is an area that could be improved upon. The decision to use a HOAS representation (see section 13.2.1) has meant that syntax has become cumbersome in places. The intention was that the file-format outlined in section 13 would render these issues unimportant, however this remains to be implemented. One possible solution is to rewrite the syntax in the future using a FOAS representation, or proceed with development of a text parser.

14.2 Objective 2

2. Infrastructure to provide solutions to the DSL
   Requirements:
   - A **DSL parser** that converts Haskell code into a CSP that can be solved by the `sbv` library.
   - **Convenience functions** that will generate a solution to DSL code using the parser. Prior to objective *(4.)* this should be rendered in a useful form to the user in the terminal.

In its current state, the DSL is fully functional and produces solutions to problems. Haskell code is converted into a representation that is understood by the constraint library `sbv`. `sbv` is then used to call an external SMT solver to produce solutions. Retrieving solutions from `sbv` programmatically did prove difficult (see section 12.2.3), however I managed to solve this issue. Addionally, to this I have created functions to print solutions to puzzles in the terminal. I am extremely happy with my progress in this area and would consider this objective achieved.

14.3 Objective 3

3. A file format to make the language more accessible [**Stretch Goal**]
   Requirements:
   - A **specification** for a file format that describes a puzzle
   - A **text parser** that will convert such a file into DSL code, using one of the libraries mentioned in section 3.4.2.

Although this goal was considered a **stretch goal**, I think I have made progress in this area. Should a contributor decide to start work on the project in the future, I have identified the problems with the language structure that need to be addressed. This includes either using a FOAS representation, or implmenting a parser with scoping of variables. In addition to this, I have outlined a **specification** for an ideal text format. Despite not having a working **parser** I am very pleased with my progress in this area.

14.4 Objective 4

4. A user interface to render the puzzles [**Stretch Goal**]
   Requirements:
   - A **user interface** for visualizing solutions to problems
   - A **specification** for how to render different types of puzzles

Having focused most of my efforts on objectives *(1.)* to *(3.)*, I made no progress

towards a front end that was not terminal-based. However, I think this is justified considering it was a **stretch goal** and relies on the completion of objectives *(1.)* to *(3.)*. This is definitely an area of the project where a future contributor could make a significant impact.

15 Conclusion

This project can be considered a huge success and is a great tool for anyone wanting to solve grid puzzles. I have achieved all of my core objectives and made progress on some of my stretch goals. In its current state, the language is already expressive enough to be useful and can currently solve 7 of the 20 puzzles originally targeted.

I have outlined a clear direction for future contributions to the project, and shown that it is extensible should other developers wish to contribute. It would be very rewarding to see this project developed further.

A Sudoku in Different Languages

A.1 Sudoku in ECLiPse

```
sudoku(Board) :-
    dim(Board, [N,N]),
    Board :: 1..N,
    ( for(I,1,N), param(Board) do
        alldifferent(Board[I,*]),
        alldifferent(Board[*,I])
    ),
    NN is integer(sqrt(N)),
    ( multifor([I,J],1,N,NN), param(Board,NN) do
        alldifferent(concat(Board[I..I+NN-1, J..J+NN-1]))
    ).
```

A.2 Sudoku in MiniZinc

```
array[1..N,1..N] of 0..N: start; %% initial board 0 = empty
array[1..N,1..N] of var PuzzleRange: puzzle;

% fill initial board
constraint forall(i,j in PuzzleRange)(
    if start[i,j] > 0 then
        puzzle[i,j] = start[i,j] else true endif );

% All different in rows
constraint forall (i in PuzzleRange) (
    alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% All different in columns.
constraint forall (j in PuzzleRange) (
    alldifferent( [ puzzle[i,j] | i in PuzzleRange ]) );

% All different in sub-squares:
constraint forall
    (a, o in SubSquareRange)
    (alldifferent(
        [ puzzle[(a-1) *S + a1, (o-1)*S + o1]
        | a1, o1 in SubSquareRange ] ) );
```

B Connectivity

I expressed the **Path Connected Relation** described in section 9.1 using `sbv`'s notation for expressing quantifiers. This notation gives a self-descriptive way for expressing universal quantifiers and existentials. The following example, taken from [46] shows how you might express a formula using this notation:

```
prove $ \(Forall x) (Exists y) -> y .> (x :: SInteger)
```

Using this meant that I could express the relation in a way that was close to how I defined it:

```
x <~> y =
  (x <=> y)
  .&& ( x `neighbours` y
        .|| quantifiedBool  (\(Exists z) ->
             (x `neighbours` z) .&& (z <~> y)
        )
      )
```

Where:

- `neighbours` Checks if two cells are neighbours
- `<=>` Checks if two cells have the same value

Bibliography

[1] E. Dolstra and M. de Jonge, "Nix: A safe and Policy-Free system for software deployment," in *18th large installation system administration conference (LISA 04)*, Atlanta, GA: USENIX Association, Nov. 2004. Available: https://www.usenix.org/conference/lisa-04/nix-safe-and-policy-free-system-software-deployment

[2] "How nix works." Available: https://nixos.org/guides/how-nix-works.html. [Accessed: Jul. 19, 2023]

[3] "Introduction - alternative haskell infrastructure for NIXPKGs." Available: https://input-output-hk.github.io/haskell.nix/. [Accessed: Jul. 19, 2023]

[4] T. C. Project, "Cachix: Nix binary cache hosting." 2023. Available: https://cachix.org/

[5] Obsidian, "Obsidian: Sharpen your thinking." 2023. Available: https://obsidian.md/

[6] "Nikoli puzzle - nikoli." Available: https://www.nikoli.co.jp/en/puzzles/. [Accessed: Jul. 19, 2023]

[7] "Braingle grid puzzle puzzlepedia." Available: https://www.braingle.com/puzzlepedia/8-2/grid-puzzles.html. [Accessed: Jul. 19, 2023]

[8] "Braingle » puzzlepedia » nurikabe." www.braingle.com. Available: https://www.braingle.com/puzzlepedia/2-633/nurikabe.html. [Accessed: Jul. 20, 2023]

[9] "Braingle » puzzlepedia » sudoku." www.braingle.com. Available: https://www.braingle.com/puzzlepedia/8-1/sudoku.html. [Accessed: Jul. 20, 2023]

[10] "Puzzle nurikabe." Available: https://www.puzzle-nurikabe.com/. [Accessed: Jul. 19, 2023]

[11] "Braingle » puzzlepedia » bridges." www.braingle.com. Available: https://www.braingle.com/puzzlepedia/2-627/bridges.html. [Accessed: Jul. 20, 2023]

[12] "Hitori - nikoli." Nikoli, Sep. 2021. Available: https://www.nikoli.co.jp/en/puzzles/hitori/. [Accessed: Jul. 20, 2023]

[13] "Skyscrapers - online puzzle game." www.puzzle-skyscrapers.com. Available: https://www.puzzle-skyscrapers.com/. [Accessed: Jul. 20, 2023]

[14] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[15] S. UCL Centre for Research on Evolution and T. (CREST), "Introduction to SAT (constraint) solving." 2023. Available: http://crest.cs.ucl.ac.uk/readingGroup/satSolvingTutorial-Justyna.pdf

[16] M. Fredrikson, "Lecture notes on satisfiability modulo theories." Carnegie Mellon University, 2023. Available: https://www.cs.cmu.edu/~15414/lectures/18-smt.pdf. [Accessed: Jul. 20, 2023]

[17] M. Research, "Z3 - microsoft research." https://www.microsoft.com/en-us/research/project/z3-3/, 2023.

[18] H. Barbosa *et al.*, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and algorithms for the construction and analysis of systems*, D. Fisman and G. Rosu, Eds., Cham: Springer International Publishing, 2022, pp. 415–442.

[19] J. SCHIMPF and K. SHEN, "ECLiPSe – from LP to CLP," *Theory and Practice of Logic Programming*, vol. 12, no. 1–2, pp. 127–156, 2012, doi: 10.1017/S1471068411000469

[20] P. Wuille and T. Schrijvers, "Monadic constraint programming with gecode," 2009. Available: https://api.semanticscholar.org/CorpusID:15902882

[21] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a standard CP modelling language," in *Principles and practice of constraint programming – CP 2007*, C. Bessière, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543.

[22] T. H. L. S. Project, "Haskell language server." 2023. Available: https://haskell-language-server.readthedocs.io/en/latest/

[23] R. Evans, S. Frohlich, and M. Wang, "CircuitFlow: A domain specific language for dataflow programming (with appendices)," *CoRR*, vol. abs/2111.12420, 2021, Available: https://arxiv.org/abs/2111.12420

[24] R. P. Pieters and T. Schrijvers, "PaSe: An extensible and inspectable DSL for micro-animations," *CoRR*, vol. abs/2002.02171, 2020, Available: https://arxiv.org/abs/2002.02171

[25] J. Carette, B. MacLachlan, and W. S. Smith, "GOOL: A generic object-oriented language (extended version)," *CoRR*, vol. abs/1911.11824, 2019, Available: http://arxiv.org/abs/1911.11824

[26] R. Cheplyaka, *Tasty: Modern and extensible testing framework.* 2023. Available: https://hackage.haskell.org/package/tasty

[27] S. Hengel, *Hspec: A testing framework for haskell.* 2023. Available: https://hspec.github.io/

[28] D. Herington, *HUnit: A unit testing framework for haskell.* 2023. Available: https://hackage.haskell.org/package/hunit

[29] P. W. Tom Schrijvers, *MonadicCP: Constraint programming.* 2017. Available: https://people.cs.kuleuven.be/~tom.schrijvers/MCP/

[30] C. Penner, *Mad-props: Monadic DSL for building constraint solvers using basic propagators.* 2019. Available: https://github.com/ChrisPenner/mad-props#readme

[31] L. Erkok, *Sbv: SMT based verification: Symbolic haskell theorem prover using SMT solving.* 2023. Available: https://github.com/ChrisPenner/mad-props#readme

[32] A. L. Daan Leijen Paolo Martini, *Parsec: Monadic parser combinators*. 2023. Available: http://www.cs.uu.nl/~daan/parsec.html

[33] B. O'Sullivan, *Attoparsec: Fast combinator parsing for bytestrings and text*. 2023. Available: https://github.com/haskell/attoparsec

[34] M. Karpov, *Megaparsec: Monadic parser combinators*. 2023. Available: https://github.com/mrkkrp/megaparsec

[35] M. Karpov, "Megaparsec tutorial." 2019. Available: https://markkarpov.com/tutorial/megaparsec.html

[36] Nc. Team, "Ncurses homepage." 2023. Available: https://invisible-island.net/ncurses/

[37] J. Daugherty, *Brick: A declarative terminal user interface library*. 2023. Available: https://github.com/jtdaugherty/brick/

[38] E. W. Weisstein, "Polyomino." https://mathworld.wolfram.com/Polyomino.html, 2023.

[39] S. Team, *Sbv documentation: data.SBV.internals.SatisfiableM*. 2023. Available: https://hackage.haskell.org/package/sbv-10.2/docs/Data-SBV-Internals.html#t:SatisfiableM

[40] D. Jungnickel, "Connectivity and depth first search," in *Graphs, networks and algorithms*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 251–273. doi: 10.1007/978-3-642-32278-5_8. Available: https://doi.org/10.1007/978-3-642-32278-5_8

[41] W. A. Sutherland, "Introduction to metric and topological spaces," in Oxford mathematics. Oxford University Press, 2009, p. 120.

[42] W. A. Sutherland, "Introduction to metric and topological spaces," in Oxford mathematics. Oxford University Press, 2009, p. 54.

[43] GitHub, "GitHub actions documentation." 2023. Available: https://docs.github.com/en/actions

[44] M. Snoyman, "Yaml: Support for parsing and rendering YAML documents." 2023. Available: https://github.com/snoyberg/yaml#readme

[45] H. V. Riedel, "HsYAML: Pure haskell YAML 1.2 processor." 2023. Available: https://github.com/haskell-hvr/HsYAML

[46] S. Team, *Sbv documentation: data.SBV*. 2023. Available: https://hackage.haskell.org/package/sbv-10.2/docs/Data-SBV.html#g:44