Final Year Project Report

Formalising Alternative Models of Computation in Isabelle
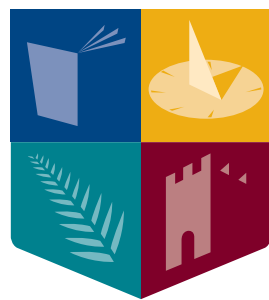
Dara MacConville | 17377693

A thesis submitted in partial fulfilment of the requirements for the
B.Sc. Computational Thinking

5 Credits

Advisor: Dr. Philippe Moser

Department of Computer Science
Maynooth University, Ireland

April 13, 2019

Abstract

The goal of this project was to formalise and implement alternative models of computation inside the proof assistant Isabelle, and then to make use of this to assist in providing definitions on and proving various results about these models. In particular it focuses on Cellular Automata, in both one and two-dimensional variants, and with differing topologies.

CONTENTS

Lists of Floats

List of Tables

List of Figures

List of Listings

Introduction

Cellular Automata (CA) are a very simple model of computation. There are many variations and extensions of them, and some like Conway's Game of Life and Rule 110 are known to be Turing Complete. Isabelle is a proof assistant that can be used for anything from mathematical proofs to formal verification of software properties.

1.1   Topic addressed in this project

This project looks at formalising models of Cellular Automaton in Isabelle to help deal with the complexity that comes with trying to mathematically prove results about them. In addition to providing a formalisation of six different variants of CA, this project provides definitions of important properties they may have, and proves some lemmas and theorems necessary to work with them.

1.2   Motivation

It can be very difficult to work with high level concepts while still being rigorous, and theoretical Computer Science is a very abstract and mathematical discipline that requires exactly that. One of the main concepts used in theoretical CS is the Turing Machine. While it provides an excellent way of approaching computation that allows for conceptually easy high level proofs, being very strict and formal in these can prove difficult.

However the Turing Machine is not the only available model in CS. One of the goals of this project was to look at *alternative* concepts that achieve the same thing, to help better appreciate the benefits and drawbacks. Cellular Automata were chosen as they strike a good balance between simplicity and complexity, as they have a very simple idea, but come with additional geometric aspects to consider.

The motivation to use a proof assistant rather than traditional pen and paper proofs comes from a desire to use the exactness of computers to cope with issues that stem from highly layered and detailed concepts.

1.3   Approach

Isabelle was chosen as the language and framework to implement these models in. This is due to it being very high level, and having advanced tools available to ease the creation process. These include the multitool command **sledgehammer** which invokes several Automatic Theorem Provers and Satisfiability-Modulo-Theories solvers, and **nitpick** which provides counter-examples to statements. These take the burden of large manual proofs off the programmer, while still encouraging simple and understandable definitions, as these are more easy to automatically prove results on.

Isabelle also has a powerful and expressive type system, and the decision was taken to directly make use of this in the CA definitions created. This involved making each kind of CA explicitly into a type, to make use of automatic type checking and other benefits. This contrasts with a more mechanical approach taken in a paper [1] that also works on implementing Turing Machines and other computability concepts in Isabelle. One reason for this difference in approach is their work involves the need to convert between different models, for instance Abacus Machine to Turing Machine, whereas in this project no conversions are done. Additionally the design of such "machines" lend themselves to being built in such a way. However it was not possible to entirely avoid that way of thinking with CA either, in particular in the two dimensional case.

1.4 METRICS

It can be difficult to evaluate exactly how well a definition has been translated from the informal to the formal. It could be possible to have a very precise and elegant program that doesn't actually capture the intuitive content of the informal idea. As such the definitions written here were evaluated in two main ways:

1. Understanding: Does reading the program lead to the same understanding as reading the original definitions? If so then it is capturing the same spirit as the original, even if the means of execution may differ slightly or significantly.

2. Functionality: As CA are systems that can be run, it is necessary to run them on example programs and see the correct results are achieved. Speed is not a trait that is sought out but accuracy is of course essential.

Also taken into account is the ability to state properties of these systems, and the ease or difficulty of proving results about them.

Technical Background

2.1 Topic Material

2.1.1 Formalisations in Computability Theory

As mentioned before, this is certainly not the first formalisation of computability theory concepts inside of proof assistants. Other works however have focused on more traditional models or with certain specific proofs in mind. These include λ-calculus in Coq [2], partial recursive functions in Lean [3], and the aforementioned Turing and Abacus machines in Isabelle [1]. Other than choices of model and language, the key difference between those works and this project is the end goal. The three mentioned above all set about to formalise computability theory from the perspective of their chosen model or models. This project works with just Cellular Automata and what results can be specifically shown about them.

Additionally the approach in [1] often involves specifying certain exact indices in lists and having to manipulate these numbers. This obviously has its technical benefits and is necessary when working with TMs, but the desire in this work was to take a "wholemeal" functional programming approach to the construction where possible.

The existing research that comes closest to both the goals and execution of this project, is a formalisation of CA in Coq for the purposes of verifying a result about the firing squad problem [4]. Their work bases the CA over the naturals $\mathbb{N}$, and includes an explicit time parameter. As well as that, their transition function is based more around individual cells, and the properties they define are all based around eventually deriving the proof. This project certainly has similarities, but trades the depth of [4] for breadth, and makes use of definitions that try to be more compact. For instance the time parameter is more external to the cellular automata, and the varieties of different CA are defined with different mechanisms.

2.1.2 Cellular Automata

The concept of CA has been around since the 1940s and were popularised in the 70s with Conway's Game of Life. However a lot of the names and terminology associated with them comes from Stephen Wolfram [5]. His work is not entirely formal, but others have provided their own formalisations of definitions he put forward [6]. This was especially useful as an aid in translating the some of the classifications of CA into Isabelle, even if their formalisation differed in many other ways.

2.2 Technical Material

2.2.1 Cellular Automata

A Cellular Automaton in general consists of a rectangular grid of cells sitting in some finite dimension, with each one of these cells in one of a finite number of different states. Each cell has a neighbourhood and a rule that determines how a cell changes with each time step, based off its own state and the states of the cells in its neighbourhood.

All the CA this project is concerned with have cells in one of only two binary states: `One` or `Zero`. On diagrams `One` is indicated by a filled in black square, and `Zero` by a white square. In one dimension these are known as *elementary cellular automata*.

This project deals with both one and two dimensional cellular automata. In the one dimensional case the overall state of the whole automaton can be thought of as a single line of cells, and in the two dimensional case as cells tiling the plane. Extensions to higher dimensions are of course possible but not dealt with here.

The neighbourhood used for one dimensional CA here is the simplest possible, consisting only of a trio of cells including the cell itself in the middle, and the cell to its immediate left and right as seen in Figure 2.1.

Figure 2.1: Neighbourhood of the blue cell includes itself and the cells in red

In two dimensions there are more possible neighbourhoods that still seem natural. The one used here is known as the *Moore neighbourhood*, made up of the nine cells that form a square including the cell itself as the centre. Alternatives include the *von Neumann neighbourhood*, which only takes the squares in the cardinal directions from the centre.
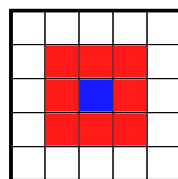
Figure 2.2: Moore neighbourhood

When showing the development of a one dimensional CA over time, it is best represented by multiple rows of cells, with each new row representing the state at the next time step, and with time increasing downwards along the y-axis. It's important to remember that the whole structure depicted never exists at any point in time, only the one dimensional slices across.
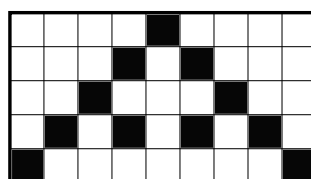
Figure 2.3: Rule 26 traces a Sierpiński triangle over time

A two dimensional CA is of course visually shown as a cell diagram in the plane, with changes over time indicated via multiple diagrams adjacent to each other and time flowing forward from left to right.
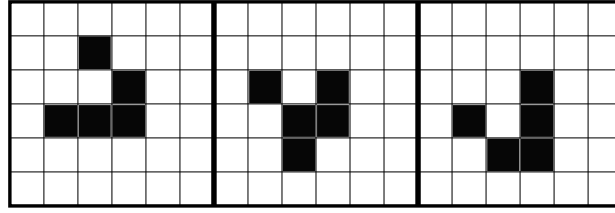

Figure 2.4: 3 steps of a "glider" in Game of Life

Rules can be specified by a mathematical formulation based on the cells in a neighbourhood as they are in the Game of Life, or just explicitly defined by listing all inputs and outputs. For elementary CA the rule is derivable from its Wolfram code which is also used as the name of the rule e.g. Rule 110. However in this project explicit representations will be given if necessary. For instance part of the definition of Rule 110 is given in Table 2.1

Neighbourhood	■■■	■■□	■□■
Output	□	■	■

Table 2.1: Rule 110

2.2.2 ISABELLE

Isabelle as a proof assistant allows us to to define algebraic datatypes and pure total recursive functions, using pattern matching, guards, and conditionals in a syntax very similar to Haskell's. Function types are represented in the usual currying friendly way for multiple arguments. For instance see listings 2.1 below.

```
Listing 2.1: Functions demonstrating multiple arguments and pattern matching

1  fun get_cell :: "state ⇒ int ⇒ int ⇒ cell" where
2  "get_cell s x y = s!(nat x)!(nat y)"
3
4  fun apply_nb :: "(cell ⇒ cell) ⇒ neighbourhood ⇒
   ↪  neighbourhood" where
5  "apply_nb f (Nb a b c) = Nb (f a) (f b) (f c)"
```

On top of these we can state and prove theorems using all the usual first order logic connectives, quantifiers and operators, along with some Isabelle specifics. There are multiple different ways of writing proofs in Isabelle, with some being written in the more human-readable Isar language and others simply being a chain of commands to apply. Both kinds will be made use of in the project.

Strictly speaking everything in quotation marks is part of HOL, the logic that Isabelle runs on by default, and the language most of the actual programming happens in. However for the

5

Listing 2.2: An example statement of theorem named "t1" along with proof

```
1  theorem t1 :"n>0 ⟹ ca yields State (run ca n)"
2    apply(simp add: yields_def)
3    apply(rule exI)
4    apply(rule conjI)
5    apply(auto)
6    done
```

sake of simplicity throughout this report the whole language and infrastructure shall simply be referred to as "Isabelle". Each file in Isabelle is referred to as a **theory**, and is a selection of definitions and proofs. These theories may draw in and reference other predefined or user made theory files as dependencies, in the same way as any programming language uses libraries.

The Problem

3.1 Problem Analysis

The goal of the project was to somehow implement a variety of types of CA in Isabelle in a workable fashion. The types of CA were one or two dimensional CA that were either finite or infinite, leading to essentially four main categories to develop. There were two main issues to address for each of these.

The first was how to implicitly or explicitly represent the state of a CA. This question was especially key in the infinite case, as it requires thinking beyond traditional finite data structures. The other topic of interest was how to develop the state of the CA as time progresses. Again this is much more a difficult and important issue in the infinite case, as you cannot simply apply a function over infinitely many values. This rules out the obvious choices such as using `map` and other functions in a similar vein.

The finite automata also had certain issues that do not occur in the infinite cases. As they are finite they have boundaries, and for cells on these boundaries the neighbourhood necessary to determine its evolution does not exist in the standard way. This requires a decision to be made to either adjust the neighbourhood of these cells, or the rule that applies to them.

The Solution

The purpose of this chapter is to clearly identify, discuss, and justify the decisions you make Depending on your type of project, you may not need to include all of these:
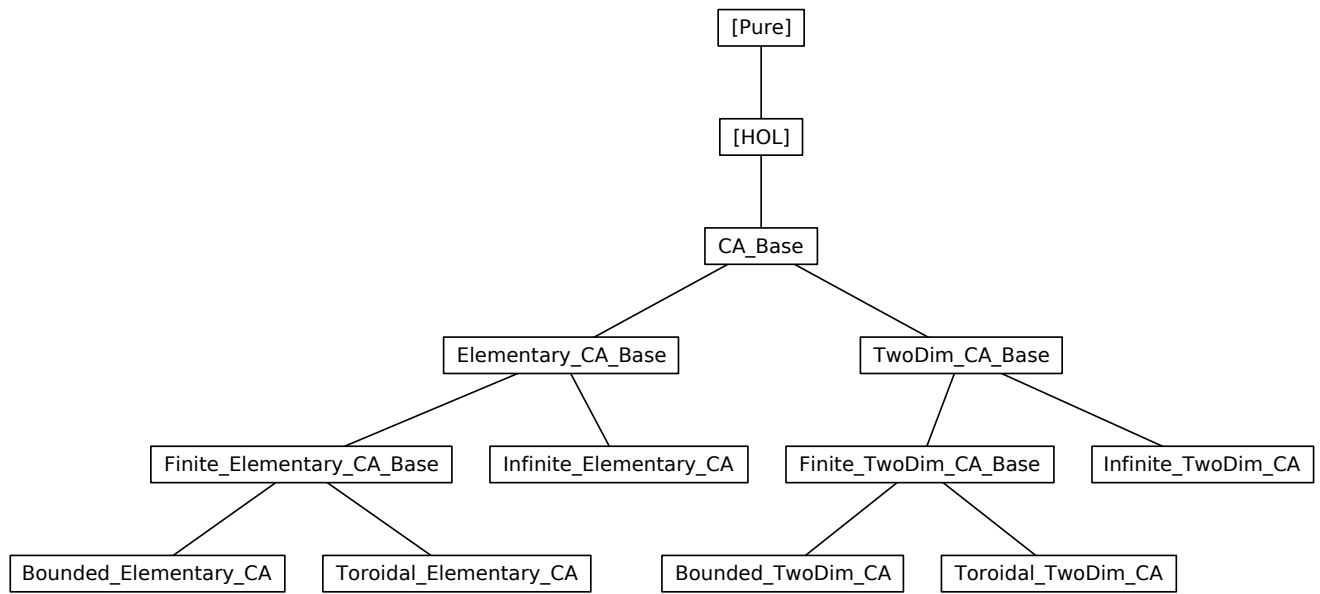
4.1 Architectural Level

Figure 4.1: Dependency graph of project Theories

Figure 4.1 represents the overall architecture of dependencies of the various theory files in the project, with files further down the tree depending on those above. [Pure] and [HOL] contain the base definitions necessary to do work in Isabelle, all the rest are new theories created for the project.

From the simplest core definitions given in CA_Base, the project essentially splits into two due to the differences in implementing one dimensional versus two dimensional CA. However the internal structure of these two subtrees are exactly the same apart from the distinction in dimension. They both contain a base file for the two kinds of finite CA, and another file dealing with the infinite case.

Each of the six root nodes contains the definition of one of the six distinct kinds of CA realised in the project. They very roughly increase in power and complexity from left to right.

4.2 The Cellular Automata Type

The type of Cellular Automata is redefined for each of the four main branches in the project. Those are elementary finite, two-dimensional finite, elementary infinite, and two-

dimensional infinite. This is due to the geometric and functional differences that have to exist in the `state` type parameter underlying each broad class of CA. Despite this, the actual signature of the type remains unchanged for each. This is the allow the definitions and proofs that sit on top of them to make the same assumptions about type structure and composition. As shown below in listings 4.2 the definition is very compact and simple. All CA also share the binary cell type given in listings 4.1.

As you would hope, a Cellular Automata consists only of the current global state it is in, and the rule necessary to transition it to the next state. This however does hide quite a lot of complexity, and in fact the actual functioning of a CA is created from more than these two parameters.

As an example of this, [6] actually defines CA as a 4-tuple (k, S, N, f), where k is the dimension, S the set of states, N the neighbourhood, and f the local rule. This is certainly more transparent then the definition given here. In this project's version the information about neighbourhoods is given implicitly from a neighbourhood function that sits in the local context where we want to actually run the CA. The reason for the approach taken here is for elegance and ease of use in further results on the CA type. Adding more information to the type directly can mean more work in pattern matching, and constructing and deconstructing the type every time it is used.

Listing 4.1: Cell definition

```
1 datatype cell = Zero | One
```

Listing 4.2: CA type signature

```
1 datatype CA = CA (State : state) (Rule : rule)
```

The `State` and `Rule` that are capitalised are just syntactic sugar for defining accessor functions for those arguments to the type. So the `State` function takes a CA and returns its state, and similar for `Rule`.

It is also worth noting that the CA here is not directly defined as a tuple. Using a unique datatype carries more semantic weight then giving a **type_synonym** to a tuple. The same concept does not exist in general mathematics so purely mathematical approaches to formalisms usually just involve a tuple.

4.2.1 STATE

As mentioned in the above paragraphs, the `state` type is designed differently across the four general distinctions of CA. As such the more detailed description of each is left to its own respective section. For a high level overview it suffices to say that the finite CA states were simply a one or two dimensional list of states, given geometry through either pattern matching or indices. The infinite CA states were modeled completely differently, as a function from the integers to cells.

4.2.2 NEIGHBOURHOODS & RULES

Unlike `state`, `rule` is defined exactly the same way for all CA, as given in listings 4.3. The only differing factor being the type of neighbourhood it acts on.

> Listing 4.3: Type definition of `rule`
>
> ```
> type_synonym rule = "neighbourhood ⇒ cell"
> ```

Neighbourhoods differ between one and two dimensions as expected but not in a conceptually deep way. As shown in listings 4.4 the only practical difference is adding more `cell` arguments.

> Listing 4.4: The two kinds of neighbourhood
>
> ```
> (* One dimensional *)
> datatype neighbourhood = Nb cell cell cell
>
> (* Two dimensional *)
> datatype neighbourhood = Nb
> (NorthWest:cell) (North:cell)  (NorthEast:cell)
> (West:cell)      (Centre:cell) (East:cell)
> (SouthWest:cell) (South:cell)  (SouthEast:cell)
> ```

This approach of generating neighbourhoods as an entity distinct from mere cells allowed a higher level approach to dealing with CA. It meant that in the finite elementary CA, the question of updating cells is solved by simply mapping a `rule` over the neighbourhoods, see listings 4.5. By abstracting out a neighbourhood function it also also for minor tweaks to that to entirely change the topology of a finite CA, as explained in Section 4.3.

> Listing 4.5: Updating via neighbourhoods
>
> ```
> (* Finite one dimensional *)
> fun update_CA :: "CA ⇒ CA" where
> "update_CA (CA s r) = CA (map r (nbhds s)) r"
>
> (* Finite two dimensional *)
> fun update_CA :: "CA ⇒ CA" where
> "update_CA (CA s r) = CA (map (λ xs. map r xs) (nbhds s)) r"
> ```

In the two dimensional finite case, as state is just a list of lists of cells, it requires a mapping of maps over those but the general principle is similar.

A CA can be run for multiple steps through repeated application of `update_CA` through another function designed for this.

4.3   FINITE CELLULAR AUTOMATA

There ended up being two separate paths taken in dealing with the cells on the boundary of the finite CA. The first, which for the purposes of the project is named *bounded*, is the simplest conceptually, but not necessarily in implementation. The other is termed *toroidal*, because using it gives the CA the geometry of a torus or band. Despite seeming more complicated, implementation is very natural compared to the bounded method.

Bounded CA deal with the problem of determining the neighbourhood of a cell on the edge simply via "padding". What this practically means is that when the neighbourhood function goes to generate the neighbourhood of a cell it knows to be on the edge, it pretends that there are actually Zero cells filling in the blanks and returns a neighbourhood accordingly.

The toroidal method is so called as it essentially "glues" the opposite ends of the state data structure together, thus creating a torus in two dimensions, or a band in one. From a purely mathematical topology perspective this works via altering the neighbourhoods of cells on the extremities, to include those cells that should be next to it if it were in a toroidal shape.

4.3.1   ONE DIMENSIONAL

In the one dimensional case this is very easy to implement. All that has to be done is stick the additional cells onto the state list, and call another function inner_nbhds that does the obvious action of stepping through the internal items in a list, returning the neighbourhoods for each.

```
Listing 4.6: Finite one dimensional neighbourhood generation

1  type_synonym state = "cell list"
2
3  (* Bounded elementary CA *)
4  fun nbhds :: "state ⇒ neighbourhood list" where
5  "nbhds  xs = inner_nbhds (Zero # xs @ [Zero])"
6
7  (* Toroidal elementary CA *)
8  fun nbhds :: "state ⇒ neighbourhood list" where
9  "nbhds xs = inner_nbhds ((last xs) # xs @ [hd xs])"
```

For the bounded automata these additional cells are defined to always be constant Zero cells. It would be equally valid and simple to take these both to be One or a mix of each, but zeroing the boundaries was the most natural choice.

As shown in listings 4.6, it only takes a very minor adjustment to the nbhd function to radically overhaul the topology of the shape produced. By changing the cells padded onto the state list to be the last item and head of the original list, the cells of each extreme end of the list are associated with each other.

Theoretically altering the neighbourhood function is all that is needed to turn a one dimensional list into any finite shape or geometry required. However this would require much greater modifications to the base structure of the function and so was not the approach taken in transitioning finite CA to two dimensions.

4.3.2 TWO DIMENSIONAL

If a one dimensional CA state uses a single list, the obvious transition to scale to two dimensions is to use a two dimensional list.

Listing 4.7: Finite two dimensional neighbourhood generation

```
1  type_synonym state = "cell list list"
2
3  (* function for all 2D neighbourhoods *)
4  fun nbhds :: "state ⇒ neighbourhood list list" where
5      "nbhds s = (let h = (int_height s)-1
6      in (let w = (int_width s)-1 in
7      [[get_nbhd s x y. y ← [0..h]]. x ← [0..w]]))"
8
9  (* Toroidal get_nbhd *)
10 fun get_nbhd :: "state ⇒ int ⇒ int ⇒ neighbourhood" where
11     "get_nbhd s x y =
12     (let w = int_width s in
13     (let h = int_height s in
14     list_to_nb [get_cell s ((x+i) mod w) ((y+j) mod h).
15         j ← rev [-1..1], i ← [-1..1]]))"
16
17 (* Bounded get_cell *)
18 fun get_cell :: "state ⇒ int ⇒ int ⇒ cell" where
19     "get_cell s x y = (if out_of_bounds s x y
20                         then Zero
21                         else  s!(nat x)!(nat y))"
```

It is very clear from listings 4.7 that the move to two dimensions involved the use of a lot more indices in lists. This is due to the fact that pattern matching is not as simple when you have lists of lists, so indexing the lists was deemed easier to work with.

The main nbhds function achieves this by a nested list comprehension over the length and width of the CA state, and fills in those indices with get_nbhd. The function get_nbhd itself delegates to get_cell.

In this case both the bounded and toroidal CA have the same nbhds function, the different shapes are instead produced through changes to the intermediary get_nbhd, and get_cell. Note how the get_nbhd function does the work to create a torus by wrapping indices around with mod, while in the bounded case get_cell is responsible for filling in the Zeros.

For the toroidal CA, `get_cell` just does not default to returning `Zero`, and for bounded CA, `get_nbhd` does not use any modular wrapping.

To understand the indexing used in the two dimensional state, the inner lists have to be though of as column vectors moving vertically.

4.4 Infinite Cellular Automata

Infinite CA are actually simpler because they have no boundaries so there was no need for the all the edge case handling that characterised finite CA. However they require discarding conventional data structures like lists, for an approach much more grounded in mathematics and functional programming.

4.4.1 Neighbourhoods

The neighbourhood does not play quite as big a part in the infinite, especially since there were no variations made of the topology. However it was still used in the interest of consistency in level of abstraction and definition, and to allow for potential future uses of it. This allowed the type signature of `rule` to remain unchanged too.

4.4.2 State

In the end infinite state, and extending it, turned out to be much simpler than the finite case. One dimensional state simply needs to pair every integer with a cell, which naturally works as a function over the integers. Note this function is total so the entire state is represented, there is no need for a partial function or something of that kind.

Two dimensional state is extended easily from the base case, although strictly speaking it is not modelled as a function from the Cartesian product of the integers, but just as a function with two integer arguments. These two are of course isomorphic, but multiple arguments is more easily extensible, and allows for partial application and currying.

```
Listing 4.8: Infinite state represented as functions

1 (* One dimensional *)
2 type_synonym state = "int ⇒ cell"
3
4 (* Two dimensional *)
5 type_synonym state = "int ⇒ int ⇒ cell"
```

Unlike the finite case, the real work for infinite CA is done in updating the state. Using `map` was no longer an option as there is no finite data structure to map over. Instead a recursive tactic was employed. The whole state never explicitly exists at once, it cannot as it is infinite, but the values of any amount of cells at any stage can always be known. The key insight that made this work, was to update the `state` function at each stage, with the new action on an integer recursively based on what the cell values around it were in the

13

previous stage. So when the function is called after a certain amount of updates, it calculates the current values needed by recursing all the way back to a base state that was supplied. This works very nicely in one dimension, but the two dimensional version can no longer be considered elegant.

Listing 4.9: Infinite transition functions

```
(* One dimensional *)
fun update_state :: "CA ⇒ state" where
"update_state (CA s r) n = r (Nb (s (n-1)) (s n) (s (n+1)))"

(* Two dimensional *)
fun update_state :: "CA ⇒ state" where
"update_state (CA s r) x y =
    r (Nb (s (x-1) (y+1)) (s x (y+1)) (s (x+1) (y+1))
          (s (x-1)  y)    (s x  y)    (s (x+1)  y)
          (s (x-1) (y-1)) (s x (y-1)) (s (x+1) (y-1)))"
```

Overall due to the definitions used, the differences in implementation between one and two dimensional infinite automata, were much smaller than the differences in the finite case.

4.5 PROPERTIES

Due to the way CA and their constituent components were built with the same type signatures, most properties built on them could hold with mostly trivial adjustments for the CA type. Additionally a lot of common properties about CA, when phrased in the language to describe CA develop here, are actually properties about the rule making up a CA. These abstractions enabled some properties to apply with no adjustments to all CA types.

In total about sixteen distinct properties of CA were defined, some totally generic, others, like those in listings 4.10, are specific to the structural constraints of one dimension. This number also does not include utility functions necessary to get the CA to work, along with ancillary lemmas proved about these for basic simplification properties. Additionally for each of these properties, they were related to either another property or a concrete CA via theorems and lemmas.

Listing 4.10: One dimensional structure properties

```
fun mirror :: "rule ⇒ rule" where
"mirror r (Nb a b c) = r (Nb c b a)"

definition amphichiral :: "rule ⇒ bool" where
"amphichiral r ≡ (ALL c. r c = (mirror r) c)"
```

Note the ALL is simply an ASCII stand in for the ∀ quantifier.

As well as properties about CA types in general, two specific rules of CA of interest were defined with some properties proven specifically about them. These were Rule 110 for one dimensional CA and Conway's Game of Life for two dimensions.

4.5.1 RULE 110

Rule 110 is an interesting example as it has been proven Turing Complete. Some negative results were proved about it, for instance it can be shown that it does not fit the definition of amphichiral from listings 4.10.

Listing 4.11: Negative result on Rule 110

```
1    lemma "¬amphichiral r110"
2 by (metis amphichiral_def cell.distinct(1) mirror.simps
   ↪   r110.simps(4) r110.simps(7))
```

This lemma does not have as nice as proof as that in listings 4.12. This is because it is a computer generated proof, found via use of **sledgehammer**. A lot of proofs in this project were found via computer search, a hallmark of this is a proof being very short but not very human readable.

4.5.2 GAME OF LIFE

The Game of Life is a two dimensional CA popularised by John Conway and Martin Gardener. Like Rule 110 it is also Turing Complete. One key aspect of Life, is that is it *outer totalistic*, meaning a cell changes depending only on its own state and the total number of One cells in its neighbourhood. Translated into Isabelle we get the following listings 4.12, which contains a one line proof of this simply via "unfurling" the definitions.

Listing 4.12: Outer totalistic Life

```
1 definition outer_totalistic :: "rule ⇒ bool" where
2 "outer_totalistic r ≡ (ALL nb1 nb2. (Centre nb1 = Centre nb2)
3 ⟶ (outer_sum nb1 = outer_sum nb2) ⟶ (r nb1) = (r nb2))"
4
5 theorem "outer_totalistic life"
6 by apply(simp add: outer_totalistic_def life_def)
```

EVALUATION

Conclusion

BIBLIOGRAPHY

[1] Jian Xu, Xingyuan Zhang, and Christian Urban. "Mechanising Turing Machines and Computability Theory in Isabelle/HOL". In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 147–162. ISBN: 978-3-642-39634-2.

[2] Yannick Forster and Gert Smolka. "Weak call-by-value lambda calculus as a model of computation in Coq". In: *International Conference on Interactive Theorem Proving*. Springer. 2017, pp. 189–206.

[3] Mario Carneiro. "Formalizing computability theory via partial recursive functions". In: *arXiv preprint arXiv:1810.08380* (2018).

[4] Jean Duprat. "Proof of correctness of the Mazoyer's solution of the firing squad problem in Coq". In: (2002).

[5] Stephen Wolfram. *A New Kind of Science*. English. Champaign, IL: Wolfram Media, 2002. ISBN: 9781579550080;1579550088;

[6] Karel Culik and Sheng Yu. "Undecidability of CA Classification Schemes". In: *Complex Systems* 2 (1988).

[7] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Jan. 2002. DOI: 10.1007/3-540-45949-9.