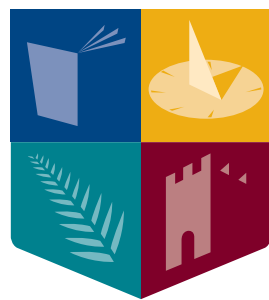

Final Year Project Report

Formalising Alternative Models of Computation in Isabelle



**Maynooth
University**

National University
of Ireland Maynooth

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

| | | |
|----------|--------------------------------------|----------|
| 1 | The Solution | 1 |
| 1.1 | Architectural Level | 1 |
| 1.2 | The Cellular Automata Type | 1 |
| 1.3 | Finite Cellular Automata | 4 |
| 1.4 | Infinite Cellular Automata | 6 |
| | Bibliography | 7 |

LISTS OF FLOATS

LIST OF TABLES

LIST OF FIGURES

| | | |
|-----|--|---|
| 1.1 | Dependency graph of project Theories | 1 |
|-----|--|---|

LIST OF LISTINGS

| | | |
|-----|---|---|
| 1.1 | Cell definition | 2 |
| 1.2 | CA type signature | 2 |
| 1.3 | Type definition of rule | 3 |
| 1.4 | The two kinds of neighbourhood | 3 |
| 1.5 | Updating via neighbourhoods | 3 |
| 1.6 | Finite one dimensional neighbourhood generation | 4 |
| 1.7 | Finite two dimensional neighbourhood generation | 5 |

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:

1.1 ARCHITECTURAL LEVEL

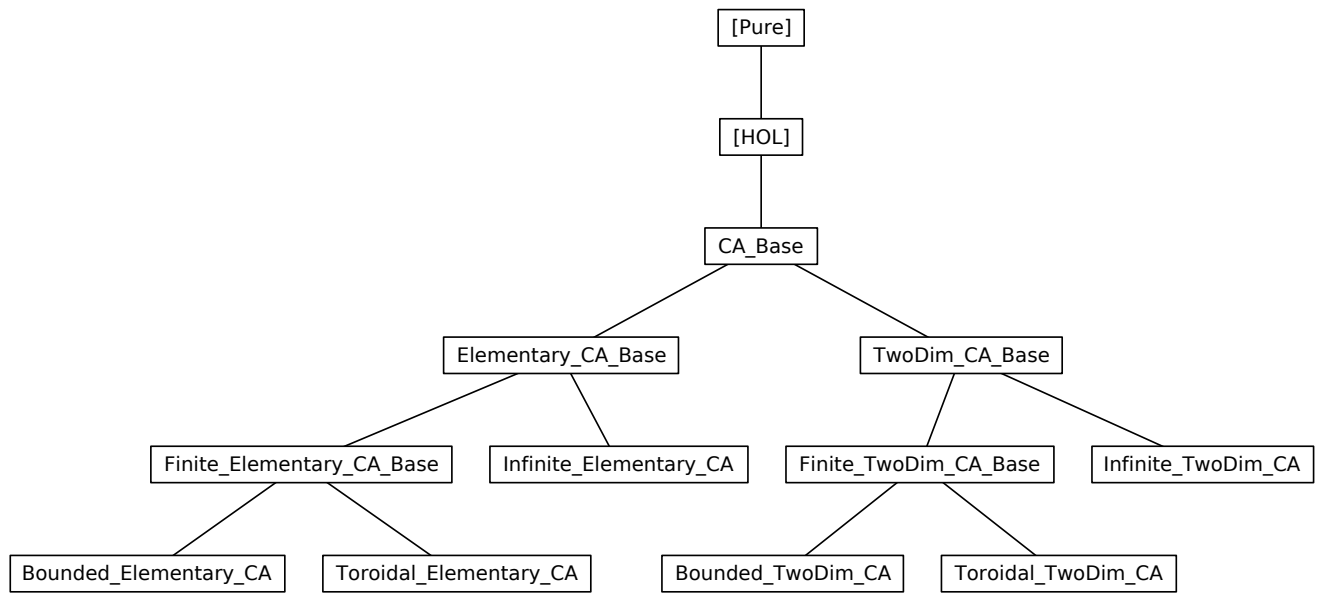


Figure 1.1: Dependency graph of project Theories

Figure 1.1 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.

1.2 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 to 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 1.2](#) the definition is very compact and simple. All CA also share the binary cell type given in [listings 1.1](#).

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 than 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 1.1: Cell definition

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

Listing 1.2: 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 than 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.

1.2.1 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.

1.2.2 NEIGHBOURHOODS & RULES

Unlike state, rule is defined exactly the same way for all CA, as given in [listings 1.3](#). The only differing factor being the type of neighbourhood it acts on.

Listing 1.3: Type definition of rule

```
1 type_synonym rule = "neighbourhood  $\Rightarrow$  cell"
```

Neighbourhoods differ between one and two dimensions as expected but not in a conceptually deep way. As shown in [listings 1.4](#) the only practical difference is adding more cell arguments.

Listing 1.4: The two kinds of neighbourhood

```
1 (* One dimensional *)
2 datatype neighbourhood = Nb cell cell cell
3
4 (* Two dimensional *)
5 datatype neighbourhood = Nb
6 (NorthWest:cell) (North:cell) (NorthEast:cell)
7 (West:cell) (Centre:cell) (East:cell)
8 (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 1.5](#). By abstracting out a neighbourhood function it also allows for minor tweaks to that to entirely change the topology of a finite CA, as explained in [Section 1.3](#).

Listing 1.5: Updating via neighbourhoods

```
1 (* Finite one dimensional *)
2 fun update_CA :: "CA  $\Rightarrow$  CA" where
3 "update_CA (CA s r) = CA (map r (nbhds s)) r"
4
5 (* Finite two dimensional *)
6 fun update_CA :: "CA  $\Rightarrow$  CA" where
7 "update_CA (CA s r) = CA (map ( $\lambda$  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.

1.3 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.

1.3.1 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 1.6: 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 1.6](#), 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.

1.3.2 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 1.7: Finite two dimensional neighbourhood generation

```

1 type_synonym state = "cell list list"
2
3 (* function for all 2D neighbourhoods *)
4 fun nbhds :: "state  $\Rightarrow$  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  $\leftarrow$  [0..h]]. x  $\leftarrow$  [0..w]]))"
8
9 (* Toroidal get_nbhd *)
10 fun get_nbhd :: "state  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  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  $\leftarrow$  rev [-1..1], i  $\leftarrow$  [-1..1]]))"
16
17 (* Bounded get_cell *)
18 fun get_cell :: "state  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  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 1.7](#) 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.

In this case both the bounded and toroidal CA have the same nbhds function, the different shapes are instead produced through changes to an 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.

1.4 INFINITE CELLULAR AUTOMATA

1.4.1 ONE DIMENSIONAL

1.4.2 TWO DIMENSIONAL

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](https://doi.org/10.1007/3-540-45949-9).