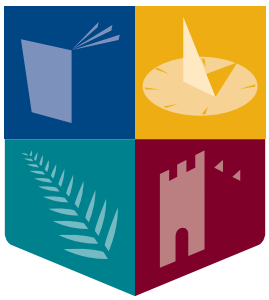
Final Year Project Report

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

Lists of Floats

List of Tables

List of Figures

List of Listings

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

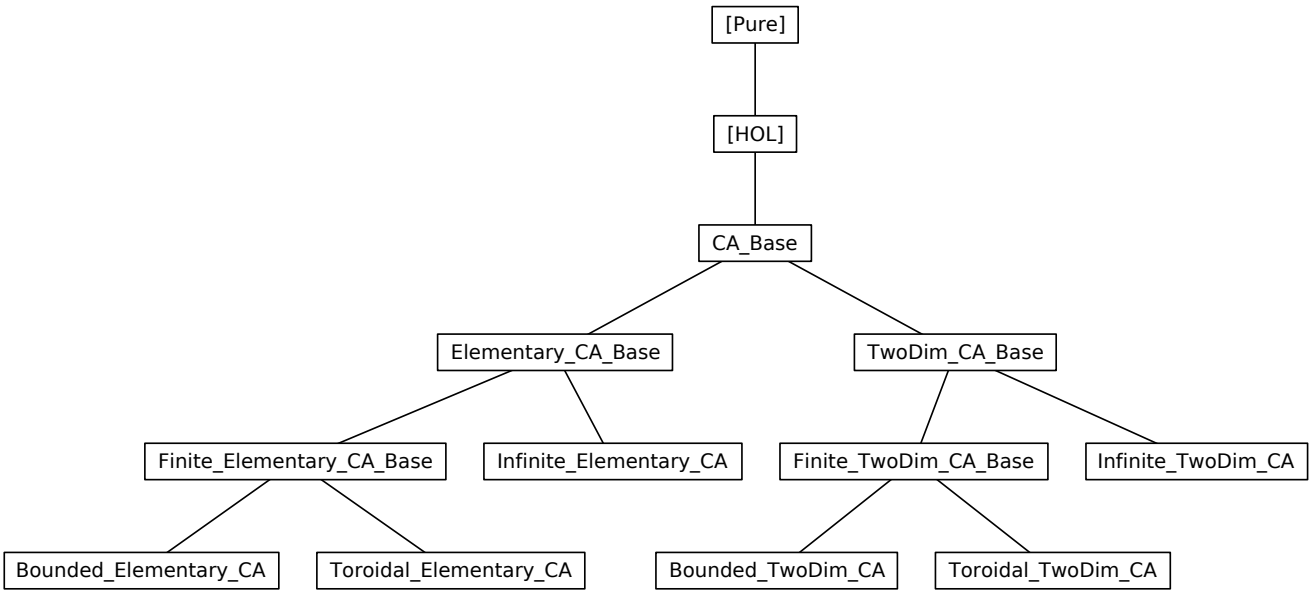


Figure 1.1: Dependency graph of project Theories

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 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 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, [1] 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 1.1: Cell definition

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

Listing 1.2: CA type signature

```
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.

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`

```
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 1.4 the only practical difference is adding more `cell` arguments.

Listing 1.4: 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 1.5. 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 1.3.

Listing 1.5: 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.

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

```
type_synonym state = "cell list"

(* Bounded elementary CA *)
fun nbhds :: "state ⇒ neighbourhood list" where
"nbhds  xs = inner_nbhds (Zero # xs @ [Zero])"

(* Toroidal elementary CA *)
fun nbhds :: "state ⇒ neighbourhood list" where
"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

```
type_synonym state = "cell list list"

(* function for all 2D neighbourhoods *)
fun nbhds :: "state ⇒ neighbourhood list list" where
    "nbhds s = (let h = (int_height s)-1
    in (let w = (int_width s)-1 in
    [[get_nbhd s x y. y ← [0..h]]. x ← [0..w]]))"

(* Toroidal get_nbhd *)
fun get_nbhd :: "state ⇒ int ⇒ int ⇒ neighbourhood" where
    "get_nbhd s x y =
    (let w = int_width s in
    (let h = int_height s in
    list_to_nb [get_cell s ((x+i) mod w) ((y+j) mod h).
        j ← rev [-1..1], i ← [-1..1]]))"

(* Bounded get_cell *)
fun get_cell :: "state ⇒ int ⇒ int ⇒ cell" where
    "get_cell s x y = (if out_of_bounds s x y
                       then Zero
                       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.

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.

1.4 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.

1.4.1 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.

1.4.2 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 1.8: Infinite state represented as functions

```
(* One dimensional *)
type_synonym state = "int ⇒ cell"

(* Two dimensional *)
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

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 1.9: 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.

1.5 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 fifteen distinct properties of CA were defined, some totally generic, others, like those in listings 1.10, 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.

Listing 1.10: 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 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.

1.5.1 Rule 110

Rule 110 is an interesting example as it has been proved to be Turing Complete

1.5.2 Game of Life

The Game of Life is a two dimensional CA popularised by John Conway and Martin Gardener. One key aspect of Life, is that is it *totalistic*, meaning a cell changes depending only on the total number of `One` cells in its neighbourhood.

Listing 1.11: Totalistic Life

```
definition totalistic :: "rule ⇒ bool" where
"totalistic r ≡ (ALL nb1 nb2. sum_nb nb1 = sum_nb nb2 ⟶ (r
 ↪  nb1) = (r nb2))"
```

Evaluation

Conclusion

Bibliography

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

[2] 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.

[3] 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.

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

[5] 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.

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

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