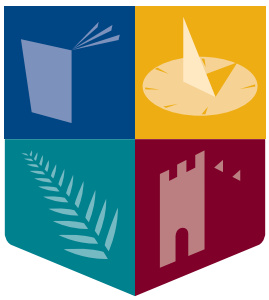
Final Year Project Report

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

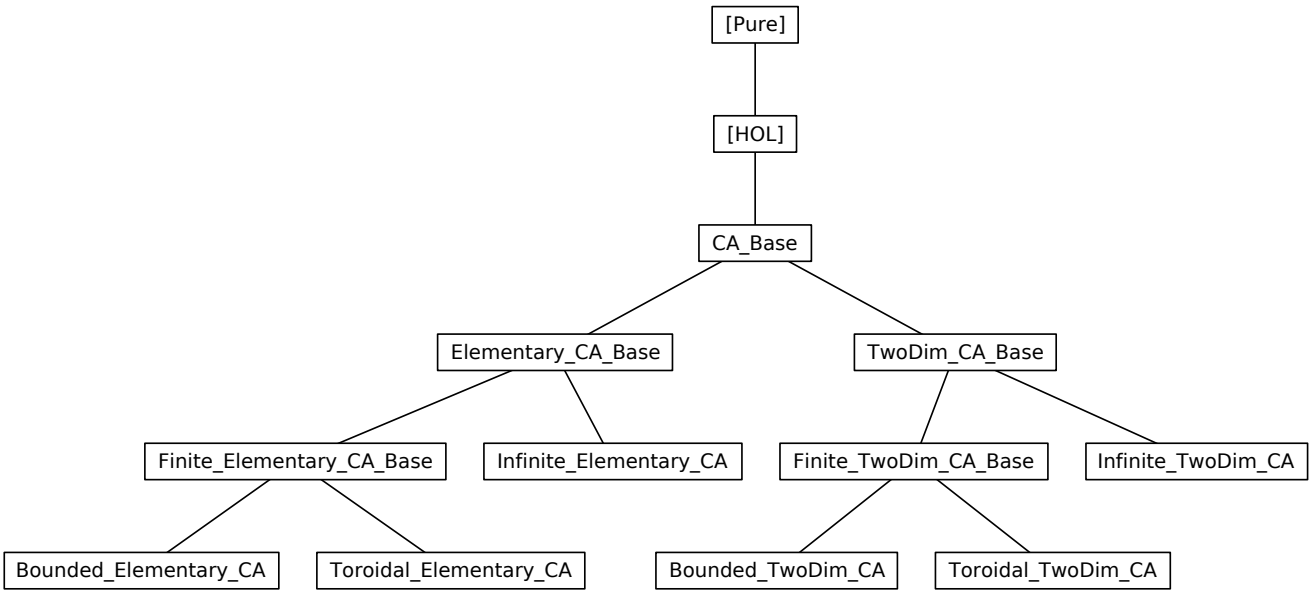


Figure 1.1: Dependency graph of project Theories

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, [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 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)
```

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.

Bounded CA deal with the problem of determining the neighbourhood of a cell on the edge simply via "padding". For the purposes

1.4 Analytical Work

E.g. Equations, etc. that describe your solution

1.5 Low Level

E.g. Method specifications, Algorithms, etc.

1.6 Implementation

Discuss anything interesting here; put full source code in an appendix or attachment

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.