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.

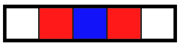

Figure 2.1: Neighbourhood of the blue cell includes itself and the cells in red

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.

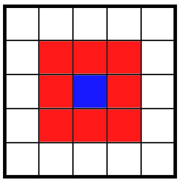

Figure 2.2: Moore neighbourhood

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.

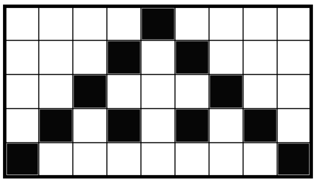

Figure 2.3: Rule 26 traces a Sierpiński triangle over time

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.

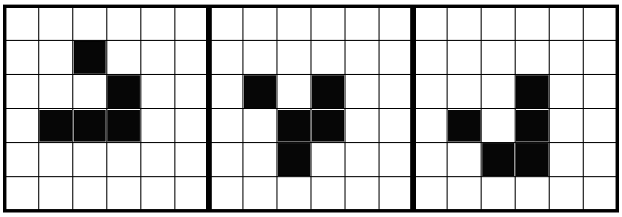

Figure 2.4: 3 steps of a "glider" in Game of Life

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

```
fun get_cell :: "state ⇒ int ⇒ int ⇒ cell" where
"get_cell s x y = s!(nat x)!(nat y)"

fun apply_nb :: "(cell ⇒ cell) ⇒ neighbourhood ⇒
↪  neighbourhood" where
"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

Listing 2.2: A statement of theorem named “t1” along with proof

```
theorem t1 :"n>0 ⟹ ca yields State (run ca n)"
  apply(simp add: yields_def)
  apply(rule exI)
  apply(rule conjI)
  apply(auto)
  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

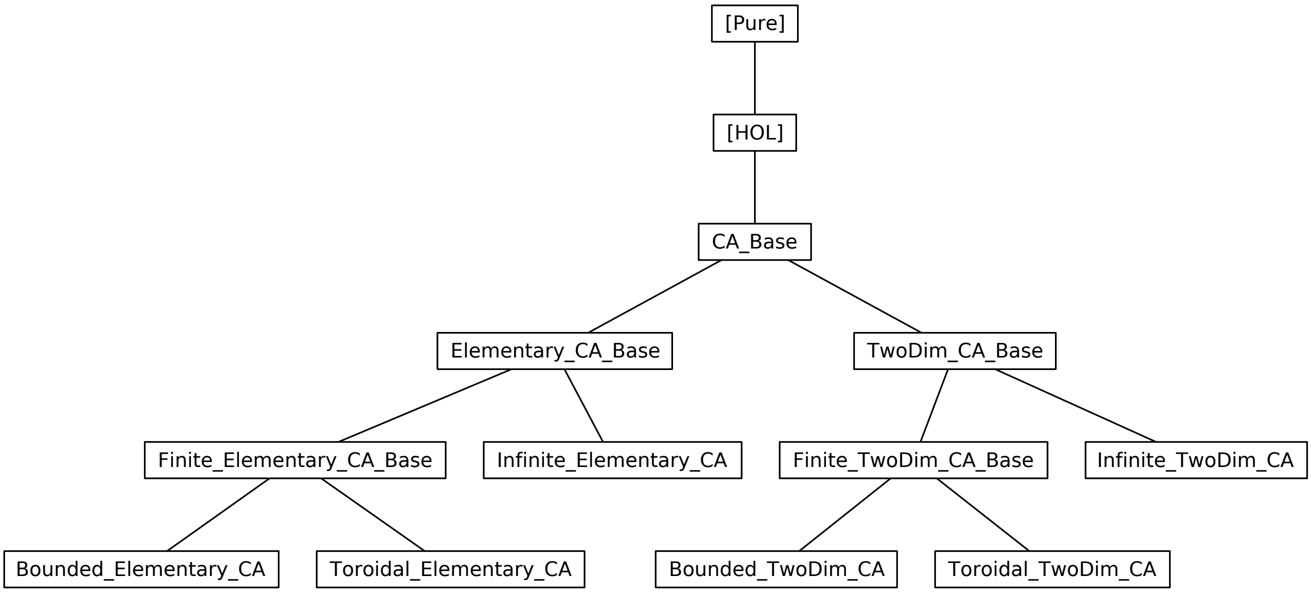

Figure 4.1: Dependency graph of project Theories

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.

4.2 Analytical Work

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

4.3 High Level

E.g. Packages, Class Diagrams, etc.

4.4 Low Level

E.g. Method specifications, Algorithms, etc.

4.5 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.