EPTCS 389

Proceedings of the

7th Symposium on

Working Formal Methods

Bucharest, Romania, 21-22 September 2023

Edited by: Horaţiu Cheval, Laurenţiu Leuştean and Andrei Sipoş

Published: 22nd September 2023
DOI: 10.4204/EPTCS.389
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

The Working Formal Methods Symposium (FROM) is a series of workshops that aim to bring together researchers and practitioners who work on formal methods by contributing new theoretical results, methods, techniques, and frameworks, and/or by creating or using software tools that apply theoretical contributions.

This volume contains the papers presented at Seventh Working Formal Methods Symposium (FROM 2023) held in Bucharest, Romania on September 21-22, 2023. The symposium was co-organized by the University of Bucharest and the Institute for Logic and Data Science and was co-located with the ILDS-FMI Coq and Lean Autumn School.

The scientific program consisted of invited talks by:

- Radu Iosif (CNRS-VERIMAG)
- Ulrich Kohlenbach (Technische Universität Darmstadt)
- Eugenio Omodeo (Università degli Studi di Trieste)
- Alicia Villanueva (Universitat Politècnica de València)

and thirteen contributed papers, which deal with varied topics such as matching logic, static analysis, interactive and automatic theorem proving, epistemic logics, graph theory, computational algebra, neural networks, and P-systems.

The members of the Programme Committee for the workshop were:

- Florin Crăciun (Babeş-Bolyai University of Cluj-Napoca)
- Temur Kutsia (Johannes Kepler University Linz)
- Laurenţiu Leuştean (University of Bucharest & ILDS & IMAR) *(co-chair)*
- Dorel Lucanu (Alexandru Ioan Cuza University of Iaşi)
- Mircea Marin (West University of Timişoara)
- David Nowak (CNRS & University of Lille)
- Peter Csaba Ölveczky (University of Oslo)
- Corina Păsăreanu (NASA & Carnegie Mellon University)
- Andrei Popescu (University of Sheffield)
- Thomas Powell (University of Bath)
- Grigore Roşu (University of Illinois at Urbana-Champaign)
- Vlad Rusu (INRIA Lille)
- Andrei Sipoş (University of Bucharest & ILDS & IMAR) *(co-chair)*

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. iii–iv, doi:10.4204/EPTCS.389.0

- Viorica Sofronie-Stokkermans (University of Koblenz and Landau)

We would like to thank the members of the programme committee and the reviewers for their effort, the authors for their contributions, EPTCS for publishing this volume and Rob van Glabbeek for his immense support to us as editors. We are also grateful for the generous support of our sponsors: BRD – Groupe Société Générale and Runtime Verification.

Horaţiu Cheval
Laurenţiu Leuştean
Andrei Sipoş
Editors

Bucharest, 2023

Modelling and Search-Based Testing of Robot Controllers Using Enzymatic Numerical P Systems

Radu Traian Bobe Florentin Ipate Ionuţ Mihai Niculescu
Department of Computer Science, Faculty of Mathematics and Computer Science, University of Bucharest, Str Academiei 14, Bucharest, 010014, Romania
radu.bobe@s.unibuc.ro florentin.ipate@unibuc.ro ionutmihainiculescu@gmail.com

The safety of the systems controlled by software is a very important area in a digitalized society, as the number of automated processes is increasing. In this paper, we present the results of testing the accuracy of different lane keeping controllers for an educational robot. In our approach, the robot is controlled using numerical P systems and enzymatic numerical P systems. For tests generation, we used an open-source tool implementing a search-based software testing approach.

Keywords: tests generation, numerical P systems, enzymatic numerical P systems, search-based software testing, cyber-physical systems, membrane computing

1 Introduction

Due to the remarkable technological progress of late years, software applications tend to have a considerable role in solving most problems of everyday life. The medical, financial or automotive fields are just three of the main areas in which software products are intensively used. Given the importance of these areas in every individual's life, ensuring product quality and functionality is an essential step in the development process. The safety of software systems for large-scale use is ensured by testing. Software testing aims to validate the fulfillment of the requirements defined for the developed product, as well as to identify possible unwanted behaviors triggered by simulating certain operational contexts.

In this paper, we propose an approach for testing two different lane keeping controllers designed to move an educational robot called *E-puck* [8]. Both controllers are based on numerical P systems, introduced by G. Păun and R. Păun in [11]. We also provide an equivalent version for the models using enzymatic numerical P systems, an extension of numerical P systems, defined by A. Pavel et al. in [9]. For this experiment we used some reliable tools which will be introduced in the following sections.

The paper is structured as described: Section 2 presents the P system variants to be used in the paper. Section 3 introduces the working environment, including the tools used. Section 4 describes the models and the main differences between them, while Section 5 illustrates the testing approach along with the results. In the end, Section 6 presents the future work and conclusions.

2 Preliminaries

Membrane computing is a field of research introduced by Gh. Păun in [10, 12]. The computational paradigm was originally inspired by the structure and functionality of the living cells. Several classes of membrane systems (P systems) have been later defined and investigated, being classified according to the structure of the membranes as *cell-like*, *tissue-like* and *neural-like* P systems. Membrane computing has

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 1–10, doi:10.4204/EPTCS.389.1

made significant breakthroughs in the last decades in fields like computer science, economics or biology. Depending on the requirements, extensions of the main concept were introduced and our experiment involves two of these types: numerical P systems and enzymatic numerical P systems.

2.1 (Enzymatic) Numerical P System

The (enzymatic) numerical P systems [11, 9] are computational models that only inherit the membrane structure from the membrane systems, more exactly a cell-like membrane structure. The membranes contain *variables* and their values are processed by the *programs* on every time unit. The whole system is synchronized by a global clock in discrete time units.

The (enzymatic) numerical P system (EN P system) is defined by the tuple:

$$\Pi = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0))) \tag{1}$$

where:

- $m \geq 1$ is degree of the system Π (the number of membranes);
- H is an alphabet of labels;
- μ is membrane structure;
- Var_i is a set of variables from membrane $i, 1 \leq i \leq m$;
- $Var_i(0)$ is the initial values of the variables from region $i, 1 \leq i \leq m$;
- Pr_i is the set of programs from membrane $i, 1 \leq i \leq m$.

The program $Pr_{l_i,i}$, $1 \leq l_i \leq m_i$ has one of the two following forms:

i) non-enzymatic

$$F_{l_i,i}(x_{1,i}, \ldots, x_{k,i}) \rightarrow c_{1,i}|v_1 + c_{2,i}|v_2 + \cdots + c_{m_i,i}|v_{m_i}$$

where $F_{l_i,i}(x_{1,i}, \ldots, x_{k,i})$ is the production function, $c_{1,i}|v_1 + c_{2,i}|v_2 + \cdots + c_{m_i,i}|v_{m_i}$ is the repartition protocol, and $x_{1,i}, \ldots, x_{k,i}$ are variables from Var_i. Variables $v_1, v_2 \ldots v_{m_i}$ can be from the region where the programs are located, and to its upper and inner compartments, for a particular region i. If a compartment contains more than one program, only one will be chosen in non-deterministically manner.

ii) enzymatic

$$F_{l_i,i}(x_{1,i}, \ldots, x_{k,i})|_{e_i} \rightarrow c_{1,i}|v_1 + c_{2,i}|v_2 + \cdots + c_{m_i,i}|v_{m_i}$$

where e_i is an enzymatic variable from Var_i, $e_i \notin \{x_{1,i}, \ldots, x_{k,i}, v_1, \ldots, v_{m_i}\}$. The program can be applied at time t only if $e_i > min(x_{1,i}(t), \ldots, x_{k,i}(t))$. The programs that meet this condition in a region will be applied in parallel.

When the program is applied by the system at time $t \geq 0$, the computed value

$$q_{l_i,i}(t) = \frac{F_{l_i,i}(x_{1,i}(t), \ldots, x_{k,i}(t))}{\sum_{j=1}^{n_i} c_{j,i}}$$

represents the *unitary portion* that will be distributed to the variables $v_1, \ldots, v_n$, proportional to coefficients $c_{1,i}, \ldots, c_{m_i,i}$, where $c_{j,i} \in \mathbf{N}^+$ and the received values will be $q_{l_i,i}(t) \cdot c_{1,i}, \ldots, q_{l_i,i}(t) \cdot c_{m_i,i}$.

The values of variables, from $t-1$, present in the production functions are *consumed*, reset to zero, and their new value is the sum of the proportions distributed to variable through the repartition protocols, if they appear in them or remain at the value zero.

3 Experimental environment

In this section we will provide brief descriptions of the tools we integrated in our experiment. Firstly, we used an open-source software which allows the simulation of numerical P systems and enzymatic numerical P systems. The simulator is called PeP and will be introduced later in this section. Since we don't have the physical education robot involved in this study, we also used a dedicated platform for robot simulations, called Webots. For tests generation, we used a tool which won the *SBST Tool Competition 2022* [4]. We will discuss later in this section the arguments for using a search-based testing tool.

3.1 PeP simulator

PeP simulator [14] is an open-source product developed by A.Florea and C.Buiu, used for simulations based on numerical P systems and enzymatic numerical P systems. The program is written in Python and receives numerical P systems as an input file. The input file includes the membrane structure and the contents of each membrane, being stored in memory and executed.
PeP can be used as a stand-alone tool for simple simulations and run from the command line with some options, like the number of simulations steps or a csv document generation containing the values at each step of the simulation. As observed in [14], the tool comes with a set of basic input files examples, both numerical P systems and enzymatic numerical P systems.

Besides the simplicity of running this tool, another advantage which can be taken into account when using PeP is that it can be used as an integrated module in complex projects. We used this approach in our experiment in order to make a controller accepted by the robot simulation platform and able to receive information from the platform. In our lane keeping experiments, the simulation ends when the robot drives off the generated lane or when the lane is kept until the end.

3.2 Webots and E-puck

Webots is a robotics simulation software which allows the user to construct a complex environment for programming, modelling and simulating mobile robots. The environment can include multiple scene objects with different properties which can be set from the graphic interface or from the generation files [7]. In addition, the robots can be equiped with a large number of objects called nodes, like sensors, camera, GPS, LED, light sensor etc.
In our approach, additionally to the original equipments of E-puck, we used a GPS attached to the turret

slot in order to examine the coordinates at each step of the simulation. The scenes, called "worlds" in Webots, are containing the road that the robot will try to follow. The roads are generated with Ambiegen, a tool that will be described later in this section. Each world is defined by a *.wbt* file. The objects can be edited in this file and we used this option in order to place the road object in the scene with coordinates exported from Ambiegen. Additional functionalities, like sensors or GPS can also be added to the robot by editing the world file which will be imported in Webots, obtaining the visual representation of the scene.

As mentioned before, for this experiment we used a robot widely known from educational and research purposes, called E-puck. At the moment, the robot has some capabilities that are not implemented in Webots, but considering the fact that both hardware and software components of E-puck are open source, this remains a challenging opportunity [1].

E-puck has eight infrared proximity sensors placed around the body [8]. For lane keeping simulation, we used just six of them: the two sensors placed in front and the four placed two on each side. This aspect can be easily adapted by changing the membrane structure and creating new membranes if more sensors are needed or deleting a few of them if required. Each sensor has a corresponding membrane in the numerical P system model and the association was made in the controller. The robot has two motors attached to the body along with two wheels, and the speed value is also changeable from the controller.

3.3 Ambiegen

Ambiegen is an open-source tool that utilizes evolutionary search for the generation of test scenarios for autonomous systems. It can be used in experiments involving lane keeping assist systems and robots navigating a room with obstacles [6]. The software is developed in Python and uses evolutionary search [16] for tests generation. The main goal of Ambiegen in this approach is to generate roads as test cases in order to challenge E-puck to keep the lane. The tool exports the roads in separate text files as a sequence of points, representing the road spine. From this points, we can build the road with a proportional size to E-puck.

Challenging different LKAS (Lane Keeping Assist Systems) involves a large diversity of road topologies in order to detect the behavior in limit situations, such as narrow curves. Ambiegen figures out the solution for diversity by using a multi-objective genetic algorithm for search-based test generation, called Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [2]. In Ambiegen implementation, NSGA-II has two-objectives: to increase the fault revealing power of test cases and to preserve their diversity [5]. This multi-objective approach, which combines roads generation with a high attention to diversity along with remarkable results at the competition mentioned above, attracted our curiosity to integrate Ambiegen with Webots and testing E-puck on the roads resulted.

3.4 Experimental Procedure

Considering the above information, we will detail the way we worked with the presented tools. PeP and Ambiegen are developed in Python and so is the robot controller.
First of all, we could easily integrate PeP with E-puck controller using the PeP module which allowed us to parse the numerical P system model as an input file for controller. Achieving this, the model membranes were associated with controller variables. Names and constant values (e.g., robot cruise speed) were taken using a text file containing membrane's values of the variables. The values were chosen empirically.

Next is a pseudocode version of the main loop in our controller, which performs the simulation steps.

Algorithm 1 Simulation steps performing algorithm

```
repeat
    for i=1 to number_of_sensors do
        sensor_membrane(i) ← value(i)
    run one simulation step
    read lw, rw from P system
    leftMotor ← lw
    rightMotor ← rw
until the end of the road or E-puck goes out of the road
```

Another challenge for us was to move the robot on the roads exported from Ambiegen with the above presented approach. Ambiegen exports the roads as *.json* files along with informations like test outcome, maximum curvature coefficient etc. We took the road points from the file and wrote them in the world file.

Webots provides the possibility to extend the set of scene nodes by adding custom nodes created by users. The mechanism is called PROTO and is described in [13]. After a node is extending with the PROTO interface, it can be instantiated from the Webots graphic interface.

We used this technique to retrieve the points forming the spines of the roads generated by Ambiegen and putting them into the *wayPoints* of the *Road* node. Using javascript, used as scripting language by the PROTO, we constructed new nodes illustrating the roads from Ambiegen. Then, in the graphic interface of Webots, the road is represented in accordance with the road from Ambiegen. With minimal Python code additions we plotted each generated road with the corresponding spine to confirm that the shape illustrated in Webots respects the original one.

4 Models

In this section we will present two models used to control the robot, the core of the controller. The controller receives data from proximity sensors, that measure distances to obstacles from the environment, to determine the direction of movement of a differential two wheeled robot, E-puck, in our case.

The proximity sensor has a range of 4 cm; if the obstacles are further than this limit the sensor returns the value of 0. The proximity sensors are placed on the left and right side of the robot in the direction of its movement at different angles.

The first model was taken from [3] and adapted. The equations that calculate the linear and angular velocity are shown below:

$$leftSpeed = cruiseSpeed + \sum_{i=1}^{n} weightLeft_i \cdot prox_i$$

$$rightSpeed = cruiseSpeed + \sum_{i=1}^{n} weightRight_i \cdot prox_i$$

The *leftSpeed* and *rightSpeed* are the speeds of the two wheels of the robot. The enzymatic numerical P system described below encapsulates this behavior.

The first model is defined as follows:

$$\Pi_{M_1} = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0)))$$

where:

- $m = k \cdot 3 + 3, k = 6$, where k is the number of proximity sensors;
- $H = \{s, s_c\} \cup \bigcup_{i=1}^{k} \{c_i, s_i, w_i\}$;
- $\mu = [[[]_{s_1}[]_{w_1}]_{c_1} \ldots [[]_{s_k}[]_{w_k}]_{c_k}[]_{s_c}]_s$;
- $Var_s = \{x_{s_l}, x_{s_r}\}, Var_{s_c} = \{x_{s_c}\}$,
  $Var_{c_i} = \{x_{c_i,s_l}, x_{c_i,s_r}, x_{c_i,w_l}, x_{c_i,w_r}, e_{c_i}\}, 1 \leq i \leq k$,
  $Var_{s_i} = \{x_{s_i,i}\}, 1 \leq i \leq k$,
  $Var_{w_i} = \{x_{w_i,w_l}, x_{w_i,w_r}, e_{w_i}\}, 1 \leq i \leq k$;
- $Var_i(0) = 0, 1 \leq i \leq k$;
- $Pr_s = \{0 \cdot x_{s_l} \cdot x_{s_r} \to 1 | x_{s_l} + 1 | x_{s_r}\}$;
  $Pr_{s_c} = \{3x_{s_c} \to 1 | x_{s_c} + 1 | x_{s_l} + 1 | x_{s_r}\}$;
  $Pr_{c_i} = \{x_{c_i,s_l} \cdot x_{c_i,w_l}|_{e_{c_i}} \to 1 | x_{s_l},$
  $x_{c_i,s_r} \cdot x_{c_i,w_r}|_{e_{c_i}} \to 1 | x_{s_r}\}, 1 \leq i \leq k$;
  $Pr_{s_i} = \{3x_{s_i,i} \to 1 | x_{s_i,i} + 1 | x_{c_i,s_l} + 1 | x_{c_i,s_r}\}, 1 \leq i \leq k$;
  $Pr_{w_i} = \{2x_{w_i,w_l}|_{e_{w_i}} \to 1 | x_{w_i,w_l} + 1 | x_{c_i,w_l},$
  $2x_{w_i,w_r}|_{e_{w_i}} \to 1 | x_{w_i,w_r} + 1 | x_{c_i,w_r}\}, 1 \leq i \leq k$;

The meaning of the variables from the model is the following:

- x_{s_l} and x_{s_r} from the region s represent *leftSpeed* and *rightSpeed*, the sum of the products are accumulated in s ;
- x_{s_c} from the compartment s_c is *cruiseSpeed*;
- each pair of weights, $weightLeft_i$ and $weightRight_i$, resides in the regions w_i, $1 \leq i \leq k$;
- for each proximity sensor, $prox_i$, a compartment is defined, namely s_i, containing a single variable, $x_{s_i,i}$, $1 \leq i \leq k$;
- the products are calculated by two distinct programs, $weightLeft_i \cdot prox_i$, and $weightRight_i \cdot prox_i$, $1 \leq i \leq k$, in the compartments c_i.

The second model is an improvement on the first one. First we define the function

$$f(x) = \begin{cases} 1, & \text{if } x = 0 \\ 0, & \text{otherwise} \end{cases}$$

This function will be used in the equations describing the behavior of the model and in the production functions from the programs.

The equations describing the behavior are:

$$\begin{aligned} weightLeft &= \sum_{i=1}^{n} weightLeft_i \cdot prox_i \\ weightRight &= \sum_{i=1}^{n} weightRight_i \cdot prox_i \\ leftSpeed &= cruiseSpeed \cdot weightLeft + f(weightLeft) \cdot cruiseSpeed \\ rightSpeed &= cruiseSpeed \cdot weightRight + f(weightRight) \cdot cruiseSpeed \end{aligned}$$

The model is defined as follows:

$$\Pi_{M_2} = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0)))$$

where:

- $m = 3k+3, k = 6$;
- $H = \{s, w, s_c\} \cup \bigcup_{i=1}^{k} \{c_i, s_i, w_i\}$;
- $\mu = [[[[]_{s_1}[]_{w_1}]_{c_1} \ldots [[]_{s_k}[]_{w_k}]_{c_k}[]_{s_c}]_w]_s$;
- $Var_s = \{x_{s_l}, x_{s_r}\}, Var_w = \{x_{w_l}, x_{w_r}, e_w\}, Var_{s_c} = \{x_{s_c}\}$,
  $Var_{c_i} = \{x_{c_i,s_l}, x_{c_i,s_r}, x_{c_i,w_l}, x_{c_i,w_r}, e_{c_i}\}, 1 \leq i \leq k$,
  $Var_{s_i} = \{x_{s_i,i}\}, 1 \leq i \leq k$,
  $Var_{w_i} = \{x_{w_i,w_l}, x_{w_i,w_r}, e_{w_i}\}, 1 \leq i \leq k$;
- $Var_i(0) = 0, 1 \leq i \leq k$;
- $Pr_s = \{0 \cdot x_{s_l} \cdot x_{s_r} \to 1 | x_{s_l} + 1 | x_{s_r}\}$;
  $Pr_w = \{x_{s_c} \cdot x_{w_l} + f(x_{w_l}) \cdot x_{s_c}|_{e_w} \to 1 | x_{s_l},$
  $x_{s_c} \cdot x_{w_r} + f(x_{w_r}) \cdot x_{s_c}|_{e_w} \to 1 | x_{s_r}\}$;
  $Pr_{s_c} = \{x_{s_c} \to 1 | x_{s_c}\}$;
  $Pr_{c_i} = \{x_{c_i,s_l} \cdot x_{c_i,w_l}|_{e_{c_i}} \to 1 | x_{s_l},$
  $x_{c_i,s_r} \cdot x_{c_i,w_r}|_{e_{c_i}} \to 1 | x_{s_r}\}, 1 \leq i \leq k$;
  $Pr_{s_i} = \{3x_{s_i,i} \to 1 | x_{s_i,i} + 1 | x_{c_i,s_l} + 1 | x_{c_i,s_r}\}, 1 \leq i \leq k$;
  $Pr_{w_i} = \{2x_{w_i,w_l}|_{e_{w_i}} \to 1 | x_{w_i,w_l} + 1 | x_{c_i,w_l},$
  $2x_{w_i,w_r}|_{e_{w_i}} \to 1 | x_{w_i,w_r} + 1 | x_{c_i,w_r}\}, 1 \leq i \leq k$;

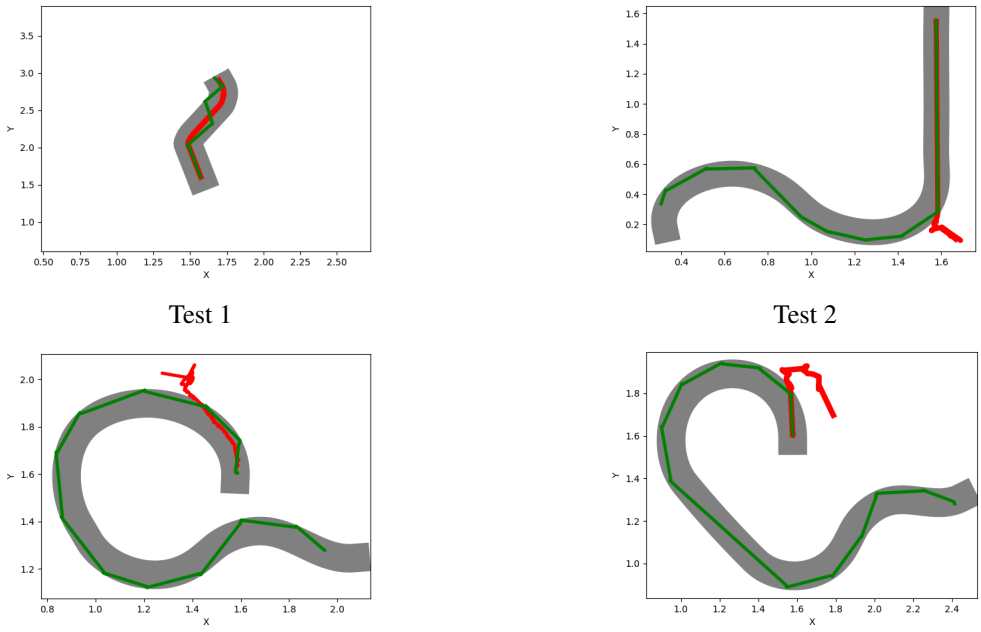
As we see from the definition of the Π_{M_1} and its equations, in the proximity of an obstacle, roadside, the speed of the wheel on the side with the obstacle increases according to the weights. In certain circumstances, particularly related to the geometry of a tight road curve, the sum between travel speed and the weights results in a rotation in the opposite direction of the obstacle, but a slight forward motion still continues, causing the robot to leave the road.

The second model, Π_{M_2}, overcomes this problem by calculating the product of the travel speed and the sum of the weights. In a similar situation, the second model performs an angular rotation, in the opposite direction to the obstacle, and when the proximity sensors no longer detect obstacles, it continues moving forward with a constant velocity. This behavior is modeled by compartment w.

In the next section, it can be seen that the Π_{M_2} model has better behavior in similar situations compared to the first model.

5 Simulation Results

Having presented the way we integrate the tools along with the models we will detail in this section the testing stage. Ambiegen offers the possibility to configure the setup for its genetic algorithms, choosing the values for parameters like population size, number of generations, mutation rate or crossover rate. Also, aspects like the amount of time allocated for tests generation, map size, out of bound percent from which the test is considered as failed can be set easily from the command line, when executing the main Python file used in [4].



Test 1 Test 2

Test 3 Test 4

Figure 1: Illustration of different road tests

After series of trials we kept the default values, so we used the population size 100 with 75 generations. Mutation rate is 0.4 and crossover rate is 1. These values can be changed from the internal configuration file of Ambiegen. From the command line we chose the time budget allocated to generation and execution to be 1800 seconds and the map size to 200x200 meters which is the default value. For the out of bound percentage we also kept the default value (95%). After each simulation, Ambiegen exported roads spines coordinates as text files and we also plotted each generated road, as mentioned before. Then we could easily chose different roads based on the number of curves and their angle as the main criteria to diversify the tests that were supposed to be given to E-puck controller in Webots.

Next we report some roads along with the simulation results. The road is marked with grey, whilst the trajectory using the first model is represented with red. The trajectory of the second model (the improved one) is colored with green.

In Figure 1 we can see the results of different roads tests. We observe that Test 1, being the simplest test of the above presented, is passed by both models. In the next scenarios, the complexity of the road increases and only the improved model can pass.

From all the experiments, we noticed that the first model (the one marked with red in results) cannot pass a huge majority of tests with curves, whilst the improved model performs a rotation movement when

the road is curved and for this reason it manages to advance until the end of the road. Additional tests were added to [15].

6 Conclusions and Future Work

In this paper we presented an approach to test different enzymatic numerical P systems models using modern tools and search-based generated tests. We evaluated our approach on a research and educational robot called E-puck, virtually represented in Webots simulator. We set up our working environment incorporating the tools with different scripts created to ensure a smooth integration between them and also a better data processing. We formally described each model involved and then showed the differences between the lane-keeping controllers resulted when using each of them. As future work, we will investigate the possibility to dynamically calculate the values for weights, which are at the moment empirically assigned. Based on the controller behavior during the previous tests, the weights values will be automatically adapted. Also, we will try to develop a method to generate more complex roads in order to better challenge the controllers.

References

[1] Micael S Couceiro, Patricia A Vargas & Rui P Rocha (2014): *Bridging the reality gap between the Webots simulator and e-puck robots*. Robotics and Autonomous Systems 62(10), pp. 1549–1567, doi:10.1016/j.robot.2014.05.007.

[2] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal & TAMT Meyarivan (2002): *A fast and elitist multiobjective genetic algorithm: NSGA-II*. IEEE transactions on evolutionary computation 6(2), pp. 182–197, doi:10.1109/4235.996017.

[3] Andrei George Florea & Cătălin Buiu (2017): *Modelling multi-robot interactions using a generic controller based on numerical P systems and ROS*. In: 2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), pp. 1–6, doi:10.1109/ECAI.2017.8166411.

[4] Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio & Fiorella Zampetti (2022): *SBST tool competition 2022*. In: Proceedings of the 15th Workshop on Search-Based Software Testing, pp. 25–32, doi:10.1145/3526072.3527538.

[5] Dmytro Humeniuk, Giuliano Antoniol & Foutse Khomh (2022): *AmbieGen tool at the SBST 2022 Tool Competition*. In: Proceedings of the 15th Workshop on Search-Based Software Testing, pp. 43–46, doi:10.1145/3526072.3527531.

[6] Dmytro Humeniuk, Foutse Khomh & Giuliano Antoniol (2023): *AmbieGen: A Search-based Framework for Autonomous Systems Testing*. arXiv preprint arXiv:2301.01234, doi:10.48550/arXiv.2301.01234.

[7] Olivier Michel (2004): *Cyberbotics ltd. webots™: professional mobile robot simulation*. International Journal of Advanced Robotic Systems 1(1), p. 5, doi:10.5772/5618.

[8] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Stephane Magnenat, Jean-Christophe Zufferey, Dario Floreano & Alcherio Martinoli (2009): *The e-puck, a robot designed for education in engineering*. In: Proceedings of the 9th conference on autonomous robot systems and competitions, IPCB: Instituto Politécnico de Castelo Branco, pp. 59–65.

[9] Ana Pavel, Octavian Arsene & Catalin Buiu (2010): *Enzymatic numerical P systems-a new class of membrane computing systems*. In: 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), IEEE, pp. 1331–1336, doi:10.1109/BICTA.2010.5645071.

[10] Gheorghe Păun (2002): *Membrane computing: an introduction*. Springer Science & Business Media, doi:10.1007/978-3-642-56196-2.

[11] Gheorghe Păun & Radu Păun (2006): *Membrane computing and economics: Numerical P systems*. *Fundamenta Informaticae* 73(1-2), pp. 213–227. Available at http://content.iospress.com/articles/fundamenta-informaticae/fi73-1-2-20.

[12] Gheorghe Păun, Grzegorz Rozenberg & Arto Salomaa, editors (2010): *The Oxford Handbook of Membrane Computing*. Oxford University Press.

[13] Webots reference manual https://cyberbotics.com/doc/reference/proto.

[14] A. G. Florea and C. Buiu, "PeP - an open-source software simulator of Numerical P systems and Numerical P systems with Enzymes," 2017. [Online]. Available: https://github.com/andrei91ro/pep.

[15] Github simulation results folder https://github.com/radubobe/Research/tree/main/Modelling%20and%20testing%20robot%20controllers%20using%20ENPS/Simulation%20results.

[16] Darrell Whitley, Soraya Rana, John Dzubera & Keith E Mathias (1996): *Evaluating evolutionary algorithms*. *Artificial intelligence* 85(1-2), pp. 245–276, doi:10.1016/0004-3702(95)00124-7.

Using Z3 to Verify Inferences in Fragments of Linear Logic

Alen Docef
Department of Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, Virginia, USA
adocef@vcu.edu

Radu Negulescu
rnegulescu@vcu.edu
rnegules@gmail.com

Mihai Prunescu
Research Center for Logic, Optimization and Security
Faculty of Mathematics and Computer Science
University of Bucharest, Bucharest, Romania
Simion Stoilow Institute of Mathematics
Romanian Academy, Bucharest, Romania
mihai.prunescu@imar.ro
mihai.prunescu@gmail.com

Linear logic is a substructural logic proposed as a refinement of classical and intuitionistic logics, with applications in programming languages, game semantics, and quantum physics. We present a template for Gentzen-style linear logic sequents that supports verification of logic inference rules using automatic theorem proving. Specifically, we use the Z3 Theorem Prover [8] to check targeted inference rules based on a set of inference rules that are presumed to be valid. To demonstrate the approach, we apply it to validate several derived inference rules for two different fragments of linear logic: MLL+Mix (Multiplicative Linear Logic extended with a Mix rule) and MILL (Multiplicative Intuitionistic Linear Logic).

Keywords: linear logic, MLL+Mix, MILL, Z3, inference rules
M.S.C. Classification: 03B47, 03F52

1 Introduction

The Z3 Theorem Prover [8] is a satisfiability modulo theories (SMT) solver targeted at software verification and program analysis. Besides SMT, the symbolic reasoning engine of Z3 also uses automatic reasoning, incremental solving, model generation, and other artificial intelligence techniques to determine satisfiability of a set of rules in a theory and to produce models.

Linear logic [13] is a substructural logic proposed as a refinement of classical and intuitionistic logics, with applications in programming languages, game semantics, and quantum physics. In [20], the author makes a functorial connection between arbitrary models of multiplicative linear logic and the category of presheaves over arbitrary rings. Other far-reaching considerations connected with category theory are made by the same author in [19]. A more accessible approach to this connection is presented in [26]. Connections with semantics for higher order quantum computing were studied in [22]. A usual interpretation of linear logic, already intended by Girard, is that formulas do not hold values as true and false, but contain information about the availability and use of given resources. In this context, [21] presents an overview of linear logic programming. The article [4] sketches a unified approach, a "Rosetta stone", based on categories as well, for interpreting linear logic in three seemingly unrelated domains: topology, quantum physics, and lambda calculus. Relations between linear logic and concurrency theory are an active area of research, for instance in [2]. General presentations of linear logic can be found in: [10], [9], [14], [17], [27], [28].

Two important fragments of linear logic are multiplicative intuitionistic linear logic (MILL) and multiplicative linear logic with the Mix-rule (MLL+Mix). MILL is crystallized in [5], where its proof-theory is studied from a categorical theoretic point of view. A variant of MILL and its proof methods

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 11–25, doi:10.4204/EPTCS.389.2

are discussed recently in [12]. MLL+Mix is crystallized in [1] as a result of the authors' game-theoretic research in linear logic. They considered formulas as games and proofs as winning strategies.

As observed for instance in [6], in linear logic some usual proof steps, such as weakening and contraction, are restricted. The proof complexity in general linear logic is Σ_0^1-hard. Some fragments are better behaved, the proof complexity in the multiplicative fragment being NP-complete. Still, proofs in linear logic are recognized to be computationally difficult.

We propose a template for modeling Gentzen-style sequents that supports verification of logic inference rules using Z3. Using our template, Z3 checks targeted inference rules based on a set of inference rules presumed to be valid. If a targeted rule does not hold, Z3 can produce a model assignment which serves as a counterexample.

We show how to apply our template to validate derived inference rules in two different fragments of linear logic. We find that this approach can flatten the learning curve when switching from classical logic to more general types of logic. Such transitions can be confusing. In particular, when concrete problems involving resource management are encoded in the satisfiability check of linear logic formulas, there is value added in using a theorem prover to offset the lack of common mathematical intuition.

Theorem provers have been used to assist in generating Gentzen-style proofs, for example Z3 in [18] and Lean in [23]. However, we are not aware of a previous attempt to generalize the approach using a template for modeling Gentzen sequents. Our choice for Z3 was motivated by convenience and we hope it will offer a benchmark for developing Lean tactics for MILL and MLL+Mix as well.

2 Modeling Inference Rules

A *multiset* is a set with multiplicities: the same element can occur several times in a multiset, and the number of occurrences is called multiplicity. While the classical multiplicity of some element in a set is 1, in a multiset elements may have as a multiplicity every set-cardinality. As our multisets of formulas will be later identified with formulas, we will consider only finite multisets. Implicitly, multiplicities are also finite.

In the sequent notation, Γ, Δ, etc., stand for multisets of formulas. A and B represent formulas. The turnstile symbol ($\vdash$, read 'entails') separates the context (antecedents) from the conclusion (consequent). Generally, an inference rule in sequent calculus, including linear logic, takes the following form:

$$\frac{\Gamma_1 \vdash A_1 \qquad \Gamma_2 \vdash A_2 \quad \ldots \quad \Gamma_n \vdash A_n}{\Gamma \vdash A}\ \textit{(Name of Rule)}$$

Above the line, we have the assumptions or premises of the rule, and below the line is the guarantee or conclusion of the rule, which holds true if all the antecedents are true. Each $\Gamma_i \vdash A_i$ is a sequent, where the context Γ_i can be understood as a set of additional assumptions. Such a rule can be stated equivalently using Boolean logic:

$$(\Gamma_1 \vdash A_1) \wedge (\Gamma_2 \vdash A_2) \wedge \ldots \wedge (\Gamma_n \vdash A_n) \rightarrow (\Gamma \vdash A),$$

where $\wedge$ and $\rightarrow$ are logical conjunction and implication, respectively, meaning: "If $\Gamma_1 \vdash A_1$ and $\Gamma_2 \vdash A_2$ and $\ldots$ and $\Gamma_n \vdash A_n$ then $\Gamma \vdash A$".

For example, the tensor rule in linear logic often has the following form [4, p.40]:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma \otimes \Delta \vdash A \otimes B}\ (\otimes)$$

This rule states that if from context Γ we can deduce A, and from context Δ we can deduce B, then from the context 'Γ tensor Δ' ($\Gamma \otimes \Delta$) we can deduce 'A tensor B' ($A \otimes B$). The same tensor rule is sometimes stated in an alternate form [16, p.3]:

$$\frac{\Gamma, A \qquad \Delta, B}{\Gamma, \Delta, A \otimes B}\,(\otimes) \qquad \text{meaning} \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\,(\otimes)$$

When the antecedent is missing, the turnstile symbol ($\vdash$), usually read as 'entails', can be read as 'provable'. That is, the consequent is provable from given rules, meaning the consequent is entailed by an empty context. In such cases, the turnstile can be omitted for legibility.

For the purpose of verification in Z3, we use its Python API to model inference rules as follows. First, we import the Z3 module and declare several objects used to model inference rules:

- A solver, which will collect all the rules and verify their consistency.
- A sort `F` for formulas and contexts (i.e. multisets of formulas). Sorts are the Z3 model for types.
- A function 'entails' which takes two `F`s (formulas or contexts) and returns a Boolean signifying that the left formula or context entails the right formula or context.
- Two operations of linear logic ('tensor' and 'lollipop') as functions of `F`s returning `F`s.
- Three variables (called *constants* in Z3) of type `F`.

The relevant code fragment is given below. The complete code for this section is provided in Listing 1.

```
## Importing the z3 module
from z3 import *

## Declarations
ll = Solver()
F = DeclareSort('F')
entails = Function('entails', F, F, BoolSort())
tensor = Function('tensor', F, F, F)
lpop = Function('lollipop', F, F, F)
x, y, z = Consts('x y z', F)
```

The operators we declared satisfy inference rules. In our model, each inference rule is universally quantified by the variables in it. Then follows an expression that returns a Boolean. The expression is in prefix notation, i.e., operator followed by operands. Some expressions use `Implies` and `And`; these operators, predefined in Z3, have Boolean parameters and return values.

Consider a simple example where there are only two given inference rules:

$$X \vdash X \qquad \text{and} \qquad (X \otimes Y \vdash Z) \leftrightarrow (Y \vdash X \multimap Z)$$

These are a subset of the inference rules in MILL, used here for illustration only; we will revisit these two rules later in Section 4. We add these two rules to our `ll` solver by invoking its `.add` method:

```
## Given rules
ll.add(ForAll([x], entails(x, x))) # (i)
ll.add(ForAll([x,y,z], entails(tensor(x,y),z) == entails(y,lpop(x,z)))) # (c)
```

To prove that a derived inference rule is valid, we add its *negation* to the existing set of rules and we check the satisfiability of the expanded set of rules. Unsatisfiability proves that the new rule is valid. In our template, checking for satisfiability is implemented using the `ll.check()` method. Upon executing this check, Z3 prints either `'sat'` or `'unsat'` depending on whether a model is found that satisfies the set of rules, including the negated rule.

- If Z3 prints `'unsat'`, the proposed rule results from the previously added rules. To see that, let $r_1, \ldots, r_n$ be the given rules, and let r be the new rule prior to negation. If the set of rules is unsatisfiable that means we have

$$\neg(r_1 \wedge r_2 \wedge \ldots \wedge r_n \wedge \neg r)$$

  Therefore, $\neg(r_1 \wedge r_2 \wedge \ldots \wedge r_n) \vee r$. Thus, $(r_1 \wedge r_2 \wedge \ldots \wedge r_n) \rightarrow r$. In short, if Z3 prints `'unsat'`, we have $(r_1 \wedge r_2 \wedge \ldots \wedge r_n) \rightarrow r$.

- If Z3 prints `'sat'`, it found a model, an assignment for the variables in the rules, under which the new rule r is false (its negation is true) while the previously added rules are all true. Thus, if Z3 prints `'sat'`, we do not have $(r_1 \wedge r_2 \wedge \ldots \wedge r_n) \rightarrow r$. If Z3 prints `'sat'`, we can also use `ll.print()` to obtain the model that caused the satisfiability, that is, under which assignment of variables did the previous rules and the negation of the new rule hold true.

For example, this is the modus ponens rule in linear logic: $x \otimes (x \multimap y) \vdash y$. To show that this rule follows from the two given rules, we add its negation to the solver. Running the satisfiability check as described above, we obtain an `'unsat'` result, thus proving the derived rule. The relevant code fragment is:

```
## Derived rules
ll.add(Not(ForAll([x,y], entails(tensor(x,lpop(x,y)),y))))

## Verification
print(ll.check())
print(ll.model())
```

It is important to verify consistency of the given inference rules without the new negated rule. An inconsistent set would produce `'unsat'` results even without the new negated rule. As a result, the given set of inference rules, if inconsistent, would imply the new negated rule because `false` implies anything. Having commented out the negated modus ponens rule, we verify that the given set of rules is consistent by running a satisfiability check. The solver produces a `'sat'` result, indicating that the given set of rules is indeed consistent.

3 Verifying MLL+Mix Properties

Multiplicative Linear Logic (MLL) [16] is a fragment of linear logic [1] that restricts the linear logic structure to the multiplicative operators. MLL+Mix is an extension of MLL with the Mix inference rule [13]. MLL+Mix has been applied to game theory as shown in [1, p.546, Table 1].

For the MLL inference rules we follow the notation in [16]. For each group of formulas in [16, Fig.1 p.3] there is an implicit entails symbol with an empty antecedent context. For example, the conclusion of formula *(ax)* is written $A, A^\star$ representing actually $\vdash A, A^\star$. To represent more easily entails with an empty antecedent, we define a unary predicate `provable` on our sort `F`, that is, a function from `F` to Booleans, as follows (for the complete code see Listing 2):

```
## Declarations
provable = Function('provable', F, BoolSort())
```

After restoring the entails symbols, the given inference rules of MLL+Mix are rephrased as follows.

$$\frac{\vdash \Gamma}{\vdash \Gamma, \bot}\,(\bot) \qquad \frac{}{\vdash 1}\,(1) \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\,(\parr)$$

$$\frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\,(\otimes) \qquad \frac{}{\vdash A, A^\star}\,(ax) \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, A^\star}{\vdash \Gamma, \Delta}\,(cut)$$

For the Mix rule, we use the formula from [1, p.546, Table 1]

$$\frac{\vdash \Gamma \qquad \vdash \Delta}{\vdash \Gamma, \Delta}\,(mix)$$

The given inference rules of MLL+Mix are modeled in Z3 as follows:

```
## Declarations
comma = Function('comma', F, F, F)
par = Function('par', F, F, F)
tensor = Function('tensor', F, F, F)
dual = Function('dual', F, F)
g, d, l, a, b, bot, one = Consts('g d l a b bot one', F)

## Axioms for comma (multiset reunion): associativity and commutativity
ll.add(ForAll([a, b, g], comma(a,comma(b,g)) == comma(comma(a,b),g)))
ll.add(ForAll([a, b], comma(a,b) == comma(b,a)))

## Given rules (Heijltjes and Houston page 3)
ll.add(ForAll([g],Implies(provable(g), provable(comma(g,bot)))))
ll.add(provable(one))
ll.add(ForAll([g, a, b],Implies(
    provable(comma(comma(g,a),b)),
    provable(comma(g,par(a, b))))))
ll.add(ForAll([g, a, d, b],Implies(
    And(provable(comma(g,a)), provable(comma(d,b))),
    provable(comma(g,comma(d,tensor(a, b)))))))
ll.add(ForAll([a],provable(comma(a,dual(a)))))
ll.add(ForAll([g, d, a],Implies(
    And(provable(comma(g,a)), provable(comma(d,dual(a)))),
    provable(comma(g,d)))))

## Mix rule (https://www.pls-lab.org/en/Mix_rule)
ll.add(ForAll([g, d],Implies(And(provable(g), provable(d)),
    provable(comma(g,d)))))
```

We also introduced an additional operator, called ‘comma’, which adjoins two `F`s into a new `F`. It models the union of multisets of formulas, or multisets and single formulas. For this operator, we define two additional rules (associativity and commutativity). These rules ensure that contexts (i.e. multisets of formulas) created using the comma operator are equivalent regardless of the ordering of formulas in them.

Different authors use different notations for linear logic and its fragments. For example, the same rule (the $\otimes$ ‘tensor’ rule) is presented differently in [4, p.40] and [16, p.3]. However, the two formulations can be linked by interpreting ‘comma’ as ‘par’. This link also provides the rationale for our merging the contexts and formulas in the same sort. As in [4] a context (multiset of formulas) is seen here as the par of the formulas in that context. Informally, par ($\parr$) can be thought of as a parallel composition of devices

[2]. In what follows, we show that this viewpoint over the meaning of contexts supports the verification of properties in two different fragments of linear logic, following the notations in [4] and [16].

First, we verify that our model for the rules of MLL+Mix is indeed satisfiable. Indeed, `ll.check()` returns `'sat'` on our model of the rules of MLL+Mix, indicating that our rules are consistent.

As shown in Section 2, we can verify a new inference rule in MLL+Mix by adding its negation to the solver and verifying the satisfiability of the resulting set of rules. Using this approach, we verified, one by one, the validity of the following derived rules from [16, Fig. 2 and Fig. 3]:

$$\frac{}{\vdash 1,\bot}\,(mix1) \qquad \frac{}{\vdash A\otimes B, A^\star \parr B^\star}\,(mix2) \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, B \qquad \vdash \Lambda, A^\star, B^\star}{\vdash \Gamma,\Delta,\Lambda}\,(mix3)$$

The relevant code is as follows:

```
## Derived rules (Heijltjes and Houston page 4, all "unsat")
# ll.add(Not(provable(comma(one,bot))))
# ll.add(Not(ForAll([a,b],
#     provable(comma(tensor(a,b), par(dual(a),dual(b)))))))
# ll.add(Not(ForAll([a,b,g,d,l],
#     Implies(
#         And(And(provable(comma(g,a)),provable(comma(d,b))),
#             provable(comma(comma(l,dual(a)),dual(b)))),
#         provable(comma(comma(g,d),l))))))
```

We also verify several alternate formulations to the Mix rule in the context of MLL, thought to be equivalent to Mix. We find that these formulations are not equivalent to Mix. One at a time, we check the following negated rules are satisfiable in the context of MLL + Mix, indicating that the respective rules do not hold.

$$\vdash 1 \leftrightarrow \bot \qquad \vdash (\bot^\star \parr 1) \qquad \vdash \bot$$

The relevant code is:

```
## Invalid derived rules (all "sat")
# ll.add(Not(one == bot))
# ll.add(Not(provable(par(dual(bot),one))))
# ll.add(Not(provable(bot)))
```

To sum up the results of our investigations into MLL+Mix inference rules, we can state the following Theorem:

Theorem 1. *The following sequents (and rule) are provable in MLL+Mix:*

$$\vdash 1,\bot \qquad \vdash A\otimes B, A^\star \parr B^\star \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, B \qquad \vdash \Lambda, A^\star, B^\star}{\vdash \Gamma,\Delta,\Lambda}$$

The following sequents are not provable in MLL+Mix:

$$\vdash 1 \leftrightarrow \bot \qquad \vdash \bot^\star \parr 1 \qquad \vdash \bot$$

The theory MLL+Mix can be substantially improved by adding some other axioms. Consider, as an example, the following form of Contraction (C):

$$\frac{\vdash 1, A}{\vdash A}\,(C)$$

We add the Contraction rule (C) to the solver as follows:

```
## Contraction (subset) rule and resulting theorem
# ll.add(ForAll([a, b], Implies(provable(comma(one,a)),provable(a))))
# ll.add(Not(provable(bot))) # unsat
```

The new theory MLL+Mix+C proves to be consistent by using Z3. This extension leads to a strictly stronger theory, as shown by Theorem 2 below:

Theorem 2. *The theory MLL+Mix+C proves the sequent:*

$$\vdash \bot.$$

Proof. From the rule (1) we get

$$\vdash 1.$$

From the rule $(\bot)$, we get

$$\vdash 1, \bot.$$

Finally, from the rule (C) we get

$$\vdash \bot.$$

□

It is however not certain that this extended theory is still a fragment of Linear Logic. Another possibility is to consider the following axiom, called $(\emptyset)$, which introduces sequents with empty hypothesis and empty conclusion:

$$\frac{}{\vdash} \ (\emptyset)$$

It is clear that MLL+Mix+$(\emptyset)$ proves $\vdash \bot$ if one also introduces the convention

$$\forall x \ \ \mathrm{Comma}(\emptyset, x) = x.$$

By $(\emptyset)$ we get the empty sequent $\vdash$. By $(\bot)$, we get $\vdash, \bot$ and applying the convention, this is $\vdash \bot$. The Z3 implementation of the new rule $(\emptyset)$ remains to be completed.

4 Verifying MILL Properties

A fragment of Linear Logic called Multiplicative Intuitionistic Linear Logic (MILL) is shown in [4] to be closely related to models of topology, quantum physics, and lambda calculus. Using the technique in Section 2, we model MILL in Z3 and we investigate several properties of MILL. The complete code is provided in Listing 3.

Our starting point is the set of rules *(i), (o), (⊗), (a), (l), (r), (b), (c)* from [4, p.40].

$$\frac{}{X \vdash X} \ (i) \qquad \frac{X \vdash Y \quad Y \vdash Z}{X \vdash Z} \ (o)$$

$$\frac{W \vdash X \quad Y \vdash Z}{W \otimes Y \vdash X \otimes Z} \ (\otimes) \qquad \frac{W \vdash (X \otimes Y) \otimes Z}{W \vdash X \otimes (Y \otimes Z)} \ (a)$$

$$\frac{X \vdash I \otimes Y}{X \vdash Y} \ (l) \qquad \frac{X \vdash Y \otimes I}{X \vdash Y} \ (r)$$

$$\frac{W \vdash X \otimes Y}{W \vdash Y \otimes X} \ (b) \qquad \frac{X \otimes Y \ \vdash \ Z}{Y \vdash X \multimap Z} \ (c)$$

Rule *(i)* with no premises is modeled in Z3 by simply adding `entails(x,x)`, universally quantified. There is no need for an `Implies` in this rule because the empty set of premises means the conjunction of premises in that set is true. Thus, rule *(i)* is $(\text{true} \rightarrow (x \vdash x))$, which is equivalent to $(\neg\text{true} \vee (x \vdash x))$, which is equivalent to just $(x \vdash x)$. Finally, we model rule *(i)* by adding to the solver an expression with no Boolean logic operators. Rule *(i)*, as well as rules *(o)* and $(\otimes)$ are modeled using the approach described in Section 2. The relevant code is:

```
## Given rules
ll.add(ForAll([x], entails(x, x))) # rule (i)
ll.add(ForAll([x,y,z], Implies(And(
    entails(x,y), entails(y,z)), entails(x, z)))) # rule (o)
ll.add(ForAll([w,x,y,z], Implies(And(
    entails(w,x), entails(y,z)), entails(tensor(w, y), tensor(x, z)))))
    # rule (tensor)
```

The double bar represents implication both ways, i.e., logical equivalence. Accordingly, rules *(a), (l), (r), (b), (c)* are modeled essentially by the technique in Section 2, except that instead of `Implies` we use `==` for logic equivalence. The relevant code for these rules is:

```
## Given rules
ll.add(ForAll([w,x,y,z],
    entails(w,tensor(tensor(x,y),z)) == entails(w, tensor(x, tensor(y, z)))))
    # rule (a)
ll.add(ForAll([x,y],
    entails(x,tensor(I,y)) == entails(x,y))) # rule (l)
ll.add(ForAll([x,y],
    entails(x,tensor(y,I)) == entails(x,y))) # rule (r)
ll.add(ForAll([w,x,y],
    entails(w,tensor(x,y)) == entails(w,tensor(y,x)))) # rule (b)
ll.add(ForAll([x,y,z],
    entails(tensor(x,y),z) == entails(y,lpop(x,z)))) # rule (c)
```

After adding to the solver the eight rules of MILL, we attempt to verify several new inference rules. We start with two properties from [4, p.41]: modus ponens and internal composition. Ther relevant code for both rules is:

```
## Derived rules (all "unsat")
# ll.add(Not(ForAll([x,y], entails(tensor(x,lpop(x,y)),y)))) # rule (ev)
# ll.add(Not(ForAll([x,y,z], entails(tensor(lpop(x,y),lpop(y,z)),
#                                    lpop(x,z))))) # internal composition rule
```

The *modus ponens* rule is

$$\frac{}{X \otimes (X \multimap Y) \vdash Y}\ (ev)$$

One way to interpret rule *(ev)* is by taking tensor $(\otimes)$ to mean linear conjunction and lollipop $(\multimap)$ to mean linear implication. Therefore, if we have X linear-and X linear-implies Y, we have Y. To verify rule *(ev)* we model it using the same technique as for rule (i) above, i.e. by adding its negation to the set of given rules and checking the satisfiability of the extended set of rules. Since the result is `'unsat'`, we conclude that rule *(ev)* holds in MILL.

Next we verify a more complex rule, called *internal composition* in [4, p.41]:

$$(X \multimap Y) \otimes (Y \multimap Z) \vdash (X \multimap Z)$$

Informally, taking $\otimes$ for linear-and and $\multimap$ for linear-implication, the internal composition rule says that we can compose chained linear-implications. To verify internal composition in MILL, we add the internal composition rule, negated. We also comment out the modus ponens rule so it won't interfere with the satisfiability check. Running the satisfiability check again, we obtain `'unsat'`, meaning that the internal composition rule holds in MILL.

Two important operators in linear logic are dual and par, denoted here by $\star$ and $\parr$, respectively, following [16]. Rules *(i)* through *(c)* above do not include rules for dual and par, but we can construct dual and par by new rules as follows:

$$\text{Dual:} \qquad X^\star = X \multimap I$$
$$\text{Par:} \qquad X \parr Y = X^\star \multimap Y$$

Here I is a constant which is the unit of tensor; its properties are specified in rules *(l)* and *(r)*. These rules are implemented as follows:

```
## Given rules for dual and par
ll.add(ForAll([x], dual(x) == lpop(x, I))) # dual
ll.add(ForAll([x,y], par(x, y) == lpop(dual(x), y))) # par
```

Now we can verify a few properties of dual and par in MILL by adding the respective negated rules to the solver, as shown in the code fragment below. We emphasize that these properties are verified one at a time by un-commenting its corresponding line while commenting all other properties.

```
## Proving properties of dual and par (all "unsat")
# ll.add(Not(ForAll([x], entails(x, dual(dual(x)))))) # x |- x dual dual
# ll.add(Not(ForAll([x,y], entails(par(dual(x), y), lpop(x, y)))))
#     # X dual par Y |- X -o Y
# ll.add(Not(ForAll([x,y],
#     entails(par(x, y), dual(tensor(dual(x), dual(y)))))))
#     # x par y |- - (- x tensor - y)
#
## out of memory
# ll.add(Not(ForAll([x], entails(dual(dual(x)), x)))) # x dual dual |- x
# ll.add(Not(ForAll([x,y], entails(lpop(x, y), par(dual(x), y)))))
#     # X -o Y |- X dual par Y
# ll.add(Not(ForAll([x,y],
#     entails(dual(tensor(dual(x), dual(y))), par(x, y)))))
#     # - (- x tensor - y) |- x par y
```

First we want to know to what extent dual is an involution in MILL, so we verify the rules $X^{\star\star} \vdash X$ and $X \vdash X^{\star\star}$ by running the solver check method. The solver yields `'unsat'` for the second rule, proving that the rule holds in MILL. The solver runs out of memory for the first rule. Details about the hardware and software used to execute the prover will be given later in the paper.

The next two rules to be checked express relationships between par ($\parr$) and linear-implies ($\multimap$):

$$X^\star \parr Y \vdash X \multimap Y \qquad \text{and} \qquad X \multimap Y \vdash X^\star \parr Y$$

Adding, one at a time, the two negated rules, the solver yielded `'unsat'` for the first rule and ran out of memory for the second rule. This means that the first rule holds in MILL and a counterexample could not be easily found in MILL for the second rule.

Now we verify relationships between par ($\parr$) and tensor ($\otimes$):

$$X \parr Y \vdash (X^\star \otimes Y^\star)^\star \qquad \text{and} \qquad (X^\star \otimes Y^\star)^\star \vdash X \parr Y$$

The first rule holds (the solver yields `'unsat'`), and for the second rule the solver runs out of memory, as above.

Finally, we prove that I is self-dual. We also check to what extent I is a neutral element for the par ($\parr$) operator.

$$I^\star \vdash I \qquad \text{and} \qquad I \vdash I^\star$$

$$X \parr I \vdash X \qquad \text{and} \qquad X \vdash X \parr I$$

The respective negated rules are implemented as shown in the code below. Three of the four rules hold (the solver yields `'unsat'`). For the rule $X \parr I \vdash X$, the solver runs out of memory, and we cannot prove anything.

```
## Proving properties of dual and par (all "unsat")
# ll.add(Not(entails(dual(I),I)))
# ll.add(Not(entails(I,dual(I))))
# ll.add(Not(ForAll([x],entails(x,par(x,I)))))
#
## out of memory
# ll.add(Not(ForAll([x],entails(par(x,I),x))))
```

Recalling the intuitionistic nature of this logic, some of the propositions that we could not prove or disprove by Z3 are unlikely to be consequences of the theory. It is however remarkable that it is so difficult for Z3 to construct counterexamples. We believe that some of the propositions that caused Z3 to run out of memory are actually consistent with the theory MILL.

To sum up the results of our investigations into MILL inference rules, we restate the derived inference rules by means of a Theorem. We use the ($\dashv\vdash$) symbol to denote bidirectional entailment.

Theorem 3. *The following sequents are provable in MILL:*

$$\begin{aligned} X \otimes (X \multimap Y) &\vdash Y \\ (X \multimap Y) \otimes (Y \multimap Z) &\vdash (X \multimap Z) \\ X &\vdash X^{\star\star} \\ X^\star \parr Y &\vdash X \multimap Y \\ X \parr Y &\vdash (X^\star \otimes Y^\star)^\star \\ X &\vdash X \parr I \\ I^\star &\dashv\vdash I \end{aligned}$$

Finally, a few details of the software and hardware platforms we used. The experiments have been run on a high performance system with four 10-core Intel Xeon E7-4870 with 2.4 GHz clock, 30 MB cache, and 1 TB total RAM. The software is Python version 3.9.13 running in a Rocky Linux operating system, version 8.7. Successful experiments completed in a few seconds. Experiments that ran out of memory never completed.

5 Conclusions

Beyond the inference rules in MILL and MLL+Mix, our modeling exercise has led us to develop rules for handling comma (union of multisets) and entails with empty antecedent context.

The exercise has shown that several known properties of MILL and MLL+Mix can be proven, as expected. It also clarifies that MLL rules are insufficient for several other inference rules to be derived

from Mix. Several rules that we checked caused Z3 to fill up the available memory and run out of time as a result.

Following [7], further work may include using Z3 as a tactic in Lean, another theorem prover. Lean uses different technologies than Z3 and the combination of Lean and Z3 can prove the rules that caused Z3 to run out of memory.

Successful validation of the derived inference rules suggests that the proposed practical approach has the potential to be more widely applicable in other formal logic systems, as Linear Temporal Logic (see e.g. [24], [11]), Reachability Logic (e.g. [25]), IMLL (e.g. [12]) and so on.

6 Appendix

Listing 1: Python code for modeling inference rules

```
## Importing the z3 module
from z3 import *

## Declarations
ll = Solver()
F = DeclareSort('F')
entails = Function('entails', F, F, BoolSort())
tensor = Function('tensor', F, F, F)
lpop = Function('lollipop', F, F, F)
x, y, z = Consts('x y z', F)

## Given rules
ll.add(ForAll([x], entails(x, x))) # (i)
ll.add(ForAll([x,y,z], entails(tensor(x,y),z) == entails(y,lpop(x,z)))) # (c)

## Derived rules
ll.add(Not(ForAll([x,y], entails(tensor(x,lpop(x,y)),y))))

## Verification
print(ll.check())
print(ll.model())
```

Listing 2: Python code for modeling MLL+Mix inference rules

```
## Importing the z3 module
from z3 import *

## Declarations
ll = Solver()
F = DeclareSort('F')
entails = Function('entails', F, F, BoolSort())
provable = Function('provable', F, BoolSort())
comma = Function('comma', F, F, F)
par = Function('par', F, F, F)
tensor = Function('tensor', F, F, F)
dual = Function('dual', F, F)
g, d, l, a, b, bot, one = Consts('g d l a b bot one', F)

## Axioms for comma (multiset reunion): associativity and commutativity
ll.add(ForAll([a, b, g], comma(a,comma(b,g)) == comma(comma(a,b),g)))
```

```
17 ll.add(ForAll([a, b], comma(a,b) == comma(b,a)))
18
19 ## Given rules (Heijltjes and Houston page 3)
20 ll.add(ForAll([g],Implies(provable(g), provable(comma(g,bot)))))
21 ll.add(provable(one))
22 ll.add(ForAll([g, a, b],Implies(
23     provable(comma(comma(g,a),b)),
24     provable(comma(g,par(a, b))))))
25 ll.add(ForAll([g, a, d, b],Implies(
26     And(provable(comma(g,a)), provable(comma(d,b))),
27     provable(comma(g,comma(d,tensor(a, b)))))))
28 ll.add(ForAll([a],provable(comma(a,dual(a)))))
29 ll.add(ForAll([g, d, a],Implies(
30     And(provable(comma(g,a)), provable(comma(d,dual(a)))),
31     provable(comma(g,d)))))
32
33 ## Mix rule (https://www.pls-lab.org/en/Mix_rule)
34 ll.add(ForAll([g, d],Implies(And(provable(g), provable(d)),
35     provable(comma(g,d)))))
36
37 ## Derived rules (Heijltjes and Houston page 4, all "unsat")
38 # ll.add(Not(provable(comma(one,bot))))
39 # ll.add(Not(ForAll([a,b],
40 #     provable(comma(tensor(a,b), par(dual(a),dual(b)))))))
41 # ll.add(Not(ForAll([a,b,g,d,l],
42 #     Implies(
43 #         And(And(provable(comma(g,a)),provable(comma(d,b))),
44 #             provable(comma(comma(l,dual(a)),dual(b)))),
45 #         provable(comma(comma(g,d),l))))))
46
47 ## Invalid derived rules (all "sat")
48 # ll.add(Not(one == bot))
49 # ll.add(Not(provable(par(dual(bot),one))))
50 # ll.add(Not(provable(bot)))
51
52 ## Contraction (subset) rule and resulting theorem
53 # ll.add(ForAll([a, b], Implies(provable(comma(one,a)),provable(a))))
54 # ll.add(Not(provable(bot))) # unsat
55
56 ## Verification
57 print(ll.check())
58 print(ll.model())
```

Listing 3: Python code for modeling MILL inference rules

```
## Importing the z3 module
from z3 import *

## Declarations
ll = Solver()
F = DeclareSort('F')
entails = Function('entails', F, F, BoolSort())
par = Function('par', F, F, F)
tensor = Function('tensor', F, F, F)
lpop = Function('lollipop', F, F, F)
dual = Function('dual', F, F)
x, y, z, w, I = Consts('x y z w I', F)

## Given rules
ll.add(ForAll([x], entails(x, x))) # rule (i)
ll.add(ForAll([x,y,z], Implies(And(
    entails(x,y), entails(y,z)), entails(x, z)))) # rule (o)
ll.add(ForAll([w,x,y,z], Implies(And(
    entails(w,x), entails(y,z)), entails(tensor(w, y), tensor(x, z)))))
    # rule (tensor)
ll.add(ForAll([w,x,y,z],
    entails(w,tensor(tensor(x,y),z)) == entails(w, tensor(x, tensor(y, z)))))
    # rule (a)
ll.add(ForAll([x,y],
    entails(x,tensor(I,y)) == entails(x,y))) # rule (l)
ll.add(ForAll([x,y],
    entails(x,tensor(y,I)) == entails(x,y))) # rule (r)
ll.add(ForAll([w,x,y],
    entails(w,tensor(x,y)) == entails(w,tensor(y,x)))) # rule (b)
ll.add(ForAll([x,y,z],
    entails(tensor(x,y),z) == entails(y,lpop(x,z)))) # rule (c)

## Derived rules (all "unsat")
# ll.add(Not(ForAll([x,y], entails(tensor(x,lpop(x,y)),y)))) # rule (ev)
# ll.add(Not(ForAll([x,y,z], entails(tensor(lpop(x,y),lpop(y,z)),
#                                    lpop(x,z))))) # internal composition rule

## Given rules for dual and par
ll.add(ForAll([x], dual(x) == lpop(x, I))) # dual
ll.add(ForAll([x,y], par(x, y) == lpop(dual(x), y))) # par

## Proving properties of dual and par (all "unsat")
# ll.add(Not(ForAll([x], entails(x, dual(dual(x)))))) # x |- x dual dual
# ll.add(Not(ForAll([x,y], entails(par(dual(x), y), lpop(x, y)))))
#     # X dual par Y |- X -o Y
# ll.add(Not(ForAll([x,y],
#     entails(par(x, y), dual(tensor(dual(x), dual(y)))))))
#     # x par y |- - (- x tensor - y)
# ll.add(Not(entails(dual(I),I)))
# ll.add(Not(entails(I,dual(I))))
# ll.add(Not(ForAll([x],entails(x,par(x,I)))))

## out of memory
# ll.add(Not(ForAll([x], entails(dual(dual(x)), x)))) # x dual dual |- x
# ll.add(Not(ForAll([x,y], entails(lpop(x, y), par(dual(x), y)))))
```

```
#     # X -o Y |- X dual par Y
# ll.add(Not(ForAll([x,y],
#     entails(dual(tensor(dual(x), dual(y))), par(x, y)))))
#    # - (- x tensor - y) |- x par y
# ll.add(Not(ForAll([x],entails(par(x,I),x))))
# ll.add(Not(ForAll([x,y,z],
#     entails(par(tensor(x,y),z), tensor(par(x,z),par(y,z))))))
#     # distributivity par tensor
# ll.add(Not(ForAll([x,y,z],
#     entails(tensor(par(x,z),par(y,z)), par(tensor(x,y),z)))))
#     # distributivity par tensor

## Verification
print(ll.check())
print(ll.model())
```

References

[1] Samson Abramsky & Radha Jagadeesan (1994): *Games and Full Completeness for Multiplicative Linear Logic*. *The Journal of Symbolic Logic* 59(2), pp. 543–574, doi:10.2307/2275407.

[2] Federico Aschieri & Francesco A. Genco (2020): *Par means parallel: multiplicative linear logic proofs as concurrent functional programs*. *Proceedings of the ACM on Programming Languages (POPL)*, pp. 1–28, doi:10.1145/3371086. Published 20 December 2019.

[3] Jeremy Avigad, Leonardo de Moura & Soonho Kong: *Theorem Proving in Lean*. Available at `https://leanprover.github.io/theorem_proving_in_lean/`. Web Portal, retrieved 2023-07-25.

[4] John C. Baez & Mike Stay (2010): *Physics, Topology, Logic and Computation: A Rosetta Stone*. In: *New Structures for Physics*, Springer Berlin Heidelberg, pp. 95–172, doi:10.1007/978-3-642-12821-9_2.

[5] Nick Benton, Gavin Bierman, Valeria de Paiva & Martin Hyland (1992): *Term assignments for intuitionistic linear logic*. Technical Report, Technical report 262, Cambridge.

[6] Florian Chudigiewitsch (2021): *Computational Complexity of Deciding Provability in Linear Logic and its Fragments*. *CoRR*, doi:10.48550/arXiv.2110.00562.

[7] Leonardo De Moura (2017): *From Z3 to Lean, Efficient Verification*. Available at `https://gateway.newton.ac.uk/sites/default/files/asset/doc/1707/from_z3_to_lean.pdf`. Turing Gateway to Mathematics.

[8] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, p. 337–340, doi:10.1007/978-3-540-78800-3_24.

[9] Roberto Di Cosmo & Vincent Danos (1992): *The linear logic primer*. published online. Available at `https://www.dicosmo.org/CourseNotes/LinLog/IntroductionLinearLogic.pdf`.

[10] Roberto Di Cosmo & Dale Miller (2006): *Linear Logic*. Stanford Encyclopedia of Philosophy. Available at `https://plato.stanford.edu/archives/sum2019/entries/logic-linear/`.

[11] Cătălin Dima (2011): *Non-axiomatizability for the linear temporal logic of knowledge with concrete observability*. *Journal of Logic and Computation* 21(6), pp. 939–958, doi:10.1093/logcom/exq031. Published 24 August 2010.

[12] Alexander V Gheorghiu, Tao Gu & David J Pym (2023): *Proof-theoretic Semantics for Intuitionistic Multiplicative Linear Logic*. *arXiv preprint*, doi:10.48550/arXiv.2306.05106.

[13] Jean-Yves Girard (1987): *Linear Logic*. *Theor. Comput. Sci.* 50(1), p. 1–102, doi:10.1016/0304-3975(87)90045-4.

[14] Jean-Yves Girard, Yves Lafont & Paul Taylor (1989): *Proofs and Types*. Cambridge Press. Available at `https://dl.acm.org/doi/10.5555/64805`.

[15] Masahito Hasegawa (1999): *Logical predicates for intuitionistic linear type theories*. In: *Proceedings of TLCA'99, 4th International Conference on Typed Lambda Calculi and Applications*, LNCS 1581, Springer-Verlag, Berlin, Heidelberg, p. 198–212, doi:10.1007/3-540-48959-2_15.

[16] Willem Heijltjes & Robin Houston (2016): *Proof equivalence in MLL is PSPACE-complete*. *Logical Methods in Computer Science* Volume 12, Issue 1, doi:10.2168/lmcs-12(1:2)2016.

[17] Yves Lafont (1993): *Introduction to Linear Logic*. Lecture notes from TEMPUS Summer School on Algebraic and Categorical Methods in Computer Science, Brno, Czech Republic.

[18] K. L. McMillan (2011): *Interpolants from Z3 proofs*. In: *2011 Formal Methods in Computer-Aided Design (FMCAD)*, Austin, TX, USA, pp. 19–27. Available at `https://dl.acm.org/doi/10.5555/2157654.2157661`.

[19] Paul-André Melliès (2009): *Categorical semantics of linear logic*. *Panoramas et syntheses* 27, pp. 15–215. Available at `https://www.irif.fr/~mellies/mpri/mpri-ens/biblio/categorical-semantics-of-linear-logic.pdf`.

[20] Paul-André Melliès (2022): *A Functorial Excursion Between Algebraic Geometry and Linear Logic*. In: *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '22)*, ACM, New York, pp. 1–13, doi:10.1145/3531130.3532488.

[21] Dale Miller (2004): *Overview of linear logic programming*. In Thomas Ehrhard, Jean-Yves Girard, Paul Ruet & Philip Scott, editors: *Linear Logic in Computer Science*, *London Mathematical Society Lecture Notes* 316, Cambridge University Press, pp. 119–150, doi:10.1017/CBO9780511550850.004.

[22] M Pagani, P Selinger & B Valiron (2014): *Applying quantitative semantics to higher-order quantum computing*. In: *Proceedings of the 41st ACM SIGPLAN*, p. 647–658, doi:10.1145/2594291.2594336.

[23] Daniel Patterson (2023): *Logic & Computation*. Course at Northeastern University. Available at `https://course.ccs.neu.edu/cs2800sp23/l31.html`.

[24] Grigore Roșu (2018): *Finite-trace linear temporal logic: Coinductive completeness*. *Formal methods in system design* 53, pp. 138–163, doi:10.1007/s10703-018-0321-3. Published 26 June 2018.

[25] Grigore Roșu, Andrei Ștefănescu, Ștefan Ciobacă & Brandon Moore (2012): *Reachability Logic*. *IDEALS*. Available at `https://www.ideals.illinois.edu/items/33225`.

[26] Andrea Schalk: *What is a categorical model for Linear Logic?* Available at `http://www.cs.man.ac.uk/~schalk/notes/llmodel.pdf`. Lecture Notes, Department of Computer Science, The University of Manchester.

[27] A. S. Troelstra & H. Schwichtenberg (1996): *Basic Proof Theory*. Cambridge University Press, doi:10.1017/CBO9781139168717.

[28] A.S. Troelstra (1992): *Lectures on Linear Logic*. CSLI (Center for the Study of Language and Information) Lecture Notes No. 29.

q-Overlaps in the Random Exact Cover Problem

Gabriel Istrate
University of Bucharest
Str. Academiei 14, Sector 6, 011014, *
Bucharest, Romania
gabrielistrate@acm.org

Romeo Negrea
Department of Mathematics,
Universitatea Politehnica din Timişoara,
Victoriei 2, 300006, Timişoara, Romania
romeo.negrea@mat.upt.ro

We prove lower and upper bounds for the threshold of the following decision problem: given $q \in (0,1)$ and $c > 0$ what is the probability that a random instance of the k-Exact Cover problem [10] has two solutions of overlap $qn \pm o(n)$?

These results are motivated by the *one-step replica symmetry breaking* approach of Statistical Physics, and the hope of using an approach based on that of [13] to prove that for some values of the order parameter the overlap distribution of k-Exact Cover has discontinuous support.

Keywords: exact cover, overlap, probabilistic method.

1 Introduction

The study of *phase transitions in Combinatorial Optimization problems* [17], [3] (see also [4, 7, 5, 15]) has recently motivated (and brought to attention) the geometric structure of the solution space of a combinatorial problem. Methods such as the *cavity method* and assumptions such as *replica symmetry* and *one step replica symmetry breaking* make significant predictions on the geometry of solution space that are a source of inspiration (and a challenge) for rigorous work.

A remarkable advance in this area is due to Mézard et al. [13]. This paper has provided rigorous evidence that for the random k-satisfiability problem (with sufficiently large k) the intuitions concerning the geometry of the solution space provided by the 1-RSB approach are correct. The evidence is based the support of the overlap distribution, shown to be discontinuous via a study of threshold properties for the q-overlap versions of k-SAT.

In this paper we follow an approach based on the same idea, studying the overlap distribution of a different optimization problem, the *random k-Exact Cover* problem. The phase transition in this problem has been studied in [9]. Zdeborová et al. [18],[12] have applied nonrigorous methods from Statistical Physics (the cavity approach) and have suggested that the *1-step Replica Symmetry Breaking* assumption is valid. This motivates us to study the problem q-overlap k-Exact Cover (defined below), and prove lower and upper bounds on its satisfiability threshold.

Our ultimate goal would be to show that for a certain range of the order parameter the k-Exact problem has a discontinuous overlap distribution. However, in this paper we cannot accomplish this goal, as the upper and lower bounds provided are too crude to guarantee this. Still, we believe that the insights provided by our partial result may be useful towards eventually obtaining such bounds.

*Corresponding author.

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 26–40, doi:10.4204/EPTCS.389.3

2 Preliminaries

We assume knowledge of the method of modeling the trajectory algorithms on random inputs using difference/differential equations using the principle of deferred decision. This is by now a material in standard textbooks [16] and surveys [1]. We will also assume knowledge of somewhat lesser popular techniques in this area, such as the "lazy server" approach [1].

Definition 1. *Let $\mathscr{D} = \{0,1,\ldots,t-1\}$, $t \geq 2$ be a fixed set. Consider the set of all $2^{t^k}-1$ potential nonempty binary constraints on k variables $X_1,\ldots,X_k$. We fix a set of constraints $\mathscr{C}$ and define the random model CSP($\mathscr{C}$). A random formula from $CSP_{n,m}(\mathscr{C})$ is specified by the following procedure: (i) n is the number of variables; (ii) we generate uniformly at random,* ***with replacement****, m clauses from all the instantiations of constraints in $\mathscr{C}$ on the n variables.*

When all constraints in $\mathscr{C}$ are boolean, we write SAT($\mathscr{C}$) instead of CSP($\mathscr{C}$).

The particular (CSP) problem we are dealing with in this paper is:

Definition 2. *An instance Φ of the k-*Exact Cover *is specified by a set of* boolean variables $V = \{x_1,\ldots,x_n\}$ *and a family of $m \geq 1$ subsets of size k (called* clauses*) of V. Instance Φ is satisfiable if there is a truth assignment A of variables in V that makes* exactly one *variable in each clause evaluate to TRUE.*

Definition 3. *The* Hamming distance *between two truth assignments A and B, on n variables is $d_{A,B} = \frac{n}{2} - \frac{1}{2}\sum_{i=1}^{n} A(x_i)B(x_i)$. The overlap of truth assignments A and B is the fraction of variables on which the two assignments coincide, that is*

$$overlap(A,B) = \frac{\{i | A(x_i) = B(x_i)\}}{n}.$$

Definition 4. *A set of constraints $\mathscr{C}$ is* interesting *if there exist constraints $C_0, C_1 \in \mathscr{C}$ with $C_0(\overline{0}) = C_1(\overline{1}) = 0$, where $\overline{0},\overline{1}$ are the "all zeros" ("all ones") assignments. Constraint C_2 is* an implicate of C_1 *iff every satisfying assignment for C_1 satisfies C_2. A boolean constraint C* strongly depends on a literal *if it has an unit clause as an implicate. A boolean constraint C* strongly depends on a 2-XOR relation *if $\exists i,j \in \{1,\ldots,k\}$ such that constraint "$x_i \neq x_j$" is an implicate of C.*

In the following definition $\varepsilon(n)$ is a function whose exact expression is unimportant (in that we get the same results), as long as $n^{1/2} = o(\varepsilon(n))$, $\varepsilon(n) = o(n)$:

Definition 5. *Let $\mathscr{D} = \{0,1,\ldots,t-1\}$, $t \geq 2$ be a fixed set. Let q be a real number in the range [0,1]. The problem q-overlap-CSP($\mathscr{C}$) is the decision problem specified as follows: (i) The input is an instance Φ of $CSP_{n,p}(\mathscr{C})$; (ii) The decision problem is whether Φ has two satisfying assignments A,B such that $overlap(A,B) \in [q-\varepsilon(n)n^{-1}, q+\varepsilon(n)n^{-1}]$. The random model for q-overlap-CSP($\mathscr{C}$) is simply the one for $CSP_{n,m}(\mathscr{C})$.*

This definition particularizes to our problem as follows:

Definition 6. *Let $q \in (0,1)$. The q-overlap k-Exact Cover is a decision problem specified as follows:*
INPUT: *an instance F of k-Exact Cover with n variables.*
DECIDE: *whether F has two assignments A and B such that*

$$overlap(A,B) \in [q-\varepsilon(n)n^{-1}, q+\varepsilon(n)n^{-1}]. \quad (1)$$

We refer to a pair (A,B) as in equation (1) as satisfying assignments of overlap approximately q.

If A,B are two satisfying assignments and $i,j \in \{0,1\}$ we will use notation $A = i, B = j$ ($A = B = i$, when $i = j$) as a shorthand for $\{x : A(x) = i, B(x) = j\}$.

Definition 7. *Let $l \geq 1$ be an integer and let A, B be two satisfying assignments of an instance Φ of k Exact Cover. Pair (A,B) is called l-*connected *if there exists a sequence of satisfying assignments $A_0, A_1, \ldots A_l$, $A_0 = A$, $A_l = B$, A_i and A_{i+1} are at Hamming distance at most l.*

Definition 8. *For $k \geq 3$, $q \in (0,1)$ define*

$$q_k = \frac{\sqrt{(k-1)(k-2)}}{2+\sqrt{(k-1)(k-2)}}, \tag{2}$$

and

$$\lambda_{q,k} := \begin{cases} \frac{(k-1)q+\sqrt{(k-1)^2q^2+k(k-2)(k-1)(1-q)^2}}{2k} & \text{if } q \in (0,q_k), \\ q & \text{otherwise.} \end{cases} \tag{3}$$

Note that for $q < q_k$ the expression for $\lambda_{q,k}$ is the unique positive root of equation

$$\frac{k-2}{x} + \frac{(q-2x)}{(k-1)(\frac{1-q}{2})^2 + x(q-x)} = 0, \tag{4}$$

and is strictly less than q. Also, $\lambda_{q,k} > q/2$, since, by (3), $\lambda_{q,k} > \frac{(k-1)q}{k} > q/2$.

Definition 9. *For $k \geq 3$, $q \in (0,1)$ define $F_{k,q} : (q/2, \lambda_{q,k}) \to (0,\infty)$ by*

$$F_{k,q}(x) = \frac{\ln(\frac{x}{q-x})}{\frac{k-2}{x} + \frac{(q-2x)}{(k-1)(\frac{1-q}{2})^2+x(q-x)}} \tag{5}$$

Note that $F_{k,q}$ is well defined, monotonically increasing (the numerator is increasing, each term in the denominator is decreasing), and that $\lim_{x\to q/2} F_{k,q}(x) = 0$, $\lim_{x\to\lambda_{q,k}} F_{k,q}(x) = \infty$. Thus function $F_{k,q}$ is a bijection. Denote by $G_{k,q}(x)$ its inverse.

3 Results

We first remark that

Lemma 1. *For every $k \geq 3$ and $q \in (0,1)$ the problem q-overlap k-Exact Cover has a sharp threshold.*

Proof. The claim is a simple application of the main result in [6]. Indeed, in [6] we studied the existence of a sharp threshold for q-overlap versions of random constraint satisfaction problems. Previously in [5, 2], a characterization of CSP with a sharp threshold was given:

Proposition 1. *Consider a generalized satisfiability problem SAT($\mathscr{C}$) with $\mathscr{C}$ interesting. (i) If some constraint in $\mathscr{C}$ strongly depends on one literal then SAT($\mathscr{C}$) has a coarse threshold; (ii) If some constraint in $\mathscr{C}$ strongly depends on a 2XOR-relation then SAT($\mathscr{C}$) has a coarse threshold; (iii) In all other cases SAT($\mathscr{C}$) has a sharp threshold.*

The folowing result (Theorem 8 in [6]) shows that under the same conditions as those in [5] the q-overlap versions also have a sharp threshold:

Proposition 2. *Consider a generalized satisfiability problem SAT($\mathscr{C}$) such that (i) $\mathscr{C}$ is interesting (ii) No constraint in $\mathscr{C}$ strongly depends on a literal; (iii) No constraint in $\mathscr{C}$ strongly depends on a 2XOR-relation. Then for all values $q \in (0,1]$ the problem q-overlap-SAT($\mathscr{C}$) has a sharp threshold.*

The conditions in Proposition 2 apply to the k-Exact Cover problem, which can be modeled as a CSP with a single k-ary constraint $C_k(x_1,x_2,\ldots,x_k)$ which requires that exactly one of $x_1,x_2,\ldots,x_k$ be true. This is because constraint C_k is interesting, does not strongly depend on a literal and, for $k \geq 3$, does not strongly depend on a 2-XOR relation. □

Our main result gives lower and upper bounds on the location of this threshold:

Theorem 1. *Let $k \geq 3$ and let $r_{up}(q,k)$ be the smallest $r_* > 0$ such that $\forall r > r_*$*

$$\begin{aligned} & r\ln(P_k(G_{k,q}(r),(1-q)/2,(1-q)/2,q-G_{k,q}(r))) - G_{k,q}(r)\ln(G_{k,q}(r)) - \\ - \quad & (q-G_{k,q}(r))\ln(q-G_{k,q}(r)) - (1-q)\ln((1-q)/2) \leq 0. \end{aligned}$$

Also let

$$r_{lb}(q) = \begin{cases} \frac{1}{6}\left[\frac{1}{(1-q)^2}-1\right] & \text{for } q < 1-\frac{1}{\sqrt{2}}, \\ \frac{1}{6} & \text{otherwise.} \end{cases} \tag{6}$$

Then:

(a). For $r > r_{up}(q,k)$ a random instance of q-overlap k-Exact Cover with n variables and $m = rn$ clauses has, with probability $1-o(1)$, no satisfying assignments of overlap approximately q.

(b). For $0 < r < r_{lb}(q)$ a random instance of q-overlap 3-Exact Cover with n variables and $m = rn$ clauses has, with probability $1-o(1)$, two satisfying assignments of overlap approximately q.

Given the non-explicit nature of $r_{up}(q)$, the only way to interpret the lower and upper bounds given in Theorem 1 is via symbolic and numeric manipulations of the quantities in the equation(s) defining $r_{up}(q)$. A Mathematica notebook to this goal is provided as [8]. The conclusion of such an analysis is that the bounds in Theorem 1 are too crude to imply the existence of a discontinuity in overlap in the k-Exact Cover problem.

4 Proof of the upper bound (Theorem 1 (a))

Let Φ be a random instance of k-Exact Cover. Our proof relies on the following fundamental observation:

Lemma 2. *Let A,B be two satisfying assignments, and let C be a clause of length k in Φ. Denote by c_0,c_1,c_2,c_3 the number of variables of C in the sets $A=B=0$, $A=0,B=1$, $A=1,B=0$, $A=B=1$ respectively. Clause C is satisfied by both A and B if and only if*

$$\begin{cases} c_0 = k-2, \quad c_1 = c_2 = 1, \quad c_3 = 0 \\ \text{or} \\ c_0 = k-1, \quad c_1 = c_2 = 0, \quad c_3 = 1 \end{cases} \tag{7}$$

Proof. The conditions that both A and B satisfy C are written as

$$\begin{cases} c_0 + c_1 = k-1, \quad c_2 + c_3 = 1 \\ c_0 + c_2 = k-1, \quad c_1 + c_3 = 1, \end{cases} \tag{8}$$

a system whose solutions are those from equation (7). □

An immediate consequence of Lemma 2 is that the probability that a pair of assignments satisfies a random instance of k-EC depends only on numbers c_0,c_1,c_2,c_3:

Lemma 3. *Let c_0, c_1, c_2, c_3 be nonnegative numbers. Then*

$$Pr[A,B \models \Phi \mid |A=B=0| = c_0, \dots |A=B=1| = c_3] = P^*(c_0,c_1,c_2,c_3)^{rn},$$

where

$$P^*(a,b,c,d) = \frac{\binom{a}{k-2}\binom{b}{1}\binom{c}{1} + \binom{a}{k-1}\binom{d}{1}}{\binom{n}{k}} = \frac{\binom{a}{k-2}}{\binom{n}{k}}\left[bc + \frac{(a-k+2)}{(k-1)}d\right] \tag{9}$$

Proof. We will prove that the probability that A, B satisfy a particular clause of Φ is $P^*(c_0,c_1,c_2,c_3)$. The result follows since the formula Φ is obtained by sampling independently, with replacement, rn clauses.

Indeed, the total number of clauses is $\binom{n}{k}$. By Lemma 2, the number of clauses satisfied by both A and B is $\binom{a}{k-2}\cdot\binom{b}{1}\cdot\binom{c}{1}+$. The first term represents the number of clauses with $k-2$ variables in the set $A=B=0$, one in the set $A=0, B=1$ and one in the set $A=1, B=0$ (so that exactly one literal of C is true in both A and B). The second term counts the second type of favorable clauses. □

We will use Lemma 3 to derive an upper bound via the first moment method.

Indeed, let $Z = Z(q,F)$ be a random variable defined as

$$Z(q,F) = \sum_{A,B} \delta[|d_{A,B} - nq| \le e(n)] \cdot \mathbf{1}_{\mathscr{S}(\mathbf{F})}(\mathbf{A}) \cdot \mathbf{1}_{\mathscr{S}(\mathbf{F})}(\mathbf{B}). \tag{10}$$

where $F = F_k(n, rn)$ is a random formula on n variables over $m = rn$ clauses of size k, the set $\mathscr{S}(F)$ is the set of the EC-assignments to this formula.

Then:

$$E[Z(q,F)] = \sum_{A,B} \delta[|d_{A,B} - nq| \le e(n)] \cdot Pr[A,B \models F]. \tag{11}$$

For fixed values a,b,c,d there are $\binom{n}{a,b,c,d} = \frac{n!}{a!\cdot b!\cdot c!\cdot d!}$ pairs of assignments of type (a,b,c,d). If we denote $\lambda \stackrel{not}{=} a+d = nq \pm \varepsilon(n)$ and $\mu \stackrel{not}{=} b+c = n-\lambda$ then the system

$$\begin{cases} a+d = \lambda \\ b+c = n-\lambda \end{cases}$$

has at most $(\lambda+1)(n-\lambda+1)$ solutions in the set of nonnegative integers. Therefore, the number of quadruples (a,b,c,d) in the sum $E[Z]$ is at most

$$\sum_{\lambda=nq-\varepsilon(n)}^{nq+\varepsilon(n)} (\lambda+1)(n-\lambda+1) = \frac{1}{3}(1+2\varepsilon(n))(3-\varepsilon(n)-\varepsilon(n)^2+3n+3n^2q-3n^2q^2) \stackrel{def}{=} M.$$

So

$$P[Z>0] \le E[Z] \le M \cdot \max_{(a,b,c,d)} \binom{n}{a,b,c,d} \cdot P^*(a,b,c,d)^{rn} \tag{12}$$

We will compute the maximum on the right-hand side and derive conditions for which this right-hand side tends (as $n \to \infty$) to zero.

Indeed, denote $\alpha=\frac{a}{n},\beta=\frac{b}{n},\gamma=\frac{c}{n},\delta=\frac{d}{n}$. Applying Stirling's formula $n!=(1+o(1))\cdot\left(\frac{n}{e}\right)^n\sqrt{2\pi n}$, and also noting that

$$P^*(a,b,c,d)\leq(1+\frac{O(1)}{n})\cdot P_k(\alpha,\beta,\gamma,\delta),$$

with

$$P_k(\alpha,\beta,\gamma,\delta)=\alpha^{k-2}k(k-1)(\beta\gamma+\frac{\alpha\delta}{k-1}) \tag{13}$$

we get

$$P[Z>0]\leq M\cdot\theta(1)\cdot\max_{(\alpha,\beta,\gamma,\delta)}\cdot\Big[\Big(\frac{1}{\alpha^\alpha\beta^\beta\gamma^\gamma\delta^\delta}\Big)\cdot P(\alpha,\beta,\gamma,\delta)^r\Big]^n$$

Define

$$g_r(\alpha,\beta,\gamma,\delta)=\frac{P_k(\alpha,\beta,\gamma,\delta)^r}{\alpha^\alpha\beta^\beta\gamma^\gamma\delta^\delta}$$

Lemma 4. *For any $r>0$ we have*

$$\max\Big\{g_r(\alpha,\beta,\gamma,\delta):\alpha+\delta=q,\beta+\gamma=1-q,\alpha,\beta,\gamma,\delta\geq 0\Big\}=g_r(\alpha_{*,r},\beta_{*,r},\gamma_{*,r},\delta_{*,r}),$$

with

$$\begin{cases}\alpha_{*,r}=G_{k,q}(r),\\ \beta_{*,r}=\gamma_{*,r}=(1-q)/2,\\ \delta_{*,r}=q-G_{k,q}(r).\end{cases} \tag{14}$$

Proof.

First, it is easy to see that

$$g_r(\alpha,\beta,\gamma,\delta)\leq g_r\Big(\alpha,\beta_{*,r},\gamma_{*,r},\delta\Big). \tag{15}$$

Indeed, function $x\ln(x)$ is convex, having the second derivative positive, and e^x is increasing so, by Jensen's inequality,

$$\beta^\beta\gamma^\gamma=e^{\beta\ln(\beta)+\gamma\ln(\gamma)}\geq e^{(\beta+\gamma)\ln(\frac{\beta+\gamma}{2})}=\Big(\frac{\beta+\gamma}{2}\Big)^{\beta+\gamma}=\beta_*^{\beta_{*,r}}\gamma_{*,r}^{\gamma_{*,r}}.$$

On the other hand since $\beta\gamma\leq\left(\frac{\beta+\gamma}{2}\right)^2=\beta_{*,r}\gamma_{*,r}$, we have $P(\alpha,\beta,\gamma,\delta)\leq P\Big(\alpha,\beta_{*,r},\gamma_{*,r},\delta\Big)$ and equation (15) follows.

Also

$$g_r\Big(\alpha,\beta_{*,r},\gamma_{*,r},\delta\Big)\leq g_r\Big(\alpha_{*,r},\beta_{*,r},\gamma_{*,r},\delta_{*,r}\Big) \tag{16}$$

Indeed, replacing $\delta=q-\alpha$, the expression

$$\begin{aligned}t(\alpha)&=\ln g_r(\alpha,\beta_{*,r},\gamma_{*,r},q-\alpha)=\\ &=r\ln\Big(P_k(\alpha,\beta_{*,r},\gamma_{*,r},q-\alpha)\Big)-\alpha\ln(\alpha)-(q-\alpha)\ln(q-\alpha)-\beta_{*,r}\ln(\beta_{*,r})-\gamma_{*,r}\ln(\gamma_{*,r})\end{aligned}$$

is a function of α whose derivative is

$$\begin{aligned}t'(\alpha)&=r\frac{P_k'(\alpha,\beta_{*,r},\gamma_{*,r},q-\alpha)}{P_k(\alpha,\beta_{*r},\gamma_{*r},q-\alpha)}-\ln(\alpha)-1+\ln(q-\alpha)+1=\\ &=r[\frac{k-2}{\alpha}+\frac{q-2\alpha}{\left(\frac{1-q}{2}\right)^2+\alpha(q-\alpha)}]+\ln(\frac{q-\alpha}{\alpha}).\end{aligned}$$

so $t(\alpha)$ has a maximum on $[0,q]$ at $\alpha_{*,r}$ which is a solution of equation

$$\frac{r(k-2)}{\alpha}+\frac{r(q-2\alpha)}{(k-1)\left(\frac{1-q}{2}\right)^2+\alpha(q-\alpha)}=\ln(\frac{\alpha}{q-\alpha}), \tag{17}$$

or $F_{k,q}(\alpha_{*,r})=r$. In other words $\alpha_{*,r}=G_{k,q}(r)$ and $\delta_{*,r}=q-G_{k,q}(r)$. □

Formula (14) implies that $P[Z>0]\stackrel{n\to\infty}{\to}0$ as long as $t(\alpha_{*,r})<0$. The critical value $r_{up}(q,k)$ is therefore given by equation $t(\alpha_{*,r})=0$, that is

$$\begin{aligned} & r\ln(P_k(G_{k,q}(r),(1-q)/2,(1-q)/2,q-G_{k,q}(r)))-G_{k,q}(r)\ln(G_{k,q}(r))- \\ - \quad & (q-G_{k,q}(r))\ln(q-G_{k,q}(r))-(1-q)\ln((1-q)/2)=0. \end{aligned} \tag{18}$$

□

For $k=3$, denoting (for simplicity) $\alpha=G_{3,q}(r)$, we have $P_3(\alpha,\beta,\gamma,\delta)=6\alpha(\beta\gamma+\frac{\alpha\delta}{2})$, so the equation (18) becomes

$$r\ln(6\alpha[(\frac{1-q}{2})^2+\frac{\alpha(q-\alpha)}{2}])-\alpha\ln(\alpha)-(q-\alpha)\ln(q-\alpha)=(1-q)\ln((1-q)/2)$$

while equation (17) becomes

$$r[1+\frac{\alpha(q-2\alpha)}{2\left(\frac{1-q}{2}\right)^2+\alpha(q-\alpha)}]=\alpha\ln(\frac{\alpha}{q-\alpha}),$$

Attempting a substitution of the type $\frac{\alpha}{q-\alpha}=t$ in this last equation seems to turn the function F_3 into a generalized version of the Lambert function. However, this generalization seems to be different from the versions already existing in the literature [14], so this attempt does not seem fruitful.

We refer again to the Mathematica notebook provided as [8]. In particular let us remark that the maximum value of $r_{up}(q)$ is reasonably close to upper bound the threshold for 3-Exact Cover derived using the first-moment method in [11].

5 Proof of the lower bound (Theorem 1 (b))

We will use a constructive method. Just as in [10], we will derive a lower bound from the probabilistic analysis of an algorithm. However, the algorithm **will not** be the one from [10]. Instead, we will investigate (a variant of) the algorithm LARGEST-CLAUSE in Figure 1.

Intuitively, the reason we prefer the algorithm LARGEST-CLAUSE to the one from [10] is simple: unlike [10], our goal is not to simply solve an instance of k-EXACT COVER, but to create **two satisfying assignments** of controlled overlap. We we would like to accomplish that via an algorithm that iteratively assigns values to variables and is left (at some point) with solving a 2-XOR SAT formula. Our aim is to keep the number of set variables to a minimum, in order to create satisfying assignments with as large an overlap as possible. But that means that one must "destroy" all clauses of length different from two as fast as possible. Instead, the algorithm in [10] is focused on killing clauses of length 2.

The algorithm may seem incompletely specified, as its performance depends on function $f(n)$. As it will turn out, the precise specification of function $f(n)$ in the algorithm LARGEST-CLAUSE will

Algorithm **LargestClause**

INPUT: a formula Φ

if (Φ contains a unit clause)
  choose a random unit clause l
  set l to TRUE
    if this creates a contradiction FAIL
    else call the algorithm recursively
else if (Φ contains a clause of length ≥ 3)
    choose a random clause C of maximal length
    choose a random literal l of C
    set l to zero and simplify the formula
  else
      create a graph G containing an edge (x,y)
      for any clause $x \oplus y$ in Φ;
      if (G is not bipartite) OR (some connected component has size $\geq f(n)$)
        FAIL
      else
        choose one variable in each connected component of G
        create satisfying assignments A and B
        by setting all chosen variables to one (zero)
        and then propagating these values to all variables in G.
        return (A,B).

Figure 1: Algorithm LARGEST-CLAUSE

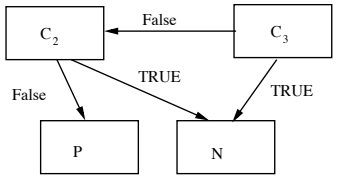


Figure 2: The dynamics of algorithm LARGEST-CLAUSE.

not matter for our purposes, as long as f is a function that grows asymptotically faster than the size of the giant component in a certain subcritical Erdős-Rényi random graph, which is with high probability $O(log(n))$.

To analyze (versions of) Algorithm LARGEST-CLAUSE, we denote by $C_i(t)$, $i \geq 2$, the number of clauses of length i that are present after t variables have been set. Also define $P(t), N_t$ to be the number of positive (negative) unit clauses present at stage t. Finally, define functions $c_1, c_2, c_3, p, n : (0,1) \to \mathbf{R}_+$ by $c_i(\alpha) = C_i(\alpha \cdot n)/n$, and similar relations for functions $p(\cdot), n(\cdot)$. We will use a standard method, *the principle of deferred decisions* to analyze algorithm LARGEST-CLAUSE. See [1] for a tutorial.

It is easy to show by induction that at any stage t, conditional on the four-tuple $(P(t), N(t), C_2(t), C_3(t))$, the remaining formula is uniform.

We divide the algorithm in two phases: in the *first phase* there exist clauses of length three. In the *second phase* only clauses of length one and two exist.

If a variable is set to TRUE then a 1-in-i clause containing that variable is turned into $i-1$ negative unit clauses. If a variable is set to FALSE then a 1-in-i clause is turned into a 1-in-$(i-1)$ clause, in particular a 1-in-2 clause is turned into a positive unit clasue. The dynamics is displayed in Figure 5.

The different dynamics of the flows in the cases when a positive (negative) literal is set makes the direct analysis of algorithm LARGEST-CLAUSE difficult. Therefore, we will instead analyze a version of the algorithm, given in Figure 3, using a "lazy-server" [1] idea. Specifically, instead of always trying to simplify the unit clauses, we will do so probabilistically (see Figure 3 for details).

Since the problems q-overlap EXACT COVER have a sharp threshold, it is enough to prove, for $r < r_{lb}(q)$ that the algorithm finds a pair of assignments of overlap r with probability $\Omega(1)$. This will be enough to conclude that with probability $1 - o(1)$ two satisfying assignments of overlap r exist.

Let $U_P(t), U_N(t), U_3(t)$ be 0/1 variables that are one exactly when choice 1 (2,3) is selected, 0 otherwise. We can write the following recurrence relations describing the dynamics of the four-tuple $(P(t), N(t), C_2(t), C_3(t))$:

$$\begin{cases} C_3(t+1) = C_3(t) - U_3(t) - \Delta_3(t), \\ C_2(t+1) = C_2(t) - \Delta_2(t) + \Delta_{3,2}(t), \\ P(t+1) = P(t) - U_P(t) - \Delta_{1,P}(t) + \Delta_{2,P}(t), \\ N(t+1) = N(t) - U_N(t) - \Delta_{1,N}(t) + \Delta_{2,N}(t) + \Delta_{3,N}(t), \end{cases} \tag{19}$$

Algorithm **LazyLargestClause**

INPUT: a formula Φ

if (at least one of the alternatives 1,2,3 below applies)
take one of the following actions with probabilities $\lambda_1(t), \lambda_2(t), \lambda_3(t)$, respectively:
1. if (Φ contains a positive unit clause)
choose a random positive unit clause l
set l to TRUE
if this creates a contradiction FAIL
else
set a random variable to TRUE
2. if (Φ contains a negative unit clause)
choose a random negative unit clause $\bar{l}$
set l to FALSE
if this creates a contradiction FAIL
else
set a random variable to FALSE
3. if (Φ contains a clause of length ≥ 3)
choose a random clause C of maximal length
choose a random literal l of C
set l to FALSE

else
run the corresponding bipartite graph construction
of algorithm LARGEST-CLAUSE.

Figure 3: The "lazy-server" version of algorithm LARGEST-CLAUSE

where

$$\begin{cases} \Delta_3(t) \stackrel{d}{=} Bin\Big(C_3(t)-U_3(t),\frac{3}{n-t}\Big). \\ \Delta_2(t)=\Delta_{2,N}(t)+\Delta_{2,P}(t) \stackrel{d}{=} Bin\Big(C_2(t),\frac{2}{n-t}\Big). \\ \Delta_{3,2}(t) \stackrel{d}{=} U_3(t)+(U_N(t)+U_3(t))\cdot Bin\Big(C_3(t)-U_3(t),\frac{3}{n-t}\Big) \\ \Delta_{3,N}(t) \stackrel{d}{=} 2U_P(t)\cdot Bin\Big(C_3(t),\frac{3}{n-t}\Big) \\ \Delta_{2,P}(t) \stackrel{d}{=} (U_N(t)+U_3(t))\cdot Bin\Big(C_2(t),\frac{2}{n-t}\Big) \\ \Delta_{2,N}(t) \stackrel{d}{=} U_P(t)\cdot Bin\Big(C_2(t),\frac{2}{n-t}\Big) \\ \Delta_{1,P}(t) \stackrel{d}{=} Bin\Big(P(t)-U_P(t),\frac{1}{n-t}\Big) \\ \Delta_{1,N}(t) \stackrel{d}{=} Bin\Big(N(t)-U_N(t),\frac{1}{n-t}\Big) \end{cases} \tag{20}$$

By an analysis completely similar to that of algorithm for random k-SAT (see e.g. [1]), we derive the following system of equations that describe the average trajectory path of Algorithm LAZY LARGEST-CLAUSE:

$$\begin{cases} c_3'(t) = -\lambda_3(t) - \frac{3c_3(t)}{(1-t)}. \\ c_2'(t) = -\frac{2c_2(t)}{(1-t)} + \frac{3c_3(t)}{(1-t)}\cdot(\lambda_2(t)+\lambda_3(t)), \end{cases} \tag{21}$$

with initial conditions $(c_2(0),c_3(0))=(0,r)$.

In this paper we will make the simplest choice

$$\lambda_1(t)=\lambda_2(t)=\lambda_3(t)=1/3. \tag{22}$$

Differential equations (21) describe the dynamics of algorithm LARGEST-CLAUSE only for $t\in[t_3,t_2)$, where $t_3=0$ and $t_2\in(0,1)$ is the smallest solution of equation $c_3(t)=0$.

Simple computations lead us to formulas:

$$\begin{cases} c_3(t) = (r+\frac{1}{6})(1-t)^3 - \frac{1-t}{6}, \\ c_2(t) = \frac{(1-t)^2}{3} - \frac{(1-t)}{3} + 2(r+\frac{1}{6})t(1-t)^2, \end{cases} \tag{23}$$

which describe the dynamics of algorithm LARGEST-CLAUSE in range $0\le t<t_2=1-\frac{1}{\sqrt{6r+1}}$.

The average flow into positive unit clauses is

$$\begin{aligned} F_2^P(t) &:= \frac{2}{3}\cdot\frac{2c_2(t)}{1-t}+\frac{1}{3}\cdot\frac{2\cdot 3c_3(t)}{1-t} = \\ &= \frac{4}{3}\Big[\frac{(1-t)^2}{3}-\frac{(1-t)}{3}+2(r+\frac{1}{6})t(1-t)\Big]+2(r+\frac{1}{6})(1-t)^2-\frac{1}{3}. \end{aligned}$$

$$(F_2^P)'(t)=\frac{8r(1-2t)}{3}-4(r+\frac{1}{6})(1-t)=\Big(\frac{4r}{3}-\frac{2}{3}\Big)(1-t)-\frac{8r}{3}<0,$$

so $F_2^P(t)$ has a maximum at 0, equal to $2r$. For $r<1/6$ this is less than $1/3$, so it is balanced by being given the opportunity (with probability 1/3) to consume a positive unit clause, if any.

The average flow into negative unit clauses is

$$F_2^N(t)=\frac{1}{3}\cdot\frac{2c_2(t)}{1-t}=\frac{2}{3}\cdot\Big[\frac{(1-t)}{3}-\frac{1}{3}+2(r+\frac{1}{6})t(1-t).\Big]=\frac{2t}{9}\Big[(6r+1)(1-t)-1\Big].$$

The maximum of $F_2^N(t)$ is reached at $t = \frac{3r}{6r+1}$, which is in the interval (t_3, t_2) for $r > 0$, and is equal to $\frac{2r^2}{6r+1} = \frac{r}{3}(1 - \frac{1}{6r+1})$, which is definitely less than $\frac{1}{3}$, for $r < 1/6$.

The conclusion is that for $r < 1/6$ with probability $1 - o(1)$ both flows into positive and negative unit clauses can be handled by the lazy server with choice $\lambda_1 = \lambda_2 = \lambda_3 = 1/3$ without creating contradictory clauses.

Around stage $t_2 n \pm o(n)$ clauses of length three and one run out. We are left with a system of $(c_2(t_2) + o(1))n$ 1-in-2 clauses in the remaining $\overline{n} = (1 - t_2)n$ variables. Consider graph G corresponding to these equations, where for every equation $x \oplus y = 1$ we add edge (x, y) to G.

By the uniformity lemma *G can be seen as an Erdős-Renyi random graph* $G(\overline{n}, \frac{\mu}{\overline{n}})$, with probability coefficient

$$\mu = 2c_2(t_2)/(1 - t_2) = 3F_2(t_2).$$

Our maximum computation shows that for $r \in (0, 1/6)$, $3F_2(t_2) < 1$. Thus G is a subcritical random graph, whose connected components are w.h.p. of size $O(\log n)$. With constant probability (depending only on μ), G is a bipartite graph. In this situation giving a value to an arbitrary node uniquely determines the values of variables in the connected component.

We create two assignments A and B as follows:

1. On variables x set by algorithm LARGEST-CLAUSE, $A(x) = B(x)$, equal to the value given by the algorithm.
2. On variables in graph G A and B take opposite values. This can be accomplished by giving A, B different values on a set of fixed variables, one in each connected component of G.

When graph G is bipartite A and B are satisfying assignments. When the connected components of G are of size $O(\log n)$ we can create a path from A to B consisting satisfying assignments by consecutively flipping values of variables on which A and B are different, one connected component at a time. The overlap of A and B is equal to $1 - \frac{1}{\sqrt{6r+1}}$.

It follows that for any $q \in (0, 1)$, the q-overlap Exact Cover is satisfiable w.h.p. for $q > 1 - \frac{1}{\sqrt{6r+1}}$, i.e. $\frac{1}{6r+1} > (1 - q)^2$, which can be rewritten as $r < r_{lb}(q)$. □

6 Remarks

The condition $r < 1/6$ in Theorem 1 has an easy probabilistic interpretation: it is the location of the phase transition for the random 3-uniform hypergraph [19]. In this range most connected components are small and tree-like or unicyclic, so the space of variables breaks down in independent clusters of size $O(\log n)$. Thus we should expect that all overlaps in some range $(\lambda, 1)$ are satisfied with probability $1 - o(1)$, which is exactly what happens, according to Theorem 1, for $\lambda = 1 - \frac{1}{\sqrt{2}}$.

In fact we can state more: in this regime there is a single cluster of solutions, and the bounds on the overlap we provide are in fact bounds on the diameter of this cluster.

Theorem 2. *Let $r < 1/k(k-1)$. There exists $C > 0$ such that, with probability $1 - o(1)$ (as $n \to \infty$), if Φ is a random instance of k-Exact-Cover with n variables and rn clauses, any two satisfying assignments of Φ are $C\log(n)$ connected.*

Proof. Since the formula hypergraph H of Φ is subcritical, there exists [19] $C > 0$ such that w.h.p. all connected components of H have size at most $C\log(n)$. That means that formula Φ is the decomposition of several variable-disjoint formulas $\Phi_1, \ldots, \Phi_p$. In turn, satisfying assignments for Φ are obtained by concatenating satisfying assignments for these formulas.

This argument immediately implies that any two satisfying assignments of Φ are $C\log(n)$ connected: Let A,B be two such satisfying assignments, and let A_1,B_1, (A_2,B_2), $\ldots$, (A_s,B_s) be the restrictions of A and B on the components on which they differ.

One can obtain a path from A to B as follows (where variables x such that $A(x)=B(x)$ are ommitted from representation):

$$\begin{aligned} A &= (A_1,A_2,\ldots,A_s)\to(B_1,A_2,\ldots,A_s)\to(B_1,B_2,\ldots,A_s)\to\ldots\to \\ &\to (B_1,\ldots,B_{s-1},A_s)\to(B_1,B_2,\ldots B_s)=B. \end{aligned}$$

The intermediate assignments are satisfying assignments since formulas $\Phi_1,\ldots,\Phi_p$ are disjoint. They are at distance at most $C\log(n)$ because of the upper bound on the component size of H.

□

Using the above result we obtain the following analog of the result proven in [6] for 2-SAT:

Corollary 1. *For $r<1/k(k-1)$ a random instance of k-Exact Cover has a single cluster of satisfying assignments and an overlap distribution with continuous support.*

The relative weakness of the bound $r<1/k(k-1)$ comes from our suboptimal choice of parameters $\lambda_1(t),\lambda_2(t),\lambda_3(t)$. For instance, for $k=3$ the bound $r<1/6$ comes entirely from handling positive unit clauses, while we have no problem satisfying negative ones, since the flow $F_2^N(t)$ always stays below one. This suggests that we are disproportionately often taking care of negative unit literals.

In what follows we sketch an approach for a better choice of these parameters. We were not able to explicitly calculate $\lambda_1(t),\lambda_2(t),\lambda_3(t)$, so we are unable to offer an improved analysis of the LAZY LARGEST-CLAUSE algorithm.

First, the algorithm has to be able to satisfy the positive unit flow, so

$$\lambda_1(t)\geq(\lambda_2(t)+\lambda_3(t))\cdot\frac{2c_2(t)}{1-t}.$$

Thus

$$\frac{\lambda_1(t)}{1-\lambda_1(t)}\geq\frac{2c_2(t)}{1-t}$$

in other words

$$\lambda_1(t)\geq\frac{2c_2(t)}{1-t+2c_2(t)}.$$

First, the algorithm has to be able to handle the negative unit flow, so

$$\lambda_2(t)\geq\lambda_1(t)\frac{6c_3(t)+2c_2(t)}{1-t}$$

We choose

$$\begin{cases} \lambda_1(t)=\frac{2c_2(t)+\varepsilon}{1-t+2c_2(t)} \\ \lambda_2(t)=\frac{(2c_2(t)+\varepsilon)(6c_3(t)+2c_2(t)+\varepsilon)}{(1-t)(1-t+2c_2(t))}, \\ \lambda_3(t)=1-\lambda_1(t)-\lambda_2(t)=\frac{(2c_2(t)+\varepsilon)(6c_3(t)+2c_2(t)+\varepsilon)}{(1-t)(1-t+2c_2(t))}. \end{cases} \tag{24}$$

It is an open problem if this approach can be completed to a full analysis.

7 Conclusions

The obvious question raised by this work is to improve our bounds enough to display the discontinuity of overlap distribution, a property of k-Exact Cover we believe to be true.

Note that there are obvious candidate approaches to improving our bounds: first, the lower bound could be improved by trying a rigorous version of the (heuristic) upper bound approach of Knysh et al. [11]. Or, it could be improved by finding explicit expressions for the parameters in (and explicitly analyzing) the LAZY LARGEST-CLAUSE algorithm, along the lines described in the previous section. Neither one of these two approaches looks particularly tractable, though.

As for the upper bound, an obvious candidate is the second moment method. We have attempted such an approach. The problem is that it seems to require optimizing of a function of 16 variables without enough obvious symmetries that would make the problem tractable.

8 Acknowledgments

The authors thank the anonymous referees for useful comments, suggestions and corrections.

References

[1] D. Achlioptas (2001): *Lower bounds for random 3-SAT via differential equations*. Theoretical Computer Science 265, pp. 159–185, doi:10.1016/S0304-3975(01)00159-1.

[2] N. Creignou & H. Daudé (2004): *Coarse and Sharp Thresholds for Random Generalized Satisfiability Problems*. In M. Drmota et al., editor: Mathematics and Computer Science III: Algorithms, Trees, Combinatorics and Probabilities, Birkhauser, pp. 507–517, doi:10.1007/978-3-0348-7915-6.

[3] A. Hartmann & M. Weigt (2005): *Phase transitions in combinatorial optimization problems*. Wiley-VCH, doi:10.1002/3527606734.

[4] G. Istrate (2000): *Computational Complexity and Phase Transitions*. In: Proceedings of the 15th I.E.E.E. Annual Conference on Computational Complexity (CCC'00), pp. 104–115, doi:10.1109/CCC.2000.856740.

[5] G. Istrate (2005): *Threshold properties of random boolean constraint satisfaction problems*. Discrete Applied Mathematics 153, pp. 141–152, doi:10.1016/j.dam.2005.05.010.

[6] G. Istrate (2007): *Satisfiability of Boolean Random Constraint Satisfaction Problems: Clusters and Overlaps*. Journal of Universal Computer Science 13(11), pp. 1655–1670, doi:10.3217/jucs-013-11-1655.

[7] G. Istrate, A. Percus & S. Boettcher (2005): *Spines of Random Constraint Satisfaction Problems: Definition and Connection with Computational Complexity*. Annals of Mathematics and Artificial Intelligence 44(4), pp. 353–372, doi:10.1007/s10472-005-7033-2.

[8] Gabriel Istrate & Romeo Negrea (2023): *Companion site on Github for the paper "q-Overlaps in the Random Exact Cover Problem" (this paper)*. Available at URL `https://github.com/Gabriel-Istrate/Overlap-Exact-Cover`.

[9] V. Kalapala & C. Moore (2008): *The phase transition in exact cover*. Chicago Journal of Theoretical Computer Science (5), doi:10.4086/cjtcs.2008.005.

[10] Vamsi Kalapala & Cris Moore (2005): *The Phase Transition in Exact Cover*. Technical Report cs/0508037, arXiv.org, doi:10.48550/arXiv.cs/0508037.

[11] S. Knysh, V. N. Smelyanskiy & R. D. Morris (2004): *Approximating satisfiability transition by suppressing fluctuations*. Technical Report cond-mat/0403416, arXiv.org, doi:10.48550/arXiv.cond-mat/0403416.

[12] E. Maneva, T. Meltzer, J. Raymond, A. Sportiello & L. Zdeborová (2007): *A hike in the phases of the 1-in-3 Satisfiability Problem*. In J.P. Bouchaud, M. Mézard & J. Dalibard, editors: *Lecture Notes of the Les Houches Summer School 2006*, Elsevier, pp. 491–498, doi:10.1016/S0924-8099(07)80022-9.

[13] M. Mézard, T. Mora & R. Zecchina (2005): *Clustering of solutions in the random satisfiability problem*. *Physical Review Letters* 94(197205), doi:10.1103/PhysRevLett.94.197205.

[14] István Mezõ (2022): *The Lambert W function: its generalizations and applications*. CRC Press, doi:10.1201/9781003168102.

[15] C. Moore, G. Istrate, D. Demopoulos & M. Vardi (2007): *A continuous-discontinuous second-order transition in the satisfiability of a class of Horn formulas*. *Random Structures and Algorithms* 31(2), pp. 173–185, doi:10.1002/rsa.20176.

[16] C. Moore & S. Mertens (2011): *The nature of computation*. Oxford University Press, doi:10.1093/acprof:oso/9780199233212.001.0001.

[17] A. Percus, G. Istrate & C. Moore, editors (2006): *Computational Complexity and Statistical Physics*. Oxford University Press, doi:10.1093/oso/9780195177374.001.0001.

[18] J. Raymond, A. Sportiello & L. Zdeborová (2007): *The phase diagram of random 1-in-3 Satisfiability*. *Phys. Rev. E* 76(011101), doi:10.1103/PhysRevE.76.011101.

[19] J. Schmidt-Pruznan & D. Shamir (1985): *Component structure in the evolution of random hypergraphs*. *Combinatorica* 5, pp. 81–94, doi:10.1007/BF02579445.

Matching-Logic-Based Understanding of Polynomial Functors and their Initial/Final Models

Dorel Lucanu
Alexandru Ioan Cuza University of Iași, Romania
dorel.lucanu@gmail.com

In this paper, we investigate how the initial models and the final models for the polynomial functors can be uniformly specified in matching logic.

1 Introduction

It is known that many data types used in programming are defined as initial algebra or final coalgebra for an appropriate functor $F : \mathbb{C} \to \mathbb{C}$, where $\mathbb{C}$ is a category of data types. In this paper we assume that $\mathbb{C}$ is the category of sets (see, e.g., [2]). If F is bicontinous, i.e., it preserves the colimits of ω-sequences and and the limits of ω^{op}-sequences, then the initial algebra (model) is obtained via colimit of the ω-sequence

$$\mathbf{0} \xrightarrow{!} F\,\mathbf{0} \xrightarrow{F\,!} F\,F\,\mathbf{0} = F^2\,\mathbf{0} \xrightarrow{F^2\,!} F^3\,\mathbf{0} \xrightarrow{F^3\,!} \cdots \tag{INI}$$

where $\mathbf{0}$ is the initial object in $\mathbb{C}$, and $\mathbf{0} \xrightarrow{!} X$ is the unique arrow from the initial object, and the final coalgebra (model) is the limit of the ω^{op}-sequence

$$\mathbf{1} \xleftarrow{!} F\,\mathbf{1} \xleftarrow{F\,!} F\,F\,\mathbf{1} = F^2\,\mathbf{1} \xleftarrow{F^2\,!} F^3\,\mathbf{1} \xleftarrow{F^3\,!} \cdots \tag{FIN}$$

where $\mathbf{1}$ is the final object in $\mathbb{C}$, and $\mathbf{1} \xleftarrow{!} X$ is the unique arrow to the final object [2].

This is a nice abstract framework, but, as we know, the evil is hidden in details. How the elements of the initial and final models look like for various concrete functors? How could they be handled in practice?

A possible answer can be obtained by capturing these objects in Matching Logic (ML), the logical foundation of the K Framework, where the program languages and the properties of their programs can be specified in a uniform way (see, e.g., [12, 8, 13, 7, 11]). First steps are done in [6], where the initial algebra semantics is captured in ML, and in [7], where it is shown how examples of inductive/coinductive data types are fully specified in ML. We say that ML captures a (inductive/coinductive) data type DT if there is an ML theory $\mathsf{Th}^{\mathsf{ML}}(DT)$ such that:

- from each $\mathsf{Th}^{\mathsf{ML}}(DT)$-model M we may extract a structure $\alpha(M)$ that is isomorphic to DT, and
- each deduction principle for DT (e.g., induction or coinduction) can be expressed as a theorem within ML using its proof system.

In this paper, we investigate how data types specified as initial F-algebras or as a final F-coalgebras, where F is a polynomial functor, can be captured in ML. The polynomial functors can be defined in two ways:

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 41–55, doi:10.4204/EPTCS.389.4

1. Using "classical" inductive definition of polynomials (see, e.g., [14]): the polynomial functors is the smallest class including the constant and the identity functors, and it is closed to sum, product, and constant-exponent functors.
2. Using unary container functors (see, e.g.,[3]): a polynomial functor is of the form $X \mapsto \sum_{a:A} X^{B[a]}$, where $a : A \vdash B[a]$ is an A-indexed family.

The constant and the identity functors, together with their initial and final models, can be easily captured in ML. Moreover, if we exclude the exponent functor, then the initial algebra can be captured using the approach similar to that from [6]. The exponent functor complicates the things. A possible approach for the classical definition is as follows: supposing that we have captured $F_1\ X$ and $F_2\ X$ by the ML theories (specifications) $\mathrm{SPEC}(F_1\ X)$ and resp. $\mathrm{SPEC}(F_2\ X)$, then use these specifications to build $\mathrm{SPEC}(F_1\ X\ op\ F_2\ X)$, i,.e., to obtain something like

$$\mathrm{SPEC}(F_1\ X\ op\ F_2\ X) = \mathrm{SPEC}(F_1\ X)\ \overline{op}\ \mathrm{SPEC}(F_2\ X)$$

where $\overline{op}$ reflects op to the level of ML specifications, and it follows to be defined. In order to accomplish that, we need a "uniform standard" definition for the specifications $\mathrm{SPEC}(F\ X)$.

The container functors already have a uniform standard definition, and therefore they are more tempting for our investigation. This approach is a work-in-progress, and the results obtained up now show that:

- it is possible to specify unary container functors, together with their initial algebras and final coalgebras, as ML theories;
- it is possible to derive the induction principle and the coinduction principle as theorems of the corresponding theories;
- the ML reasoning can be used to understand better the intimate structure of the initial algebra and the final coalgebra.

The approach is instantiated on the lists example, in order to see the relationship with the classical approach of these data types, and on that of Moore machines, in order to see the (constant) exponential functor at work.

The paper is structured as follows. Section 2 recalls the definitions for polynomial functors and for unary container functors, and the the relationship between them. Section 3 briefly recalls the main elements of matching logic, together with the theories of the equality and of the sorts. Section 4 includes the main contribution, showing that how the unary container functors and their related concepts can be specified in matching logic. The instantiation of the general approach on the examples of lists and Moore machines are included in Section 5 and Section 6, respectively. The paper ends with some concluding remarks.

2 Polynomials Functors

This section briefly recalls the definition of the polynomial functors and their (co)algebras. We consider only the particular case of the functors defined over the category of sets $\mathbb{C}$.

Definition 2.1. [10] Given a functor $F : \mathbb{C} \to \mathbb{C}$, an *$F$-algebra* consists of an object X in $\mathbb{C}$ and an arrow $\alpha : F\ X \to X$. An algebra morphism $(X, \alpha) \to (X', \alpha')$ is an arrow $h : X \to X'$ in $\mathbb{C}$ such that $h \circ \alpha = \alpha' \circ F\ h$. An *initial algebra* for the functor F is an initial object in the category of F-algebras and F-algebra morphisms.

Example 2.1 (Lists)**.** The lists over a set of elements E can be defined as an L-algebra $\alpha : L\,X \to X$ for the functor $L : \mathbb{C} \to \mathbb{C}$ given by $L\,X = \mathbf{1} + E \times X$. Usually, such an algebra is defined by a constant $nil : \mathbf{1} \to X$ and a binary operation $cons : E \times X \to X$. The initial L-algebra is isomorphic to the finite lists inductively defined [15] by the grammar

$$List ::= nil \mid cons(E, List) \tag{LST}$$

We use $\mu X.\, L\,X$ or μL to denote the initial model of the functor L.

Definition 2.2. [10] Given a functor $F : \mathbb{C} \to \mathbb{C}$, an *$F$-coalgebra* consists of an object X in $\mathbb{C}$ and an arrow $\gamma : X \to F\,X$. A coalgebra morphism $(X, \gamma) \to (X', \gamma')$ is an arrow $h : X \to X'$ in $\mathbb{C}$ such that $h \circ \gamma = \gamma' \circ F\,h$. An *final coalgebra* for the functor F is a final object in the category of F-coalgebras and F-coalgebra morphisms.

Example 2.2 (Colists)**.** The colists over a set of elements E can be defined as an L-coalgebra $\gamma : X \to L\,X$, where L is the functor used for lists (see Example 2.1). Usually, such a coalgebra is defined by

- a total operation $\_.? : X \to \{1, 2\}$ such that $x.? = \mathsf{if}\ g(x) = \iota_1(\star)\ \mathsf{then}\ 1\ \mathsf{else}\ 2\ \mathsf{fi}$, where $\star$ is the unique element of $\mathbf{1}$, and
- two partial operations: $\_.\,hd : X \rightharpoonup E$, and $\_.\,tl : X \rightharpoonup X$ such that $x.\,hd = e$ and $x.\,tl = x'$ iff $x.? = 2$ and $g(x) = \iota_2(\langle e, x' \rangle)$.

The final coalgebra is isomorphic to the possible infinite lists coinductively defined by the grammar (LST) [15]. We use $\nu X.\, L\,X$ or νL to denote the final model of the functor L.

Example 2.3 (Moore machines)**.** A Moore machine is an M-coalgebra $\alpha : X \to M\,X$, where M is the functor $M : X \to O \times X^I$. Usually, a Moore machine is defined by an *output* function $out : X \to O$ and a *transition* function $tr : X \to X^I$ [14]. The final coalgebra $\nu X.\, M\,X$ is isomorphic to $(\overline{out}, \overline{tr}) : O^{I^*} \to O \times (O^{I^*})^I$, $\overline{out}(f) = f(\varepsilon)$, $\overline{tr}(f) = g$ with $g(i)(w) = f(\langle i \rangle \cdot w)$ for $i \in I$ and $w \in I^*$, where f is a function $F : I^* \to O$, g a function $g : I \to O^{I^*}$, ε the empty sequence, and $\cdot$ the concatenation of sequences.

Definition 2.3. [14] The class of *polynomial functors* is inductively defined as follows:

- the constant functor A (where A is an object in $\mathbb{C}$) is a polynomial functor;
- the identity functor ID is a polynomial functor;
- the sum $F_1 + F_2$ of two polynomial functors F_1 and F_2 is a polynomial functor;
- the product $F_1 \times F_2$ of two polynomial functors F_1 and F_2 is a polynomial functor; and
- the function space functor $F(X) = X^A$, where A is an arbitrary object.

An alternative to define polynomial functors is given by the (unary) container functors [1, 4, 3].

Definition 2.4. An *A-indexed family* $a : A \vdash B[a]$ is a family of objects of $\mathbb{C}$ indexed by elements of A. Categorically, it is an object B of $\mathbb{C}/A$ and $B[a]$ denotes the elements of B mapped to a.

Definition 2.5. Given an A-indexed family $a : A \vdash B[a]$, the *dependent product* $\prod_{a:A} B[a]$ is the object of the *dependent functions*, which maps an $a{:}A$ into a $b{:}B[a]$. Set theoretically, we have

$$\prod_{a:A} B[a] = \left\{ f \in (\bigcup_{a:A} B[a])^A \;\middle|\; \forall a{:}A.\, f(a) \in B[a] \right\}$$

The *dependent sum* $\sum_{a:A} B[a]$ is the dual of the dependent product and it consists of the pairs $\langle a, b \rangle$ with $a{:}A$ and $b{:}B[a]$.

Definition 2.6. A functor $F : \mathbb{C} \to \mathbb{C}$ is a *(unary) container functor* iff it is naturally isomorphic to a functor of the form $F\,X = \sum_{a:A} X^{B[a]}$, for some objects A in $\mathbb{C}$ and an A-indexed family $a{:}A \vdash B[a]$.

Remark. An element of $\sum_{a:A} X^{B[a]}$ is a pair $\langle a, f \rangle$, where $a{:}A$ is the *shape* and $f : B[a] \to X$ is the function that labels the *positions* $B[a]$ with elements from X. Another way to define B is as an object in $\mathbb{C}/A$.

2.1 Polynomial Functors as Container Functors

Here we recall the relationship between polynomial functors and container functors (see, e.g., [1]).

Constant Functor

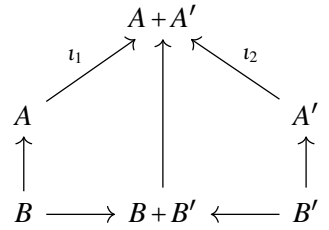
The main idea is to identify the constant value with the shapes A. Consider B as being $a{:}A \vdash \mathbf{0}$ (no positions of shape a), where $\mathbf{0}$ denotes the initial object of $\mathbb{C}$. We get $\sum_{a:A} X^{B[a]} \cong \sum_{a:A} X^{\mathbf{0}} \cong A$.
Remark. The elements of $\sum_{a:A} X^{\mathbf{0}}$ are pairs $\langle a, f : \mathbf{0} \to X\rangle$, where $a \in A$. Since $f : \mathbf{0} \to X$ is unique, we obtain $\langle a, f : \mathbf{0} \to X\rangle \cong a$.

Identity Functor

Consider A as being $\mathbf{1}$ (just one shape) and B as being $\star{:}\mathbf{1} \vdash \mathbf{1}$ (just one position), where $\star$ is the unique element in $\mathbf{1}$. It follows that $\sum_{a:A} X^{B[a]} \cong \sum_{\star} X^{\mathbf{1}} \cong X$.
Remark. The elements of $\sum_{\star} X^{\mathbf{1}}$ are pairs $\langle \star, f : \mathbf{1} \to X\rangle$. Since f selects just one element x in X, it follows that $\langle \star, f : \mathbf{1} \to X\rangle \cong x$.

Sum

Assume that $F\,X = \sum_{a:A} X^{B[a]}$ and $F'\,X = \sum_{a':A'} X^{B'[a]'}$. Then $(F + F')\,X$ is $\sum_{a'':A+A'} X^{[B,B'][a'']}$, where $[B,B'][a''] = B[a]$ if $a'' = \iota_1\,a$, and $[B,B'][a''] = B'[a']$ if $a'' = \iota_2\,a'$.
Remark. The following commutative diagram may help to understand the definition of $a''{:}A + A' \vdash [B,B'][a'']$:



The arrow $B + B' \to A + A'$ is equivalently written as the $A + A'$-indexed set $a''{:}A + A' \vdash [B,B'][a'']$. Set theoretically, $\sum_{a'':A+A'} X^{[B,B'][a'']}$ is the set of pairs $\langle a'', [f,f'] : [B,B''\,]\,a'' \to X\rangle$, where

- either $a'' = \iota_1[a]$, $\langle a, f : B[a] \to X\rangle$ in $F\,X$, and $\forall x{:}B[a].\ [f,f']\,x = f\,x$, or
- $a'' = \iota_2[a]'$, $\langle a', f' : B'[a]' \to X\rangle$ in $F'\,X$, and $\forall x'{:}B[a'].\ [f,f']\,x' = f'\,x'$.

In other words, the shape of the sum is the sum of the component shapes, and a labelling function for the sum is the sum of two corresponding component labelling functions.

Product

We have $(F + F')\,X = \sum_{\langle a,a'\rangle:A\times A'} X^{\langle B,B'\rangle[\langle a,a'\rangle]}$, where $F\,X$ and $F'\,X$ are similar to those from the sum, and $\langle B,B'\rangle[\langle a,a'\rangle] = B[a] + B'[a']$ (the positions of the product is the disjoint union of the component positions).
Remark. We prefer to write $\langle B,B'\rangle[a,a']$ for $\langle B,B'\rangle[\langle a,a'\rangle]$. Set theoretically, $\sum_{\langle a.a'\rangle:A\times A'} X^{\langle B,B'\rangle[a,a']}$ is the set of pairs $\langle\langle a,a'\rangle, [f,f'] : B[a] + B'[a'] \to X\rangle$, where $f : B[a] \to A$ and $f' : B'[a'] \to A'$. In other words, the shape of the product is the product of the component shapes and a labelling function for the product is the sum of two corresponding component labelling functions.

Exponentiation

Assuming $F\ X = \sum_{a:A} X^{B[a]}$, the (constant) exponent functor $(F\ X)^C$ is $\sum_{g:C\to A} X^{\sum_{c:C} B[g\ c]}$.

Remark. An element of $\sum_{g:C\to A} X^{\sum_{c:C} B[g\ c]}$ is a pair $\langle g, f\rangle$ consisting of a function $g : C \to A$ assigning shapes to C and a C-indexed function $c{:}C \vdash f_c : B[g\ c] \to X$ labelling the positions $B[g\ c]$ for each c in C.

3 Matching Logic (ML)

Matching logic [12, 8, 7] provides a unifying framework for defining semantics of programming languages. A programming language is defined in matching logic as a *logical theory*, i.e., a set of axioms. The key concept in matching logic is that of *patterns*, which can be *matched* by certain elements. By building complex patterns, we can match elements that have complex structures or certain properties, or both. The presentation of matching logic in this review section follows [7].

Definition 3.1. Let us fix two sets EV and SV. The set EV includes *element variables* $x, y, \ldots$. The set SV includes *set variables* $X, Y, \ldots$. A *matching logic signature* Σ is a set of *(constant) symbols*, denoted $\sigma, \sigma_1, \sigma_2, \ldots$. Let us fix a signature Σ. The set of *(Σ-)patterns* is inductively defined as follows:

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\ \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\ \varphi \mid \mu X.\ \varphi$$

where in $\mu X.\ \varphi$, called a least-fixpoint pattern, we require that φ is positive in X, i.e., X does not occur in an odd number of times of the left-hand sides of implications $\varphi_1 \to \varphi_2$.

Definition 3.2. A *(matching logic) Σ-model M* consists of

1. a nonempty *carrier set*, which we also denote M;
2. an *application function* $\_\cdot\_ : M \times M \to \mathbb{P}(M)$, where $\mathbb{P}(M)$ is the powerset of M; and
3. a *symbol interpretation* $\sigma_M \subseteq M$ as a subset for $\sigma \in \Sigma$.

Definition 3.3. Given M and a *variable valuation* $\rho{:}(EV \cup SV) \to M \cup \mathbb{P}(M)$ such that $\rho(x) \in M$ for $x \in EV$ and $\rho(X) \subseteq M$ for $X \in SV$, we inductively define *pattern valuation* $|\varphi|_{M,\rho}$ as follows:

1. $|x|_{M,\rho} = \{\rho(x)\}$ for $x \in EV$
2. $|X|_{M,\rho} = \rho(X)$ for $X \in SV$
3. $|\sigma|_{M,\rho} = \sigma_M$ for $\sigma \in \Sigma$
4. $|\varphi_1\ \varphi_2|_{M,\rho} = \bigcup_{a_i \in |\varphi_i|_{M,\rho}, i\in\{1,2\}} a_1 \cdot a_2$; note that $a_1 \cdot a_2$ is a subset of M.
5. $|\bot|_{M,\rho} = \varnothing$
6. $|\varphi_1 \to \varphi_2|_{M,\rho} = M \setminus (|\varphi_1|_{M,\rho} \setminus |\varphi_2|_{M,\rho})$
7. $|\exists x.\ \varphi|_{M,\rho} = \bigcup_{a\in M} |\varphi|_{M,\rho[a/x]}$
8. $|\mu X.\ \varphi|_{M,\rho} = \mathbf{lfp}(A \mapsto |\varphi|_{M,\rho[A/X]})$

where $\rho[a/x]$ (resp. $\rho[A/X]$) is the valuation ρ' with $\rho'(x) = a$ (resp. $\rho'(X) = A$) and agreeing with ρ on all the other variables. in $EV \cup SV \setminus \{x\}$ (resp. $EV \cup SV \setminus \{X\}$). We use $\mathbf{lfp}(A \mapsto |\varphi|_{M,\rho[A/X]})$ to denote the smallest set A such that $A = |\varphi|_{M,\rho[A/X]}$. The existence of such an A is guaranteed by the requirement that φ is positive in X. We abbreviate $|\varphi|_{M,\rho}$ as $|\varphi|_\rho$ and further as $|\varphi|$ if φ is closed.

Definition 3.4. We say that φ *holds* in M, written $M \vDash \varphi$, if $|\varphi|_{M,\rho} = M$ for all ρ. For a pattern set Γ, we write $M \vDash \Gamma$, if $M \vDash \psi$ for all $\psi \in \Gamma$. We write $\Gamma \vDash \varphi$, if $M \vDash \Gamma$ implies $M \vDash \varphi$ for all M.

The following common constructs can be defined from the basic pattern syntax as syntactic sugar in the usual way:

$$\neg\varphi \equiv \varphi \to \bot \qquad \varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \to \varphi_2 \qquad \varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$$
$$\top \equiv \neg\bot \qquad \forall x.\ \varphi \equiv \neg\exists x.\ \neg\varphi \qquad \nu X.\ \varphi \equiv \neg\mu X.\ \neg\varphi[\neg X/X]$$

We assume the standard precedence between the above constructs.

Equality

The equality can be defined as a derived construct (see [12, 7]). Two patterns φ_1 and φ_2 are equal, written $\varphi_1 = \varphi_2$, iff it is equivalent to $\top$ if the two patterns are matched by the same elements. Otherwise, it is equivalent to $\bot$. To express that in ML, a new symbol $def \in \Sigma$ is introduced, called the *definedness* symbol, and specify it with the axiom (Definedness). The resulted theory can be described as follows:

spec EQUALITY
  Symbols: def
  Notations: $\lceil\varphi\rceil \equiv \mathsf{def}\ \varphi$
  Axioms: (Definedness) $\forall x.\ \lceil x\rceil$
  Notations:
    $\lfloor\varphi\rfloor \equiv \neg\lceil\neg\varphi\rceil$ // totality
    $\varphi_1 = \varphi_2 \equiv \lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor$ // equality
    $\varphi_1 \subseteq \varphi_2 \equiv \lfloor\varphi_1 \to \varphi_2\rfloor$ // set inclusion
    $x \in \varphi \equiv x \subseteq \varphi$ // membership
endspec

Sorts

Matching logic has no builtin support for sorts. Instead, we define a *theory of sorts* to support arbitrary sort structures following the "sort-as-predicate" paradigm. A *sort* has a name and is associated with a set of its *inhabitants*. In matching logic, we use a symbol $s \in \Sigma$ to represent the sort name and use $(\mathsf{inh}\ s)$ to represent all its inhabitants, where $\mathsf{inh} \in \Sigma$ is an ordinary symbol. For better readability, we define the notation $\top_s \equiv \mathsf{inh}\ s$.

spec SORT Imports: EQUALITY
  Symbols: *inh*, *Sort*
  Notations:
    $\top_s \equiv inh\ \ s$ // inhabitants of sort s
    $s_1 \le s_2 \equiv \top_{s_1} \subseteq \top_{s_2}$ // subsort relation
    $\neg_s\varphi \equiv (\neg\varphi) \wedge \top_s$ // negation within sort s
    $\forall x{:}s.\ \varphi \equiv \forall x.\ x \in \top_s \to \varphi$ // $\forall$ within sort s
    $\exists x{:}s.\ \varphi \equiv \exists x.\ x \in \top_s \wedge \varphi$ // $\exists$ within sort s
    $\mu X{:}s.\ \varphi \equiv \mu X.\ X \subseteq \top_s \wedge \varphi$ // μ within sort s
    $\nu X{:}s.\ \varphi \equiv \nu X.\ X \subseteq \top_s \wedge \varphi$ // ν within sort s
    $\varphi{:}s \equiv \exists z{:}s.\ \varphi = z$ // "typing"
    $f{:}s_1 \otimes \cdots \otimes s_n \to s \equiv \forall x_1{:}s_1 \ldots \forall x_n{:}s_n.\ \exists y{:}s.\ f\,x_1 \ldots x_n = y$
      // functional
    $f{:}s_1 \otimes \cdots \otimes s_n \rightharpoonup s \equiv \forall x_1{:}s_1 \ldots \forall x_n{:}s_n.\ \exists y{:}s.\ f\,x_1 \ldots x_n \to y$
      // partially functional

Axioms:
$\exists x.\ Sort = x$
$Sort \in \top_{Sort}$
endspec

The ML specifications for the sum, product, and function sorts are given in [7]. For reader convenience, we recall them in Appendix A.

4 Specifying Initial Algebra and Final Coalgebra for Container Functors in ML

In this section we show how to specify a unary container functor in ML and how to extract the structures for their initial algebra and the final coalgebra.

4.1 Capturing Elements from the Category $\mathbb{C}$

- $\mathbf{0}$ is specified by $\bot$;
- $\mathbf{1} = \{\star\}$ is specified by
  - a symbol $star \in \Sigma$;
  - a notation: $\star \equiv star$; and
  - an axiom
    $\exists y.\ \star = y$ (SINGLETON)
- $\mathbf{0} \overset{¡}{\to} X$ is specified by
  - a symbol $iniMor \in \Sigma$;
  - a notation: $¡ \equiv iniMor$; and
  - an axiom
    $\forall x.\ ¡\, x = \bot$ (CAPTURES $\mathbf{0} \overset{¡}{\to} X$)
- $\mathbf{1} \overset{!}{\leftarrow} X$ is specified by
  - a symbol $finMor \in \Sigma$;
  - a notation: $! \equiv finMor$; and
  - an axiom
    $\forall x.\ !\, x = \star$ (CAPTURES $\mathbf{1} \overset{!}{\leftarrow} X$)

Remark. An alternative way to specify $\mathbf{1}$ is by $\top$, in which case $\mathbf{1} \overset{!}{\leftarrow} X$ is the inclusion: $\forall x.\ x \in X \to !\, x = x$. This version is used when we compute the greatest fixpoint.

4.2 Expressing Indexed Families in ML

An A-indexed family $a{:}A \vdash B[a]$ is specified by a constant symbol $depOf \in \Sigma$, a notation $B[a] \equiv depOf\ B\ a$, and two axioms:

- A is a sort: $A{:}Sort$ (equivalent to $A \in \top_{Sort}$), and
- for each $a{:}A$, $B[a]$ is a sort: $\forall a{:}A.\ B[a]{:}Sort$

where we assumed that A and B are specified as functional patterns.

4.3 Expressing Dependent Products/Sums in ML

A dependent product $\prod_{a:A} B[a]$ is specified by a constant symbol $DepProd \in \Sigma$, a notation $\prod_{a:A} B[a] \equiv DepProd\ A\ B$ (a plays a local role and its name can be changed[1]), and by adding the axioms:

- $\prod_{a:A} B[a]$ is a sort: $\prod_{a:A} B[a] : Sort$
- $\top_{\prod_{a:A} B[a]}$ is the set of dependent functions:

$$\top_{\prod_{a:A} B[a]} = \exists f.\ f \wedge ((\lceil \top_A \rceil \wedge \forall a{:}A.\ \exists b{:}B[a].\ f\ a = b) \vee (\neg \lceil \top_A \rceil \wedge f = \text{¡}))$$

  The above axiom distinguishes between two cases: the sort A is non-empty, in which case $\top_{\prod_{a:A} B[a]}$ includes the dependent functions that maps an $a{:}A$ into a $b{:}B[a]$, and when the sort A is empty, in which case $\top_{\prod_{a:A} B[a]}$ consists of the function given by the initial morphism.

Similarly, a dependent sum $\sum_{a:A} B[a]$ is specified by a constant $DepSum \in \Sigma$, a notation $\sum_{a:A} B[a] \equiv DepSum\ A\ B$ (again, a plays a local role and its name can be changed), and by adding the axioms:

- $\sum_{a:A} B[a]$ is a sort: $\sum_{a:A} B[a] : Sort$
- $\top_{\sum_{a:A} B[a]}$ is the set fo dependent pairs:

$$\top_{\sum_{a:A} B[a]} = \exists a{:}A.\ \exists b{:}B[a].\ \langle a, b\rangle$$

4.4 Expressing Unary Container Functors in ML

Let $X \mapsto F\ X = \sum_{a:A} X^{B[a]}$ be a container functor. Recall that $F\ X$ is the set of pairs $\langle a, f\rangle$ with $a{:}A$ and $f{:}X^{B[a]}$ (or, equivalently, $f{:}B[a] \to X$). If X is specified as the set of inhabitants $\top_s$ of a sort $s{:}Sort$ and $\top_{B[a]} \neq \bot$ (that is equivalent to $B[a] \not\cong \mathbf{0}$ in $\mathbb{C}$), then $X^{B[a]}$ is specified by the sort $B[a] \textcircled{\to} s$ [7] (see also Appendix A.3) and the specification of $\sum_{a:A} X^{B[a]}$ is a particular case of dependent sum specification. Otherwise, we have to explicitly specify $X^{B[a]}$ by the notation

$$X^{B[a]} \equiv \exists f.\ f \wedge (((\top_{B[a]} \neq \bot) \wedge \forall b{:}B[a].\ \exists x.\ f\ b = x \wedge x \in X) \vee ((\top_{B[a]} = \bot) \wedge f = \text{¡}))$$

and use it directly in the specification of $\sum_{a:A} X^{B[a]}$:

$$\exists a{:}A.\ \exists f.\ \langle a, f\rangle \wedge f \in X^{B[a]}$$

which is equivalent to

$$\exists a{:}A.\ \exists f.\ \langle a, f\rangle \wedge (((\top_{B[a]} \neq \bot) \wedge \forall b{:}B[a].\ \exists x.\ f\ b = x \wedge x \in X) \vee ((\top_{B[a]} = \bot) \wedge f = \text{¡}))$$

Recall that if $\top_{B[a]} = \bot$ ($B[a] \cong \mathbf{0}$) then there is just one function $\mathbf{0} \xrightarrow{\text{¡}} X$ specified by $\text{¡} \equiv iniMor$.

4.5 Specifying Initial Algebra

Let $X \mapsto F\ X = \sum_{a:A} X^{B[a]}$ be a container functor. The initial F-algebra is specified by using the characterization given by the "no junk and no confusion" properties for the constructors [7]:

- a constructor $cons \in \Sigma$ specified by the following axioms:
  $\forall a{:}A.\ \forall f.\ f \in X^{B[a]} \to \exists x.\ x \in X \wedge cons\ \langle a, f\rangle = x$ (FUNCTIONAL)
  $\forall a, a'{:}A.\ \forall f, f'.\ cons\ \langle a, f\rangle = cons\ \langle a', f'\rangle \to a = a' \wedge f = f'$ (NO CONFUSION)
- a sort μF with initial semantics:
  $\top_{\mu F} = \mu X.\ \exists a{:}A.\ cons\ \langle a, X^{B[a]}\rangle$ (NO JUNK)

[1] Actually, Π should be captured as a binder [9], but this is not needed for the purpose of this paper.

Computing the least fixpoint Let $\varphi_F(X)$ denote the pattern $\exists a{:}A.\ cons\ \langle a, X^{B[a]}\rangle$. Given a model M and a valuation ρ, we have $|\mu X.\ \varphi_F(X)|_{M,\rho} = \mathbf{lfp}(A \mapsto |\varphi_F(X)|_{M,\rho[A/X]})$. We also denote by $A \mapsto \phi_F(A)$ the function $A \mapsto |\varphi_F(X)|_{M,\rho[A/X]}$. If ϕ_F is continuous, then $\mathbf{lfp}(\phi_F) = \bigcup_{n\geq 0} \phi_F^n(\varnothing) = \varnothing \cup \phi_F(\varnothing) \cup \phi_F^2(\varnothing) \cup \cdots$. Since $\phi_F(\varnothing) = |\varphi_F(\bot)|_{M,\rho})$ and writing $\varphi(\psi)$ for $\varpi[\psi/X]$, we (informally) obtain that $\mathbf{lfp}(\phi_F)$ is the interpretation of the infinite disjunction

$$\bot \vee \varphi_F(\bot) \vee \varphi_F^2(\bot) \vee \varphi_F^3(\bot) \vee \cdots \tag{LFP}$$

according to M and ρ. We have $\varphi_F^n(\bot) \to \varphi_F^{n+1}(\bot)$ and each $\varphi_F^n(\bot)$ gives an approximation of $\top_{\mu F} \cong \mathbf{lfp}(\phi_F)$. So, in order to understand how the elements of the initial algebra look like, we have to investigate these ML patterns.

$\varphi_F(\bot)$**.** We have $X^{B[a]} = \bot^{B[a]} \not\equiv \bot$ iff $\top_{B[a]} = \bot$, because otherwise we have $(\forall b{:}B[a].\ \exists y.\ f\ b = y \wedge y \in \bot) = \bot$. It follows that $\bot^{B[a]}$ consists of the unique function ¡. We obtain

$$\varphi_F(\bot) = \exists a_1{:}A.\ cons\langle a_1, ¡\rangle \wedge (\top_{B[a_1]} = \bot)$$

i.e., each $a_1{:}A$, with its corresponding dependent sort $B[a_1]$ empty, defines a constant constructor. If $\forall a{:}A.\ \top_{B[a]} \not\equiv \bot$ then $(\exists a{:}A.\ cons\ \langle a, \top_{B[a] \ominus \ltimes X}\rangle) = \bot$ and hence $\top_{\mu L} = \bot$.

$\varphi_F^2(\bot) = \exists a_2{:}A.\ cons\ \langle a_2, \varphi_F(\bot)^{B[a_2]}\rangle$**.** If $\top_{B[a_2]} = \bot$ then $\varphi_F(\bot)^{B[a_2]}$ consists of the constant constructor $cons\ \langle a_2, ¡\rangle$, i.e., $\varphi_F(\bot)^{B[a_2]} = \bot^{B[a_2]}$. If $\top_{B[a_2]} \not\equiv \bot$ then we have

$$\varphi_F(\bot)^{B[a_2]} = \exists f.\ f \wedge \forall b{:}B[a_2].\ \exists a_1{:}A.\ f\ b = cons\langle a_1, ¡\rangle \wedge (\top_{B[a_1]} = \bot)$$

We obtain

$$\varphi_F^2(\bot) = \varphi_F(\bot) \vee \exists a_2{:}A.\ cons\ \langle a_2, \varphi_F(\bot)^{B[a_2]}\rangle \wedge (\top_{B[a_2]} \not\equiv \bot)$$

...

Remark. The ML pattern (LFP) can be seen as an informal translation in ML of the colimit (INI).

Deriving Induction Principle Once we have seen how the least fixpoint is computed, we may derive the following *Induction Principle*:

$$\frac{\forall a{:}A.\ cons\ \langle a, \psi\rangle \to \psi}{\top_{\mu F} \to \psi}$$

The justification for this principle is similar to that for lists given in [7].

4.6 Specifying Final Coalgebra

Let $X \mapsto F\ X = \sum_{a:A} X^{B[a]}$ be a container functor. The final F-coalgebra is specified by:

- the constructor *cons* together with its axioms;
- a sort νF with final semantics:
  $\top_{\nu F} = \nu X.\ \exists a{:}A.\ cons\ \langle a, X^{B[a]}\rangle$ (NO REDUNDANCY (COJUNK))
- two destructors $out, nxt \in \Sigma$ together with the notations:
  $x.out \equiv out\ x$
  $x.nxt \equiv nxt\ x$
  and the axioms:
  $\forall a{:}A.\ \forall f.\ (cons\ \langle a, f\rangle).out = a$ (NO AMBIGUITY (COCONFUSION).1))
  $\forall a{:}A.\ \forall f.\ (cons\ \langle a, f\rangle).nxt = f$ (NO AMBIGUITY (COCONFUSION).2))
  $\forall x{:}\nu F.\ (cons\ \langle x.out, x.nxt\rangle) = x$ (NO AMBIGUITY (COCONFUSION).3))

Computing the greatest fixpoint Since $\mathbf{gfp}(\phi_F) = \bigcap_{n\geq 0} \phi_F^n(M) = M \cap \phi_F(M) \cap \phi_F^2(M) \cap \cdots$, we have to investigate the infinite conjunction

$$\top \wedge \varphi_F(\top) \wedge \varphi_F^2(\top) \wedge \varphi_F^3(\top) \wedge \cdots \tag{GFP}$$

in order to understand how the elements of the final coalgebra look like.

$\varphi_F(\top)$**.** We have $cons\ \langle a, \top^{B[a]} \rangle = \exists x, y.\ x \wedge x.out = a \wedge x.nxt = y$.

$\varphi_F^2(\top)$**.** We have

$$\begin{aligned}\varphi_F^2(\top) &= \exists a{:}A.\ cons\ \langle a, \varphi_F(\top)^{B[a]} \rangle \\ &= (\exists a{:}A.\ \exists x.\ \exists y{:}\ltimes F\ \top.\ x \wedge x.out = a \wedge x.nxt = y) \\ &= (\exists a, a'{:}A.\ \exists x, z.\ \exists y{:}\ltimes F\ \top.\ x \wedge x.out = a \wedge x.nxt = y \wedge y.out = a' \wedge y.nxt = z)\end{aligned}$$

...

Remark. The ML pattern (GFP) can be seen as an ML informal translation of the colimit (FIN).

Deriving Coinduction Principle Once we have seen how the greatest fixpoint is computed, we may derive the following *Conduction Principle*:

$$\frac{\forall a{:}A.\ \psi \to cons\ \langle a, \psi \rangle}{\top_{\psi \to \nu F}}$$

The justification for this principle is similar to that for streams given in [7].

5 Case Study: Lists Using Container Functors

First, we express $L\,X = \mathbf{1} + E \times X$ as a unary container functor, using $\mathbf{1} \cong \sum_{\star:\mathbf{1}} X^{\mathbf{0}[\star]}$, $E \cong \sum_{e:E} X^{\mathbf{0}[e]}$, and $X \cong \sum_{\star:\mathbf{1}} X^{\mathbf{1}[\star]}$:

$$\begin{aligned}L\,X &= \sum_{\star:\mathbf{1}} X^{\mathbf{0}[\star]} + \sum_{e:E} X^{\mathbf{0}[e]} \times \sum_{\star:\mathbf{1}} X^{\mathbf{1}[\star]} \\ &= \sum_{\star:\mathbf{1}} X^{\mathbf{0}[\star]} + \sum_{\langle e,\star\rangle:E\times\mathbf{1}} X^{\mathbf{0}[e]+\mathbf{1}[\star]} \\ &= \sum_{a:\mathbf{1}+E\times\mathbf{1}} X^{[\mathbf{0},\langle\mathbf{0},\mathbf{1}\rangle][a]}\end{aligned}$$

Using the isomorphisms $E \times \mathbf{1} \cong E$ and $\langle \mathbf{0}, \mathbf{1} \rangle [\langle e, \star \rangle] = \mathbf{0}[e] + \mathbf{1}[\star] = \mathbf{0} + \mathbf{1} \cong \mathbf{1}$ in the category $\mathbb{C}$, we obtain the reduced form $L^c\,X = \sum_{a:\mathbf{1}+E} X^{[\mathbf{0},\mathbf{1}][a]}$ of the container functor L. From the definition of the sum of container functors we deduce that the elements of $\sum_{a:\mathbf{1}+E} X^{[\mathbf{0},\mathbf{1}][a]}$ are pairs $\{\langle \star, f\rangle \mid f : \mathbf{0} \to X\} \uplus \{\langle e, f'\rangle \mid e : E, f' : \mathbf{1} \to X\}$. The only function $\mathbf{0} \to X$ is $\mathfrak{i}$, and $f' : \mathbf{1} \to X \cong f' \in X$.

The specification in ML of the initial algebra μL^c includes:

- a constructor symbol *cons* and a sort symbol μF;

- the axioms:

$\exists y{:}\mu L^c.\ cons\ \langle \star, \mathfrak{i} \rangle = y$ (FUNCTIONAL.1)

$\forall e{:}E.\ \forall f'{:}\mu L^c.\ \exists y{:}\mu L^c.\ cons\ \langle e, f' \rangle = y$ (FUNCTIONAL.2)

$\forall e{:}E.\ \forall f'{:}\mu L^c.\ cons\ \langle \star, \mathfrak{i} \rangle \neq cons\ \langle e, f' \rangle$ (NO CONFUSION.1)

$\forall e, e'{:}E.\ \forall f, f'{:}\mu L.\ cons\ \langle e, f \rangle = cons\ \langle e, f' \rangle \rightarrow e = e' \wedge f = f'$ (NO CONFUSION.2)

$\top_{\mu L^c} = \mu X.\ (cons\ \langle \star, \mathfrak{i} \rangle \vee cons\ \langle E, X \rangle)$ (NO JUNK)

Comparing with the specification from [7], we obviously have the equivalences $nil \cong cons\ \langle \star, \mathfrak{i} \rangle$ and $cons\ e\ \ell \cong cons\ \langle e, \ell \rangle$.

The ML specification of the final coalgebra further includes:

- the destructor symbols $out, nxt \in \Sigma$;
- the axioms

$cons\ \langle \star, \mathfrak{i} \rangle\ .\ out = \star$ (NO COCONFUSION.1.1))

$\forall e{:}E.\ \forall f{:}\nu L^c.\ cons\ \langle e, f \rangle\ .\ out = e$ (NO COCONFUSION.1.2))

$cons\ \langle \star, \mathfrak{i} \rangle\ .\ nxt = \mathfrak{i}$ (NO COCONFUSION.2.1))

$\forall e{:}E.\ \forall f{:}\nu L^c.\ cons\ \langle e, f \rangle\ .\ nxt = f$ (NO COCONFUSION.2.2))

$\forall x{:}\nu L^c.\ (cons\ \langle x.out, x.nxt \rangle) = x$ (NO COCONFUSION.3))

$\top_{\nu L^c} = \nu X.\ (cons\ \langle \star, \mathfrak{i} \rangle \vee cons\ \langle E, X \rangle)$ (NO COJUNK)

Comparing with the specification of streams (inifinite lists) from [7], we obviously have the equivalences $\ell.\ out \cong hd\ \ell$ and $\ell.\ nxt \cong tl\ \ell$. Using $nil \cong cons\ \langle \star, \mathfrak{i} \rangle$, we get $hd\ nil = \star$ and $tl\ nil = \mathfrak{i}$, which is different from the usual approach, where *hd* and *tl* are partial operations.

6 Case Study: Moore Machines

Here we consider an example of (constant) exponential functor, whose ML specification is more tricky. Moore machines (automata) have the signature given by the functor $M\ X = O \times X^I$, where O is for outputs and I for inputs.

Capturing Final M-Coalgebra in ML Using Container Functors

We first express M as a unary container functor:

$$\begin{aligned}
M\ X &= O \times X^I \\
&\cong \sum_{o:O} X^{\mathbf{0}} \times \Big(\sum_{\star:\mathbf{1}} X^{\mathbf{1}}\Big)^{\sum_{i:I} X^{\mathbf{0}}} \\
&= \sum_{o:O} X^{\mathbf{0}} \times \sum_{g:I\to\mathbf{1}} X^{\sum_{i:I} \mathbf{1}[g\ i]} \\
&= \sum_{\langle o,g\rangle:(O\times(I\to\mathbf{1}))} X^{\mathbf{0}[o]+\sum_{i:I}\mathbf{1}[g\ i]} \\
&\cong \sum_{o:O} X^{\sum_{i:I}\mathbf{1}} \\
&\cong \sum_{o:O} X^{I} \\
&= M^c\ X
\end{aligned}$$

The instantiation of the ML specification for the final coalgebra is as follows:

- the constructor $cons \in \Sigma$ is specified by the following axioms:
  $\forall o{:}O.\ \forall f.\ f \in X^I \to \exists x.\ x \in X \wedge cons\ \langle a, f\rangle = x$ (FUNCTIONAL)
  $\forall o, o'{:}O.\ \forall f, f'.\ cons\ \langle a, f\rangle = cons\ \langle a', f'\rangle \to a = a' \wedge f = f'$ (NO CONFUSION)
- the destructors $out, nxt \in \Sigma$ are specified by the axioms:
  $\forall o{:}O.\ \forall f.\ f \in X^I \to (cons\ \langle a, f\rangle).out = a$ (NO COCONFUSION.1))
  $\forall a{:}A.\ \forall f.\ f.\ f \in X^I \to (cons\ \langle a, f\rangle).nxt = f$ (NO COCONFUSION.2))
  $\forall x{:}\nu F.\ (cons\ \langle x.out, x.nxt\rangle) = x$ (NO COCONFUSION.3))
- the sort νM^c with final semantics:
  $\top_{\nu M^c} = \nu X.\ \exists o{:}O.\ cons\ \langle o, X^I\rangle$ (NO COJUNK)

We should have

$$\top_{\nu M^c} \cong \nu X.\ M^C\ X$$

Computing $\top \wedge \varphi_M(\top) \wedge \varphi_M^2(\top) \wedge \varphi_M^3(\top) \wedge \cdots$, where $\varphi_M(X) \equiv \exists o{:}O.\ cons\ \langle o, X^I\rangle$:

$\varphi_M(\top) = \exists o_0{:}O.\ cons\ \langle o, X^I\rangle = \exists o_0{:}O.\ \exists f_0.\ cons\ \langle o_0, f_0\rangle \wedge (\forall i{:}I.\ \exists x_1.\ f_0\ i = x_1)$. When describing a dynamic system, the use of destructors is more intuitive:

$$\varphi_M(\top) = \exists x_0.\ \exists o_0{:}O.\ x_0 \wedge (x_0.out = o_0 \wedge \forall i{:}I.\ \exists y.\ x_0.nxt\ i = y)$$

It is easy to see that $x_0 = cons\ \langle o_0, f_0\rangle$ and $f_0\ i = x_0.nxt\ i$.

$\varphi_M^2(\top) = \exists o_1{:}O.\ cons\ \langle o_1, X^I\rangle = \exists o_1{:}O.\ \exists f_1.\ cons\ \langle o_1, f_1\rangle \wedge (\forall i{:}I.\ \exists x_2.\ f_1\ i = x_2 \wedge x_2 \in \varphi_M(\top))$. Again, it becomes more suggestive using the destructors:

$$\begin{aligned}\varphi_M^2(\top) &= \exists x_1.\ \exists o_1{:}O.\ x_1 \wedge (x_1.out = o_1 \wedge \forall i{:}I.\ \exists x_0.\ x_1.nxt\ i = x_0 \wedge x_0 \in \varphi_M(\top))\\ &= \exists x_1.\ \exists o_1{:}O.\ x_1 \wedge (x_1.out = o_1 \wedge \forall i{:}I.\ \exists x_0.\ \exists o_0{:}O.\ x_0.out = o_0 \wedge \forall i{:}I.\ \exists y.\ x_0.nxt\ i = y)\end{aligned}$$

...

We have $\top_{\nu M} \ni x \cong f \in O^{I^*}$ iff $x.out = f\ \varepsilon$ and $\forall i{:}I.\ x.nxt\ i = \overline{tr}(f)(i)$.

7 Conclusion

The technical experiments reported in this paper show that both the initial models and the final models for polynomial functors can be fully captured in matching logic (ML) using their representation as unary container functors. The ML specification of the polynomial functors is possible due to the fact the sum, product, and function sorts can be specified in ML [7], and these specifications are a part of capturing the category of sets $\mathbb{C}$ in ML.

A functor represented as a "classical" polynomial can be translated into a container functor shape using the fact the later are closed under sum, product, and exponential. However, the result could be cumbersome and not easy to handle in matching logic because the construction starts from constant and identity functors. Therefore it is preferable to simplify it using the isomorphisms in the category of sets.

This result can help in defining in ML programming languages using both inductive data-types and coinductive data-types. Another advantage is given by a better understanding of the abstract constructions from the category theory. A possibly use is as follows:

- define in front-end a suitable syntax for data types intended to be defined as initial algebra or final coalgebra;
- extract the canonical form of the functor underlying the front-end definition;
- generate the corresponding ML theory;
- derive the proof principles needed to soundly handle the defined data type.

Future work will focus on the following aspects of the proposed approach:

- a more formal presentation of the approach;
- how to capture in ML the iteration principle and the primitive recursive principle;
- extending the approach to larger classes of functors admitting initial algebras and final coalgebras, e.g., indexed containers [3], the bounded natural functors (BNFs) underlying Isabelle/HOL's datatypes [16], or the quotients of polynomial functors, experimentally implemented in Lean [5].

References

[1] Michael Gordon Abbott, Thorsten Altenkirch & Neil Ghani (2005): *Containers: Constructing strictly positive types*. *Theor. Comput. Sci.* 342(1), pp. 3–27, doi:10.1016/j.tcs.2005.06.002.

[2] Jirí Adámek (2003): *On final coalgebras of continuous functors*. *Theor. Comput. Sci.* 294(1/2), pp. 3–29, doi:10.1016/S0304-3975(01)00240-7.

[3] Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride & Peter Morris (2015): *Indexed containers*. *J. Funct. Program.* 25, doi:10.1017/S095679681500009X.

[4] Thorsten Altenkirch & Peter Morris (2009): *Indexed Containers*. In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, IEEE Computer Society, pp. 277–285, doi:10.1109/LICS.2009.33.

[5] Jeremy Avigad, Mario Carneiro & Simon Hudon (2019): *Data Types as Quotients of Polynomial Functors*. In John Harrison, John O'Leary & Andrew Tolmach, editors: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, *LIPIcs* 141, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 6:1–6:19, doi:10.4230/LIPIcs.ITP.2019.6.

[6] Xiaohong Chen, Dorel Lucanu & Grigore Roşu (2020): *Initial Algebra Semantics in Matching Logic*. Technical Report, University of Illinois at Urbana-Champaign. Available at `http://hdl.handle.net/2142/107781`.

[7] Xiaohong Chen, Dorel Lucanu & Grigore Roşu (2021): *Matching logic explained*. *Journal of Logical and Algebraic Methods in Programming* 120, pp. 1–36, doi:10.1016/j.jlamp.2021.100638.

[8] Xiaohong Chen & Grigore Roşu (2019): *Matching μ-logic*. In: *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*, IEEE, Vancouver, Canada, pp. 1–13, doi:10.1109/LICS.2019.8785675.

[9] Xiaohong Chen & Grigore Roșu (2020): *A general approach to define binders using matching logic*. *Proc. ACM Program. Lang.* 4(ICFP), pp. 88:1–88:32, doi:10.1145/3408970.

[10] Bart Jacobs & Jan Rutten (2012): *An introduction to (co)algebra and (co)induction*. In Davide Sangiorgi & Jan J. M. M. Rutten, editors: *Advanced Topics in Bisimulation and Coinduction*, *Cambridge tracts in theoretical computer science* 52, Cambridge University Press, pp. 38–99. Available at `http://www.cambridge.org/gb/knowledge/isbn/item6542021`.

[11] Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang & Grigore Roșu (2023): *Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier*. *Proc. ACM Program. Lang.* 7(OOPSLA1), pp. 56–84, doi:10.1145/3586029.

[12] Grigore Roşu (2017): *Matching logic*. *Logical Methods in Computer Science* 13(4), pp. 1–61, doi:10.23638/LMCS-13(4:28)2017.

[13] Grigore Roșu & Xiaohong Chen (2020): *Matching logic: the foundation of the K framework (invited talk)*. In Jasmin Blanchette & Catalin Hritcu, editors: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, ACM, p. 1, doi:10.1145/3372885.3378574.

[14] Jan J. M. M. Rutten (2000): *Universal coalgebra: a theory of systems*. *Theor. Comput. Sci.* 249(1), pp. 3–80, doi:10.1016/S0304-3975(00)00056-6.

[15] Davide Sangiorgi (2012): *An Introduction to Bisimulation and Coinduction*. Cambridge University Press. A preliminary version of Chapter 2 from which we adapt paterial can be found at `http://www.cs.unibo.it/~sangio/DOC_public/corsoFL.pdf`.

[16] Dmitriy Traytel, Andrei Popescu & Jasmin Christian Blanchette (2012): *Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving*. In: *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, IEEE Computer Society, pp. 596–605, doi:10.1109/LICS.2012.75.

A Capturing Sort Operator in ML

We recall from [7] the ML specifications for the main operators over sorts.

A.1 Sum Sort

Given two sorts s_1 and s_2, we define a new sort $s_1 \otimes s_2$, called the *product (sort) of* s_1 *and* s_2, as follows:

spec SUM_{s_1,s_2}
Imports: SORT
Symbols: $\oplus, \iota_1, \iota_2, \varepsilon_1, \varepsilon_2$
Notations: $s_1 \oplus s_2 \equiv \oplus\, s_1\, s_2$
Axioms:

(Sum Sort)
$s_1 \in \top_{\mathit{Sort}} \wedge s_2 \in \top_{\mathit{Sort}} \to s_1 \oplus s_2 \in \top_{\mathit{Sort}}$

(Inject Left)
$\iota_1 : s_1 \to s_1 \oplus s_2$

(Inject Right)
$\iota_2 : s_2 \to s_1 \oplus s_2$

(Eject Left)
$\varepsilon_1 : s_1 \oplus s_2 \rightharpoonup s_1$

(Eject Right)
$\varepsilon_2 : s_1 \oplus s_2 \rightharpoonup s_2$

(Inverse InjEj1)
$\forall x{:}s_i.\ \varepsilon_i(\iota_i x) = x, \quad i = 1,2$

(Inverse InjEj2)
$\forall x{:}s_{3-i}.\ \varepsilon_i(\iota_{3-i} x) = \bot, \quad i = 1,2$

(CoProduct)
$\forall s_1, s_2 : \top_{\mathit{Sort}}.\ \top_{s_1 \oplus s_2} \subseteq (\iota_1\, \top_{s_1}) \vee (\iota_2\, \top_{s_2})$

endspec

A.2 Pair Sort

spec PAIR_{s_1,s_2} Imports: SORT
Symbols: $Pair, pair, fst, snd$
Notations: $s_1 \otimes s_2 \equiv Pair\ \ s_1\ \ s_2,\ \ \langle \varphi_1, \varphi_2 \rangle \equiv pair\ \ \varphi_1\ \ \varphi_2$
Axioms: // all axioms are quantified by "$\forall s_1, s_2 : \top_{Sort}$"

(Pair Sort)	$(s_1 \otimes s_2) : Sort$
(Pair)	$\forall x_1 : s_1.\ \forall x_2 : s_2.\ \langle x_1, x_2 \rangle : (s_1 \otimes s_2)$
(Pair Fst)	$\forall x_1 : s_1.\ \forall x_2 : s_2.\ fst\ \ \langle x_1, x_2 \rangle = x_1$
(Pair Snd)	$\forall x_1 : s_1.\ \forall x_2 : s_2.\ snd\ \ \langle x_1, x_2 \rangle = x_2$
(Pair Inj)	$\forall x_1, y_1 : s_1.\ \forall x_2, y_2 : s_2.$ $\langle x_1, x_2 \rangle = \langle y_1, y_2 \rangle \rightarrow x_1 = x_2 \wedge y_1 = y_2$
(Pair Domain)	$\top_{s_1 \otimes s_2} = \langle \top_{s_1}, \top_{s_2} \rangle$

endspec

A.3 Function Sort

spec FUN_{s_1,s_2} Imports: SORT
Symbols: $Function$
Notations: $s_1 \ominus s_2 \equiv Function\ \ s_1\ \ s_2$
Axioms: // all axioms are quantified by "$\forall s_1, s_2 : \top_{Sort}$"

(Func Sort)	$(s_1 \ominus s_2) : Sort$
(Func Domain)	$\top_{s_1 \ominus s_2} = \exists f.\ f \wedge \forall x : s_1.\ (f\ x) : s_2$
(Func Ext)	$\forall f, g : s_1 \ominus s_2.\ (\forall x : s_1.\ f\ \ x = g\ \ x) \rightarrow f = g$

endspec

Remark. Note that we do not explicitly define the constructors of function sorts. Instead, we axiomatize the *behaviors* of a function. Indeed, axiom (Func Domain) states that a "function" f of sort $s_1 \ominus s_2$ is one such that for any x of sort s_1, $(f\ x)$ has sort s_2. Axiom (Func Ext) states that two functions f and g of sort $s_1 \ominus s_2$ are equal iff they are behavioral equivalent, i.e., they return the same values on all arguments of sort s_1.

A Parallel Dynamic Epistemic Perspective over Muddy Children Puzzle

Bogdan Macovei
Department of Computer Science
Faculty of Mathematics and Computer Science
University of Bucharest, Academiei 14, 010014 Bucharest, Romania

Research Center for Logic, Optimization and Security (LOS)
Department of Computer Science, Faculty of Mathematics and Computer Science,
University of Bucharest, Academiei 14, 010014 Bucharest, Romania
los.cs.unibuc.ro
bogdan.macovei@fmi.unibuc.ro

Epistemic protocols represents a current field of interest, with numerous approaches still being studied. In this paper we formalize parallel sessions of the The Muddy Children Puzzle using Public Observation Logic, a system that allows epistemic update. We consider agents with roles in multiple sessions and the information update in all parallel sessions as new information is discovered in any particular session.

1 Introduction

In this article, we introduce a formalism to analyze the parallel execution of actions in the epistemic protocol of the *Muddy Children Puzzle*, previously modeled using *public announcement logic* in [4]. We consider agents with roles in multiple sessions and the information update in all parallel sessions as new information is discovered in any particular session. Although the formalism is specifically tailored to model this protocol, it can be generalized since its theoretical framework is based on [5], where a logical system starting from generic actions is defined. Our goal is to investigate how parallel sessions interact and how agents' information is modified not only within a single session but also in a parallel setting.

In the second section, we introduce the *public observation logic* system, based on [5]; in the third section, we model the *Muddy Children Puzzle* protocol using this formalism, starting from the description in [4]; in the fourth section, we construct the parallel sessions and define a method of information propagation between these sessions to maintain consistency in the main parallel model; in the fifth section, we provide an example of such a parallel framework where we study the speed of information propagation and how the number of actions required to reach a result decreases with this update. Finally, we present some conclusions and future directions for further research.

2 Preliminaries

In this preliminary section, we will present the main results from [5], where a new logical system, *public observation logic*, is defined. This system is built upon notions such as *observations* and *expectation models*, which are based on *epistemic models*. These concepts will be used in the formalism we will propose later to define a modeling for the *Muddy Children Puzzle*.

Let AGENT be a finite set of agents, Φ be a set of formulas, with $\Phi_0 \subseteq \Phi$ the set of atomic propositions.

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 56–64, doi:10.4204/EPTCS.389.5

Definition 2.1 (Epistemic model)**.** An *epistemic model* $\mathscr{M}$ is $\mathscr{M} = (W, R, V)$, where W is a non-empty set of worlds, $R \subseteq W^2$ is a binary accessibility relation, $R = \{R_a \mid a \in \text{AGENT}\}$, and $V : \Phi \to \mathscr{P}(W)$ is the valuation function, assigning to each formula the set of worlds where formula is true.

Additionally, the authors of [5] introduce *observation expressions*, as regular expressions over a finite set of actions, named Σ.

Definition 2.2 (Observation expressions)**.** Given a finite set of action symbols Σ, the language $\mathscr{L}_{obs}$ of *observation expressions* is defined by:

$$\pi ::= \delta \mid \varepsilon \mid \pi \cdot \pi \mid \pi + \pi \mid \pi^* \tag{1}$$

where δ is the empty action, with a corresponding empty set $\emptyset$ of observations, the constant ε represents the empty string, and $a \in \Sigma$.

Definition 2.3 (Observations)**.** Given an observation expression π, the corresponding *set of observations*, that is denoted by $\mathscr{L}(\pi)$ is the set of finite string over Σ, defined as follows:

$$\begin{array}{ll} \mathscr{L}(\delta) = \emptyset & \mathscr{L}(\pi_1 \cdot \pi_2) = \{vw \mid v \in \mathscr{L}(\pi_1) \text{ and } w \in \mathscr{L}(\pi_2)\} \\ \mathscr{L}(\varepsilon) = \{\varepsilon\} & \mathscr{L}(\pi_1 \cup \pi_2) = \mathscr{L}(\pi_1) + \mathscr{L}(\pi_2) \\ \mathscr{L}(a) = \{a\} & \mathscr{L}(\pi^*) = \{\varepsilon\} \cup \bigcup_{n>0} \mathscr{L}(\underbrace{\pi;\pi;\ldots;\pi}_{n \text{ times}}) \end{array}$$

Definition 2.4 (Epistemic expectation model)**.** An *epistemic expectation model* $\mathscr{M}_{exp}$ is a quadruple $\mathscr{M}_{exp} = (W, R, V, Exp)$, where (W, R, V) is an epistemic model, also named the *epistemic skeleton* of $\mathscr{M}_{exp}$, and $Exp : W \to \mathscr{L}_{obs}$ is an expected observation function, assigning to each state an observation expression π, such that $\mathscr{L}(\pi) \neq \emptyset$.

With these notions defined, there is introduced a new formalism, named *Public observation logic*, that is a dynamic logic with knowledge operators, made to reason about knowledge via the matching of observations and expectations.

Definition 2.5 (Update by observation)**.** Let α be an observation over Σ, and let $\mathscr{M} = (W, R, V, Exp)$ be an epistemic expectation model. The updated model $\mathscr{M}|_\alpha$ is defined as $\mathscr{M}|_\alpha = (W', R', V', Exp')$, where $W' := \{w \mid \mathscr{L}(Exp(w) - \alpha) \neq \emptyset\}$, $R' = R|_{W'^2}$, $V' = V|_{W'}$, and $Exp'(w) = Exp(w) - \alpha$. Here, $\pi - \alpha$ is defined as $\pi - \alpha = \{\beta \mid \alpha\beta \in \mathscr{L}(\pi)\}$, the right residuation with respect to the monoid $(\Sigma^*, \cdot, \varepsilon)$.

Definition 2.6 (Public observation logic)**.** The formula φ of POL (Public Observation Logic) are given by the following BNF:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_a\varphi \mid [\pi]\varphi \tag{2}$$

where $p \in \Phi_0$ and for any $a \in \text{AGENT}$, K_a is the knowledge operator corresponding to agent a.

Definition 2.7 (Truth definition for POL)**.** Given an epistemic expectation model $\mathscr{M} = (W, R, V, Exp)$, a state $w \in W$ and a POL-formula φ, the truth of φ at w, denoted by $\mathscr{M}, w \models \varphi$ is defined as follows:

$$\mathscr{M}, w \models p \iff w \in V(p) \tag{3}$$

$$\mathscr{M}, w \models \varphi \wedge \psi \iff \mathscr{M}, w \models \varphi \text{ and } \mathscr{M}, w \models \psi \tag{4}$$

$$\mathscr{M}, w \models \neg\varphi \iff \mathscr{M}, w \not\models \varphi \tag{5}$$

$$\mathscr{M}, w \models K_i\varphi \iff \text{for all } v \text{ such that } (w, v) \in R_i \text{ it holds } \mathscr{M}, v \models \varphi \tag{6}$$

$$\mathscr{M}, w \models [\pi]\varphi \iff \text{for all } \alpha \in \mathscr{L}(\pi), \text{ if } \alpha \in \text{init}(Exp(w)), \text{ then } \mathscr{M}|_\alpha, w \models \varphi \tag{7}$$

where $\alpha \in \text{init}(\pi)$ if and only if there is a $\beta \in \Sigma^*$ such that $\alpha\beta \in \mathscr{L}(\pi)$.

3 Muddy Children Puzzle: formalization in POL

The Muddy Children Puzzle represents a scenario where a group of children who have been playing outside is called into their house by their father. Some of the children have mud on their foreheads, others don't, and none of them know their own state. Knowing that they can all see each other, but don't know anything about their own situation, the father initially announces them that at least one child is muddy, then asks them who is certain that they have mud on their forehead. In this scenario, we assume that the children reason correctly from a logical point of view and know how to update their information based on their father's question. If a solution has not been reached yet, the father does nothing else but repeat the request until a final resolution is achieved, where all the muddy children are aware of their own state, so the ones who are not muddy will also know this fact.

We will model the *Muddy Children Puzzle* starting from the description in [4] (where the problem was modeled using *Public announcement logic*), but using the formalism of *Public observation logic*. In this case, instead of updating the model based on formulas, we will update it based on certain actions, and the main action consists of the successive questions that the father addresses.

Let AGENT be a finite set of agents, with $|\text{AGENT}| = n_a$, that we will denote with the first letters of the alphabet, a, b, c, ... during the specification of our protocols. In order to formalize the *Muddy Children Puzzle*, we will define one set of formulas Φ, where $\Phi_0 \subseteq \Phi$ is the set of atomic propositions, and one set of actions Π, with $\Pi_0 \subseteq \Pi$ contains one atomic action, QF, the father's question. Each time when the action is performed, the general knowledge is changed and the agents enrich their individual knowledge. We will number the father's questions to be able to update the model sequentially based on the information obtained at each step.

For any $a \in$ AGENT, Φ_0 contains one atomic proposition m_a, that is true if a is muddy, and false otherwise.

Definition 3.1. Given a finite set of atomic actions, $\Pi_0 = \{\text{QF}\}$, we define the $\mathscr{L}_\Pi$ language by the following BNF grammar:

$$\pi ::= \lambda \mid \text{QF} \mid \pi;\pi \mid \pi \cup \pi \mid \pi^* \tag{8}$$

where λ stands for the empty action, with a corresponding empty set of actions.

We introduce the notation $\text{QF}_i := \underbrace{\text{QF};\ldots;\text{QF}}_{i \text{ times}}$

Definition 3.2. Given an action π, we define the corresponding set of actions $\mathscr{L}(\pi)$ as follows:

$$\begin{array}{ll} \mathscr{L}(\lambda) = \emptyset & \mathscr{L}(\pi_1;\pi_2) = \{vw \mid v \in \mathscr{L}(\pi_1) \text{ and } w \in \mathscr{L}(\pi_2)\} \\ \mathscr{L}(\text{QF}) = \{\text{QF}\} & \mathscr{L}(\pi_1 \cup \pi_2) = \mathscr{L}(\pi_1) \cup \mathscr{L}(\pi_2) \\ & \mathscr{L}(\pi^*) = \{\lambda\} \cup \bigcup_{n>0} \mathscr{L}(\underbrace{\pi;\pi;\ldots;\pi}_{n \text{ times}}) \end{array}$$

In the following, let $\mathscr{M} = (W, R, V, Exp)$ be an epistemic expectations model, where the set of possible worlds W consists of **ordered pairs** $(w_{a_1}, \ldots, w_{n_a}) \in \{0,1\}^{n_a}$. Each w_{a_i} for $a_i \in$ AGENT describe either if a_i is muddy or not. We denote that property by the predicate **muddy**, defined as, for every agent a and for every pair w, $\mathbf{muddy}(a,w) = 0$ if a is not muddy and $\mathbf{muddy}(a,w) = 1$ if a is muddy.

Definition 3.3 (State property)**.** Let $a \in$ AGENT and let $w \in W$, where AGENT $= \{a_1, \ldots, a, \ldots, a_{n_a}\}$, where a is on the p-th position, $0 < p \leq N$. Then $\mathbf{muddy}(a,w) = \mathbf{proj}_p(w)$, where **proj** is the projection function: for a pair $x = (x_1, \ldots, x_n)$, $\mathbf{proj}_i(x)$ returns the value of the i-th component x_i, $0 < i \leq n$.

Definition 3.4 (Model property)**.** Let $a \in$ AGENT. We define $\mathbf{muddy}(a)$ as:

$$\mathbf{muddy}(a) = \begin{cases} 0, & \text{if } \bigcap\{\mathbf{muddy}(a,w) \mid w \in W\} = \{0\} \\ 1, & \text{if } \bigcap\{\mathbf{muddy}(a,w) \mid w \in W\} = \{1\} \\ \text{undefined}, & \text{if } \bigcap\{\mathbf{muddy}(a,w) \mid w \in W\} = \emptyset \end{cases} \tag{9}$$

By abuse of notation, when $\mathbf{muddy}(a)$ is defined and there is no risk for confusion, we will use **muddy** as a predicate.

This predicate can also be defined as:

$$\mathbf{muddy}(a) = 1 \iff \text{for all } w \in W, \mathscr{M}, w \models K_a m_a \tag{10}$$

$$\mathbf{muddy}(a) = 0 \iff \text{for all } w \in W, \mathscr{M}, w \models K_a \neg m_a \tag{11}$$

$$\mathbf{muddy}(a) = \text{undefined} \iff \text{otherwise} \tag{12}$$

Definition 3.5. Let $\mathscr{M} = (W, R, V)$ be an epistemic (Kripke) model, and $Exp : W \to \mathscr{L}_\Pi$ an expectation function that assigns to each state an action π such that $\mathscr{L}(\pi) \neq \emptyset$. Then, $\mathscr{M}_{exp} = (\mathscr{M}, Exp)$ is an epistemic expectations model.

Definition 3.6. Let w be an action over Π and let $\mathscr{M}_{Exp} = (W, R, V, Exp)$ be an expectation model. The updated model is $\mathscr{M}|_\alpha = (W', R', V', Exp')$, where $W' = \{v \mid \mathscr{L}(Exp(v) - \alpha) \neq \emptyset\}$, $R' = R \cap W'^2$, $V' = V|_{W'}$ and for any $v \in W$, $Exp'(v) = Exp(v) - \alpha$.

Each action QF_i, $i \geq 0$, corresponds to the i-th question of the father. For $i = 0$, we have $\mathscr{M}|_{\mathrm{QF}_0} := \mathscr{M}$. Starting from $i = 1$, the states are eliminated sequentially. For $i = 1$, the state $(0, \ldots, 0) \in W^{n_a}$ is eliminated, as QF_1 is the question that announces the existence of at least one muddy child, so $\bigcup_{a \in \mathrm{AGENT}} m_a$ is true: $\mathscr{M}|_{QF_1} = (W' := W - \{0^{n_a}\}, R|_{W'^2}, V|_{W'}, Exp)$, so after the first questions, we have states that contains at least one component that is 1. If we continue this process, we obtain that for $\mathscr{M}|_{QF_2}$ there are removed all the states containing just one muddy agent. In general, for $\mathscr{M}|_{QF_i}$ a set of corresponding worlds with pairs $w \in W_i$ such that w contain at least i values of 1: $\sum_j \mathbf{proj}_j(w) \geq i$. This fact is supported by what happens during the father's questions: he repeats the questions because the agents knows whether the other agents are muddy or not, while none knows their own state, which means that each new question confirms to the agents that there is at least one more muddy agent they must take into account during the inference process. At this point, we have that

$$\mathscr{M}|_{\mathrm{QF}_i} = (W' := W - \{v \in W \mid \sum_j \mathbf{proj}_j(v) < i\}, R|_{W'^2}, V|_{W'}, Exp') \tag{13}$$

Formulas are defined by the following BNF grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid K_a\varphi \mid [\pi]\varphi \tag{14}$$

where $p \in \Phi_0$ is an atomic formula, and $a \in$ AGENT.

The interpretation of formulas is as follows:

$$\mathscr{M}, w \models p \iff w \in V(p) \tag{15}$$

$$\mathscr{M}, w \models \varphi \wedge \psi \iff \mathscr{M}, w \models \varphi \text{ and } \mathscr{M}, w \models \psi \tag{16}$$

$$\mathscr{M}, w \models \neg\varphi \iff \mathscr{M}, w \not\models \varphi \tag{17}$$

$$\mathscr{M}, w \models K_i\varphi \iff \text{for all } v \text{ such that } (w, v) \in R_i \text{ it holds } \mathscr{M}, v \models \varphi \tag{18}$$

$$\mathscr{M}, w \models [\pi]\varphi \iff \text{for all } \alpha \in \mathscr{L}(\pi), \text{ if exists } v \text{ such that } Exp(v) = \alpha; \pi_1; \ldots; \pi_n, \text{ then } \mathscr{M}|_\alpha, w \models \varphi \tag{19}$$

The deductive system contains all instances of propositional tautologies, to which are added the following axioms ([4], [2]):

(A_1)	$K_a(\varphi \to \psi) \to (K_a\varphi \to K_a\psi)$	distributivity of K
(A_2)	$K_a\varphi \to \varphi$	truth
(A_3)	$K_a\varphi \to K_aK_a\varphi$	positive introspection
(A_4)	$\neg K_a\varphi \to K_a\neg K_a\varphi$	negative introspection
(A_5)	$[\pi](\varphi \to \psi) \to ([\pi]\varphi \to [\pi]\psi)$	distributivity of π
(A_6)	$[\pi](\varphi \wedge \psi) \leftrightarrow [\pi]\varphi \wedge [\pi]\psi$	distributivity over conjunction
(A_7)	$[\pi_1;\pi_2]\varphi \leftrightarrow [\pi_1][\pi_2]\varphi$	sequential operator
(A_8)	$[\pi_1 \cup \pi_2]\varphi \leftrightarrow [\pi_1]\varphi \wedge [\pi_2]\psi$	choice
(A_9)	$\varphi \wedge [\pi][\pi^*]\varphi \leftrightarrow [\pi^*]\varphi$	Kleene star
(A_{10})	$\varphi \wedge [\pi^*](\varphi \to [\pi]\varphi) \to [\pi^*]\varphi$	induction axiom

The deduction rules are: MP $\frac{\varphi \quad \varphi\to\psi}{\psi}$ NEC $\frac{\varphi}{K_a\varphi}$ GEN $\frac{\varphi}{[\pi]\varphi}$

Theorem 1. For $n > 1$, there are $n+1$ questions needed to discover n muddy children.

Proof. The proof for this theorem is well-known [1], but we will sketch it for our system.

Let $\mathscr{M} = (W, R, V, Exp)$ be an epistemic model for *Muddy Children Puzzle* and let n be the number of muddy agents. We want to prove that, after $n+1$ questions, every agent knows either if it is muddy or not:

$$\mathscr{M}|_{\mathrm{QF}_{n+1}} \models \bigwedge_{a \in \mathrm{AGENT}} K_a m_a$$

We will prove that by induction over n.

- *base case* $(n = 1) \Rightarrow$ there are 2 father's questions QF needed to discover one muddy agent. After the first one, the state 0^n is removed. After the second question, there is just one muddy agent a_m that sees $n-1$ non-muddy other agents, and with the 0^n state removed, this agent knows that is muddy:

$$\bigwedge_{\substack{a \in \mathrm{AGENT} \\ a \neq a_m}} K_{a_m}\neg m_a \wedge K_a m_{a_m}$$

- *induction step* $\Rightarrow$ there are already n asked questions and, based on that, $n-1$ identified muddy agents. We want to prove that if we add a new muddy agent, there will be one additional question. In this $\mathscr{M}|_{\mathrm{QF}_n}$ model, where is just one world left, $w \in W_n$, such that there are $n-1$ values of 1. If we have an additional agent, which we will represent as the last component of the pair, without losing generality, then we will not be able to distinguish between states $(w,0)$ and $(w,1)$. With a new question, the agent is able to make the inference and to figure wheter it is muddy or not.

□

4 Muddy Children Puzzle with parallel sessions

In the parallel setup, we consider that the puzzle unfolds simultaneously in multiple sessions. Each session is just a regular sequence of actions, so for each group, there is a father who addresses the questions, and the model updates based on these actions. However, there are agents who can be simultaneously present in multiple such groups, which makes them agents playing in multiple sessions simultaneously. In this case, instead of just having a number of independent parallel sessions, the sessions interact with

each other through the knowledge of agents who play simultaneously in at least two sessions. If an agent finds out from one session that they are muddy (or not), they actually possess this information in all other sessions as well.

Let SESSION be a finite set of sessions. For every session $s \in \text{SESSION}$, we have a finite set of agents AGENT_s. Given a finite set of atomic actions on sessions, $\Pi_0 = \{\text{QF}^s\}|_{s\in\text{SESSION}}$, we define the $\mathscr{L}_\Pi$ language by the following BNF grammar:

$$\pi ::= \lambda \mid \text{QF}^s \mid \pi;\pi \mid \pi \cup \pi \mid \pi^* \tag{20}$$

where λ stands for the empty action, $s \in \text{SESSION}$ and $a \in \text{AGENT}_s$.

Definition 4.1 (Session model)**.** A session model $\mathscr{M}_s$ is defined as an expectation model $\mathscr{M}_s$, where $\mathscr{M}_s = (W_s, R_s, V_s, Exp_s)$ and $s \in \text{SESSION}$ is a session; W_s is a finite set of worlds on the session s, $R_s \subseteq W_s \times W_s$ is the binary accessibility relation between worlds, $V_s : \Phi \to W_s$ is the valuation function and $Exp_s : W_s \to \mathscr{L}_\Pi$ the expectation assignment for states.

Definition 4.2 (Parallel model)**.** A parallel model $\mathscr{M}$ is defined as $\mathscr{M} = \mathscr{M}_{s_1} \times \mathscr{M}_{s_2} \times \cdots \times \mathscr{M}_{s_{n_s}} = \bigtimes_{i=1}^{n_s} \mathscr{M}_{s_i}$, where $\text{SESSION} = \{s_1, s_2, \ldots, s_{n_s}\}$, and for an arbitrary $s \in \text{SESSION}$, $\mathscr{M}_s = (W_s, R_s, V_s, Exp_s)$ is a session model.

Suppose that we have a parallel model $\mathscr{M}$ that consists of n_s session models, so $\mathscr{M} = \bigtimes_i \mathscr{M}_i$. We know that, for every model, we have an initial set of worlds $W_s = \{0,1\}^{|\text{AGENT}_s|}$, where 0 means that the property doesn't hold (for an agent a, we have that a is not muddy), and 1 means that the property holds (for an agent a, we have that a is muddy). The $(\text{AGENT}_s)_{s\in\text{SESSION}}$ sets are not mutual disjunctive: for $s_i, s_j \in \text{SESSION}$, $s_i \neq s_j$, is it possible that $\text{AGENT}_{s_i} \cap \text{AGENT}_{s_j} \neq \emptyset$. In that case, imagine that an action α_{s_i} **occurs** in the i-th model, $0 < i \leq |\text{SESSION}|$ (so we have a formula $[\beta_1;\beta_2;\ldots;\alpha_{s_i};\ldots;\beta_n]\varphi$, where $\beta_1, \ldots, \beta_n \in \Pi$ are also actions that occurs in the i-th model, and $\varphi \in \Phi$), that changes the model with respect to updated model definition; we will have that model updated, so $\mathscr{M}_i := \mathscr{M}|_{\alpha_{s_i}}$. But there is the following problem: imagine that in the new $\mathscr{M}_i$ model, we obtained a new set of worlds W_i that, for an arbitrary agent a, we know whether $\mathbf{muddy}(a)$ is either 0 or 1, and that agent also appears in other session models $\mathscr{M}_{k_1}, \mathscr{M}_{k_2}, \ldots, \mathscr{M}_{k_p}$. In that case, we have to also update these models with respect to the new restrictions.

We define $\mathbf{propagate}(\alpha_{s_i}, \mathscr{M}_{s_j}) = (W_j', R_j|_{W_j'^2}, V_j|_{W_j'}, Exp_j)$, where W_j' contains all the worlds from W_j, but without the worlds that doesn't have the same value of the property for the agent a as in $\mathscr{M}_i|_{\alpha_{s_i}}$. If the position for the agent a is i_a and the property value is 0 (1), we will remove from W_j all the worlds that have 1 (0) on the i_a-th position.

Definition 4.3 (Propagate actions)**.** Let $s_i \in \text{SESSION}$ and let α_{s_i} be an action that occurs on the i-th session. We define the propagation on the $\mathscr{M}_{s_j}$ models, ($s_i \neq s_j$ and for any j in this case, $s_j \in \text{SESSION}$) by:

$$\mathbf{propagate}(\alpha_{s_i}, \mathscr{M}_{s_j}) = (W_j', R_j|_{W_j'^2}, V_j|_{W_j'}, Exp_j) \tag{21}$$

where W_j' is defined as:

$$W_j' := W_j - \bigcup_{\substack{a\in\text{AGENT}_{s_i}\cap\text{AGENT}_{s_j} \\ \mathbf{muddy}_{s_i}(a) \text{ is defined}}} \{v \in W_j \mid \mathbf{muddy}_{s_j}(a,v) \neq \mathbf{muddy}_{s_i}(a)\} \tag{22}$$

where $\mathbf{muddy}_{s_j}(a,v)$ is $\mathbf{muddy}(a,v)$ on the j-th session, and $\mathbf{muddy}_{s_i}(a)$ is $\mathbf{muddy}(a)$ on the i-th session.

Definition 4.4 (Updated parallel model)**.** Let $\mathscr{M}$ be a parallel model, and α_{s_i} an action that occurred on the i-th session, $0 < i \leq |\text{SESSION}|$. Then

$$\mathscr{M}|_{\alpha_{s_i}} = \bigtimes_{j<i} \mathbf{propagate}\big(\alpha_{s_i}, \mathscr{M}_{s_j}\big) \times \mathscr{M}|_{\alpha_{s_i}} \times \bigtimes_{i<j} \mathbf{propagate}\big(\alpha_{s_i}, \mathscr{M}_{s_j}\big) \tag{23}$$

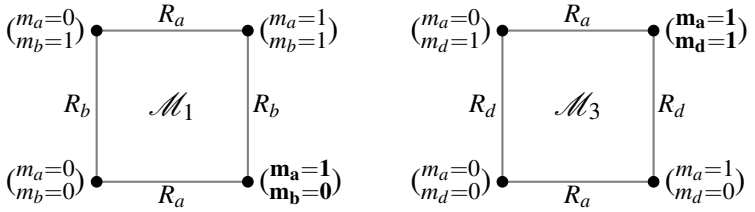
is the updated parallel model.

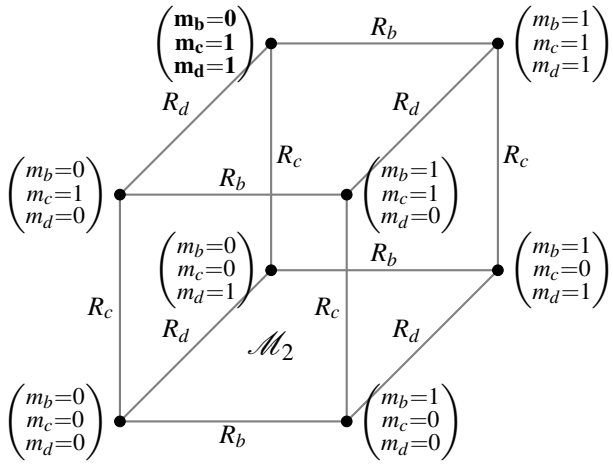
5 Application

In this section, we present a concrete parallel setting, and we study how the session models interact. We have the following setup:

1. $\text{SESSION} = \{s_1, s_2, s_3\}$;
2. $\text{AGENT}_1 = \{a, b\}$, $\text{AGENT}_2 = \{b, c, d\}$ and $\text{AGENT}_3 = \{a, d\}$;
3. $\mathscr{M} = \mathscr{M}_1 \times \mathscr{M}_2 \times \mathscr{M}_3$;
4. suppose that the agents a, c and d are muddy, and b is not, so in $\mathscr{M}$ holds that $\mathbf{muddy}(a) \wedge \neg\mathbf{muddy}(b) \wedge \mathbf{muddy}(c) \wedge \mathbf{muddy}(d)$.

In the following, we represent the models:





Consider the scenario in which we run this models independently, so each agent has its own scope for every scenario. Using the results of Theorem 1, we know that for the first model are 2 questions needed (so $\mathscr{M}_1 \models [QF_2^1](K_a m_a \wedge K_b \neg m_b)$), for the second one there are 3, and for the last one also 3 questions needed. We conclude that if we analyze this scenario in a sequential manner, there are 8 question needed to solve the puzzle. Our target is to see if the number of questions can be minimised if we run these models in parallel.

In the parallel setting, the order of actions is *non-deterministic*. We will analyze a possible situation. Suppose that the first actions that occurs are the father's question for the first model. We already provided that are only two question needed to solve the puzzle, so

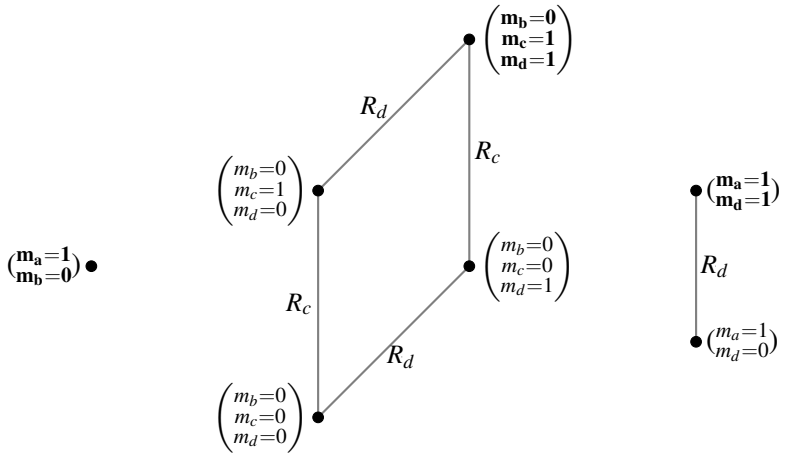
$$\mathscr{M}_1|_{QF_2} = \big(\{(1,0)\}, \emptyset, V_1|_{\{(1,0)\}}, Exp_1'\big) \tag{24}$$

Bearing in mind that we run the model in parallel, instead of just updating the $\mathscr{M}_1$ model, we have to update the entire model, $\mathscr{M}$. In that case, we have to **propagate** the new states, so $\mathscr{M}_2$ and $\mathscr{M}_3$ will be modified by keeping just the states in which m_a holds and m_b does not.

$$\begin{aligned}\textbf{propagate}(\mathrm{QF}_1, \mathscr{M}_2) &= (W_2', R_2' := R_2|_{W_2'}, V_2' := V_2|_{W_2'}, Exp_2) \\ &= (W_2 - \{v \in W_2 \mid \textbf{muddy}_{s_2}(b,v) \neq 0\}, R_2', V_2', Exp_2) \\ &= (W_2' := \{(000),(010),(011),(001)\}, R_2', V_2', Exp_2)\end{aligned}$$

$$\begin{aligned}\textbf{propagate}(\mathrm{QF}_1, \mathscr{M}_3) &= (W_3', R_3' := R_3|_{W_3'}, V_3' := V_3|_{W_3'}, Exp_3) \\ &= (W_3 - \{v \in W_3 \mid \textbf{muddy}_{s_3}(a,v) \neq 1\}, R_3', V_3', Exp_3) \\ &= (\{W_3' := (10),(11)\}, R_3', V_3', Exp_3)\end{aligned}$$

We represent the new $\mathscr{M}|_{QF_1}$ model as:



$\mathscr{M}_1|_{QF_1}$ $\quad$ **propagate**$(\mathrm{QF}_1, \mathscr{M}_2)$ $\quad$ **propagate**$(\mathrm{QF}_1, \mathscr{M}_3)$

Suppose that the next action is father's question in $\mathscr{M}_3$. The agent a already knows that **muddy**(a) holds, so by seeing d can answer the question by **muddy**$(a) \wedge$ **muddy**(d). We also have a **propagation**, that transforms $\mathscr{M}_2$ into a model with two worlds, $W_2 = \{(0,1,1),(0,0,1)\}$. After the father's question, both b and d cand solve the puzzle, by seeing that c is also muddy. In this setting, we only needed four father's questions instead of eight.

Conclusion and Further Work

The formalism for representing parallel models presented in this article, using the *Muddy Children Puzzle* as an example, is based on POL (public observation logic [5]) and builds upon the modeling in PAL (public announcement logic [4]). In POL, observations are seen as strings over an alphabet Σ, which

we replaced in our formalism with actions. For the specific problem we presented, the only actions considered are represented by the father's questions QF. Although our aim was to demonstrate how information can be propagated in parallel sessions for a simple case, this approach can be generalized and applied to the analysis of security protocols.

In [3], we introduced a formalism for analyzing security protocols that use symmetric key cryptography (called DELP, *dynamic epistemic logic for protocols*), also starting from POL but providing a different semantics for actions ($[\pi]\varphi$). An idea for future work is to combine DELP with the method of transmitting information in parallel across multiple sessions to study various types of attacks. This integration could enable a more comprehensive analysis of security protocols, incorporating the dynamics of information updates and potential attacks in a parallel setting. By combining the concepts from DELP and the parallel model, security analysts can investigate how various security protocols perform under different conditions and explore possible vulnerabilities in a more intricate scenario.

References

[1] Nina Gierasimczuk & Jakub Szymanik (2011): *A note on a generalization of the muddy children puzzle*. In: *Proceedings of the 13th Conference on Theoretical Aspects of Rationality and Knowledge*, pp. 257–264, doi:10.1145/2000378.2000409.

[2] David Harel, Dexter Kozen & Jerzy Tiuryn (2001): *Dynamic logic*. *ACM SIGACT News* 32(1), pp. 66–69, doi:10.1145/568438.568456.

[3] Ioana Leuştean & Bogdan Macovei (2021): *DELP: Dynamic Epistemic Logic for Security Protocols*. In: *2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, IEEE, pp. 275–282, doi:10.1109/SYNASC54541.2021.00053.

[4] Hans Van Ditmarsch, Wiebe van Der Hoek & Barteld Kooi (2007): *Dynamic epistemic logic*. 337, Springer Science & Business Media, doi:10.1007/978-1-4020-5839-4.

[5] Hans Van Ditmarsch, Sujata Ghosh, Rineke Verbrugge & Yanjing Wang (2014): *Hidden protocols: Modifying our expectations in an evolving world*. *Artificial Intelligence* 208, pp. 18–40, doi:10.1016/j.artint.2013.12.001.

Enumerating All Maximal Clique-Partitions of an Undirected Graph

Mircea Marin
West University of Timişoara
Timişoara, Romania
mircea.marin@e-uvt.ro

Temur Kutsia Cleo Pau
Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria
kutsia@risc.jku.at ioana.pau@risc.jku.at

Mikheil Rukhaia
Institute of Applied Mathematics
Tbilisi State University
Tbilisi, Georgia
mrukhaia@gmail.com

We address the problem of enumerating all maximal clique-partitions of an undirected graph and present an algorithm based on the observation that every maximal clique-partition can be produced from the maximal clique-cover of the graph by assigning the vertices shared among maximal cliques, to belong to only one clique. This simple algorithm has the following drawbacks: (1) the search space is very large; (2) it finds some clique-partitions which are not maximal; and (3) some clique-partitions are found more than once. We propose two criteria to avoid these drawbacks. The outcome is an algorithm that explores a much smaller search space and guarantees that every maximal clique-partition is computed only once.

The algorithm can be used in problems such as anti-unification with proximity relations or in resource allocation tasks when one looks for several alternative ways to allocate resources.

Acknowledgments. Partially supported by the Austrian Science Fund (FWF) under project P 35530 and by the Shota Rustaveli National Science Foundation of Georgia under project FR-21-16725.

1 Introduction

In this paper, we are interested in computing all maximal clique-partitions in a graph. The original motivation comes from anti-unification with proximity relations. Anti-unification is a well-known technique in computational logic. It was introduced in [13, 14] and was quite intensively investigated in the last years, see, e.g. [1, 2, 11, 10, 6]. Given two first-order logic terms t_1 and t_2, it aims at computing a least general generalization of those terms. That means, one is looking for a term s from which t_1 and t_2 can be obtained by variable substitutions. Such an s is called a generalization of t_1 and t_2. Moreover, there should be no other generalization r of t_1 and t_2, which can be obtained from s by a substitution. For instance, if t_1 and t_2 are the ground terms $f(a,a)$ and $f(b,b)$, then anti-unification computes their least general generalization $f(x,x)$. Replacing variable x by a (resp. by b) in it, one gets $f(a,a)$ (resp. $f(b,b)$). Note that $f(x,y)$ and x are also generalizations of $f(a,a)$ and $f(b,b)$, but they are not least general. Anti-unification has been successfully used in inductive reasoning, inductive logic programming, reasoning and programming by analogy, term set compression, software code clone detection, etc.

In many applications, that can be also relevant for anti-unification, one has to deal with imprecise or vague information. In such circumstances, one tends to consider two objects the same, if they are "sufficiently close" to each other. However, such a proximity relation is not transitive. Nontransitivity

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 65–79, doi:10.4204/EPTCS.389.6

has to be dealt with in a special way. Proximity relations (reflexive symmetric fuzzy binary relations) characterize the notion of 'being close' numerically. They become crisp once we fix the threshold from which on, the distance between the objects can be called 'close'.

Symbolic constraint solving (for unification, matching, and anti-unification constraints) over proximity relations has been studied recently by various authors, e.g., [11, 12, 1, 7, 8]. The approaches can be characterized as class-based and block-based. Considering proximity relations as (weighted) undirected graphs, a proximity class of a vertex is its neighborhood (i.e., the set of vertices to which the current vertex is connected by an edge), while a proximity block is a clique. In the class-based approach to proximity constraint solving, two objects are considered proximal if one of them belongs to the proximity class of another. In the block-based approach, two objects are proximal if they belong to the same *unique* maximal proximity block. The block-based approach is one that is closely related to the subject of this paper. To compute a minimal complete set of generalizations of two first-order logic terms with this approach, one needs to consider all maximal clique-partitions of the graph induced by the proximity relation between constants and between function symbols. For instance, if a is close to both b and c, but b and c are not close to each other, then $f(a,a)$ and $f(b,c)$ have two minimal common generalizations: $f(a,x)$ and $f(x,a)$. In this example, the proximity graph would be $(\{a,b,c\},\{(a,b),(a,c)\})$. It has two maximal clique partitions $\{\{a,b\},\{c\}\}$ and $\{\{a,c\},\{b\}\}$ that tell exactly which symbols should be considered the same. In the first case these are a and b, leading to the generalization $f(a,x)$, and in the second case they are a and c, giving $f(x,a)$. Also, in the block-based approach to approximate unification, one would need to maintain maximal clique-partitions of the proximity graph in order to detect that, e.g., $f(x,x)$ and $f(b,c)$ are not unifiable in the abovementioned proximity relation, see, e.g., [8].

Also, the resource allocation problem, when one looks for several alternative ways to allocate resources, can be an application area of the algorithm considered in this paper.

Whereas the problem of computing all maximal cliques is well studied [4, 16, 15, 5], the problem of computing all maximal clique-partitions became of interest only recently. To the best our knowledge, the only previous study of it is the one reported in 2022 by C. Pau her PhD thesis [12, Sect. 3.3.2]. In this paper we provide a more in-depth analysis of the problem and propose another algorithm which performs better than the one described in [12]. For a given undirected graph G, in order to compute all its maximal clique-partitions, we use a kind of top-down approach. First, we compute the maximal clique cover of G, and the list S of all graph vertices which are shared among maximal cliques. By a systematic enumeration of all possibilities to assign each vertex in S to only one clique where it belongs, we obtain an algorithm that finds all clique-partitions of G in a tree-like search space starting from the maximal clique-cover of G. Our algorithm is optimal in the following sense: (1) It computes only maximal clique-partitions, by avoiding computations below some nodes of the search space which yield non-maximal clique-partitions, and (2) Each maximal clique-partition is computed only once. Moreover, our algorithm computes the maximal clique-partitions incrementally, in the following sense: If one does not want to get all solutions, he/she can stop the algorithm after computing a certain number of solutions. As a result, the computation of maximal clique-partitions can be streamlined with other operations on them.

The paper is structured as follows. In Section 2 we introduce some preliminary notions, the search space $\mathbb{T}^S(G)$ for maximal clique-partitions of an undirected graph G and its main properties. The following two sections describe our main contributions: a criterion to avoid the computation of nonmaximal clique-partitions (Section 3), and a criterion to avoid the redundant computations of the same maximal clique-partition (Section 4). In Section 5 we indicate how to combine these two criteria and define an algorithm to enumerate all maximal clique-partitions of an undirected graph. An analysis of the runtime complexity of our algorithm is performed in Section 7. In the last section we draw some conclusions.

2 Preliminaries

We consider undirected graphs $G = (V, E)$ where V is the set of nodes and E is the set of edges. A **clique** in G is a nonempty subset of V such that every two vertices of it are incident. A clique C is **maximal** if it is not a proper subset of another clique.

A **cover** of G is a finite family $\{V_1, \ldots, V_m\}$ of nonempty sets of nodes such that $\bigcup_{i=1}^{m} V_i = V$. A **clique-cover** of G is a cover $\mathscr{P}$ of V such that every $C \in \mathscr{P}$ is a clique in G. A partition into cliques, or shortly **clique-partition** of G, is a partition $\mathscr{P}$ of V such that every $C \in \mathscr{P}$ is a clique in G. $\mathscr{P}$ is a **maximal clique-partition** of G if $\mathscr{P}$ is a clique-partition of G and $\mathscr{P}$ does not contain two different cliques C, C' such that $C \cup C'$ is a clique.

A graph may have several maximal clique-partitions. In the literature, a problem that was studied intensively is to compute a maximal clique-partition with the smallest number of cliques. Tseng's algorithm [17], introduced to solve this problem, was motivated by its application in the design of processors. Later, Bhasker and Samad [3] proposed two other algorithms. They also derived the upper bound on the number of cliques in a partition and showed that there exists a partition containing a maximal clique of the graph. A problem closely related to clique-partition is the vertex coloring problem, which requires to color the vertices of a graph in such a way that two adjacent vertices have different colors. In fact, a clique-partitioning problem of a graph is equivalent to the coloring problem of its complement graph. Both problems are NP-complete [9].

We write $\mathbb{N}$ for the set of natural numbers starting from 1, and $\mathbb{N}^*$ for the monoid of finite sequences of numbers from $\mathbb{N}$ with the operation of sequence concatenation and neutral element ε. If $n \in \mathbb{N}$ we assume that $[n]$ is the set of natural numbers k such that $1 \leq k \leq n$.

From now on we assume that $G = (V, E)$ is an undirected graph for which we know:

1. An enumeration $cfg_0 := [\overline{C}_1, \ldots, \overline{C}_m]$ of all maximal cliques of G. We denote the maximal cliques of G with identifiers with an overbar. The value m indicates the number of maximal cliques of graph G.
2. For every vertex $v \in V$ and set of nodes $C \subseteq V$ we define:
   - $cliques(v) := \{i \mid v \in \overline{C}_i\}$, and $d(v) := |cliques(v)|$,
   - $cliques(C) := \bigcap_{v \in C} cliques(v)$ for every set of nodes $C \subseteq V$.
3. An enumeration $S := [v_1, \ldots, v_s]$ of all nodes $v \in V$ with $d(v) > 1$.
4. $Rgd := \{k \in [m] \mid$ there is a vertex $v \in V$ with $cliques(v) = \{k\}\}$.

$cliques(v)$ is the set of indices of maximal cliques where node v belongs, and $cliques(C)$ is the set of indices of maximal cliques which contain C. Thus, a nonempty set of nodes C is not a clique iff $cliques(C) = \emptyset$. The nodes $v \in S$ are those that belong to more than one maximal clique. To transform the maximal clique cover $\{\overline{C}_1, \ldots, \overline{C}_m\}$ into a clique partition, we must assign every node $v \in S$ to belong to only one clique. To formalize this process, we introduce a couple of auxiliary notions.

A **configuration** is an enumeration $[C_1, \ldots, C_m]$ of empty sets or cliques of G, such that $\bigcup_{k=1}^{m} C_k = V$ and $C_i \subseteq \overline{C}_i$ for all $1 \leq i \leq m$. In particular, $cfg_0 = [\overline{C}_1, \ldots, \overline{C}_m]$ is a configuration. Every configuration $cfg = [C_1, \ldots, C_m]$ represents a clique cover denoted by

$$repr(cfg) := \{C_i \mid 1 \leq i \leq m \text{ and } C_i \neq \emptyset\}.$$

We distinguish two sets of configurations of interest: the set $\Pi_{cp}(G)$ of configurations cfg for which $repr(cfg)$ is a clique partition of G; and the set $\Pi_{mcp}(G)$ is the set of configurations cfg for which $repr(cfg)$ is a maximal clique partition of G.

Lemma 1. *Every maximal clique-partition of G is represented by a configuration in $\Pi_{mcp}(G)$.*

Proof. Let $\mathscr{P}$ be a maximal clique-partition of G. Then for every $C \in \mathscr{P}$ there exists a maximal clique $\varphi(C)$ such that $C \subseteq \varphi(C)$. If $C_1, C_2 \in \mathscr{P}$ and $\varphi(C_1) = \varphi(C_2)$ then $C_1 \cup C_2 \subseteq \varphi(C_1)$ is a clique, and the maximality of $\mathscr{P}$ implies $C_1 = C_2$. Thus φ is injective and we can define $cfg = [C_1, \ldots, C_m]$ by

$$C_k := \begin{cases} C & \text{if } C \in \mathscr{P} \text{ and } \varphi(C) = \overline{C}_k, \\ \emptyset & \text{otherwise} \end{cases}$$

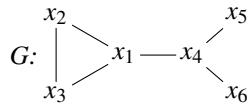
for all $k \in [m]$. Then $repr(cfg) = \mathscr{P}$ because φ is injective. Thus $cfg \in \Pi_{mcp}(G)$. □

For every node $v \in V$ we define the relation $[C_1, \ldots, C_m] \to_{(v,i)} [C'_1, \ldots, C'_m]$ to hold if $v \in C_i$ for some $1 \le i \le m$, $C'_i = C_i$ and $C'_j = C_j - \{v\}$ for all $j \in [m] - \{i\}$. This relation corresponds to the decision to assign node v_i to the i-th clique of the configuration.

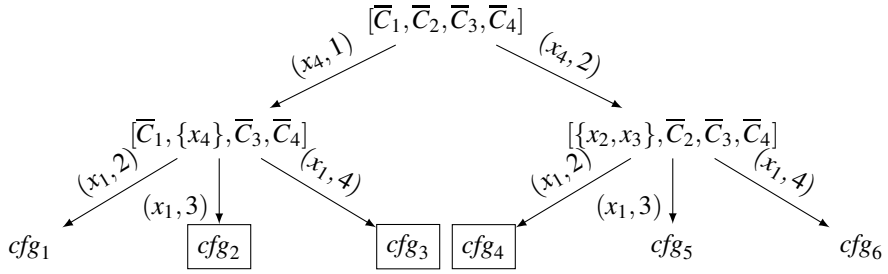
2.1 The search space $\mathbb{T}^S(G)$

It is easy to see that, if $S = [v_1, v_2, \ldots, v_s]$ and $i_1 \in cliques(v_1), i_2 \in cliques(v_2), \ldots, i_s \in cliques(v_s)$ then $cfg_0 = [\overline{C}_0, \ldots, \overline{C}_m] \to_{(v_1,i_1)} cfg_1 \to_{(v_2,i_2)} \cdots \to_{(v_s,i_s)} cfg_s$ is a sequence of decision steps that ends with a configuration whose representation is a partition of G.

We let $\mathbb{T}^S(G)$ be the tree with root cfg_0 and edges $cfg \to_{(v,i)} cfg'$ which correspond to the decision to keep the shared node $v \in S$ in the i-th component of the configuration. We will use $\mathbb{T}^S(G)$ as the search space for maximal clique-partitions, and let $Leaf(\mathbb{T}^S(G))$ be the set of leaf configurations in $\mathbb{T}^S(G)$.

Example 1. *The simple graph* G: 

has four maximal cliques: $\overline{C}_1 = \{x_1, x_2, x_3\}$, $\overline{C}_2 = \{x_1, x_4\}$, $\overline{C}_3 = \{x_4, x_5\}$, $\overline{C}_3 = \{x_4, x_6\}$ and two nodes shared among maximal cliques: $S = [v_1, v_2]$ where $v_1 = x_4$, $v_2 = x_1$. In this example we have $cliques(x_1) = \{1, 2\}$, $cliques(x_2) = \{2, 3, 4\}$. The exhaustive search space for maximal clique partitions is the tree $\mathbb{T}^S(G)$ depicted below:



where the leaf configurations are

$cfg_1 = [\overline{C}_1, \{x_4\}, \{x_5\}, \{x_6\}]$ *with* $repr(cfg_1) = \{\overline{C}_1, \{x_4\}, \{x_5\}, \{x_6\}\}$.
$cfg_2 = [\overline{C}_1, \emptyset, \overline{C}_3, \{x_6\}]$ *with* $repr(cfg_2) = \{\overline{C}_1, \overline{C}_3, \{x_6\}\}$.
$cfg_3 = [\overline{C}_1, \emptyset, \{x_5\}, \overline{C}_4]$ *with* $repr(cfg_3) = \{\overline{C}_1, \{x_5\}, \overline{C}_4\}$.
$cfg_4 = [\{x_2, x_3\}, \overline{C}_2, \{x_5\}, \{x_6\}]$ *with* $repr(cfg_4) = \{\{x_2, x_3\}, \overline{C}_2, \{x_5\}, \{x_6\}\}$.
$cfg_5 = [\{x_2, x_3\}, \{x_1\}, C_3, \{x_6\}]$ *with* $repr(cfg_5) = \{\{x_2, x_3\}, \{x_1\}, C_3, \{x_6\}\}$.
$cfg_6 = [\{x_2, x_3\}, \{x_1\}, \{x_5\}, C_4]$ *with* $repr(cfg_6) = \{\{x_2, x_3\}, \{x_1\}, \{x_5\}, C_4\}$.

Only the final configurations cfg_2, cfg_3, cfg_4 represent maximal clique-partitions of G:
$\mathscr{P}_1 = \{\overline{C}_1, \overline{C}_3, \{x_6\}\} = repr(cfg_2)$,
$\mathscr{P}_2 = \{\overline{C}_1, \{x_5\}, \overline{C}_4\} = repr(cfg_3)$, *and*
$\mathscr{P}_3 = \{\{x_2, x_3\}, \overline{C}_2, \{x_5\}, \{x_6\}\} = repr(cfg_4)$.
The other final configurations in $\mathbb{T}^S(G)$ represent non-maximal clique-partitions of G. □

2.1.1 Properties of the search space $\mathbb{T}^S(G)$

The following are immediate consequences of the definition: if $S = [v_1, v_2, \ldots, v_s]$ is the list of nodes shared among the maximal cliques of G then

1. $\mathbb{T}^S(G)$ has depth s, and all its leaf configurations occur at depth s.
2. Every internal configuration at depth $\ell < s$ in $\mathbb{T}^S(G)$ has $d(v_\ell)$ children.

For every configuration *cfg* in $\mathbb{T}^S(G)$ there is a unique path

$$cfg_0 \rightarrow_{(v_1,i_1)} cfg_1 \rightarrow_{(v_2,i_2)} \cdots \rightarrow_{(v_\ell,i_\ell)} cfg$$

from the root configuration to *cfg*. We let $\delta(cfg) := [i_1, \ldots, i_\ell]$ be the sequence of assignment decisions made for the shared nodes $v_1, \ldots, v_\ell \in S$.

Lemma 2. *If $cfg \in \mathbb{T}^S(G)$ with $\delta(cfg) = [i_1, \ldots, i_\ell]$ then all descendants $[C_1, \ldots, C_m]$ of cfg in $\mathbb{T}^S(G)$, including cfg, have $C_i \neq \emptyset$ for every $i \in Rgd \cup \{i_1, \ldots, i_\ell\}$.*

Proof. If $i = i_p \in \{i_1, \ldots, i_l\}$ then the shared node $v_p \in S$ was assigned to the clique with index i, thus $C_i \neq \emptyset$ because $v_p \in C_i$. If $i \in Rgd$ then $\overline{C}_i$ has a node v with $cliques(v) = \{i\}$. Node v persists in the i-th component of all configurations in $\mathbb{T}^S(G)$. In particular, $C_i \neq \emptyset$ because $c \in C_i$. □

Lemma 3. $\Pi_{mcp}(G) \subseteq Leaf(\mathbb{T}^S(G))$.

Proof. Let $C = [C_1, \ldots, C_m] \in \Pi_{mcp}(G)$. For every $v \in S$, there is a unique $\kappa(v) \in [m]$ such that $v \in C_{\kappa(v)}$. Then $cfg_0 \rightarrow_{(v_1,\kappa(v_1))} cfg_1 \rightarrow_{(v_2,\kappa(v_2))} \cdots \rightarrow_{(v_s,\kappa(v_s))} cfg_s$ is a valid sequence of decision steps. Moreover, cfg_s is a final configuration in $\mathbb{T}^S(G)$ and $repr(cfg_s) = \mathscr{P}$. □

Corollary 1. *Every maximal clique-partition is represented by a configuration in $\mathbb{T}^S(G)$.*

Proof. Immediate consequence of Lemmas 1 and 3. □

From these preliminary results, we derive the following algorithm to find all clique-partitions of G: we traverse systematically (e.g., in a depth-first manner) the search space $\mathbb{T}^S(G)$, and for every final configuration *cfg* in $\mathbb{T}^S(G)$ we check if $repr(cfg)$ is a maximal clique-partition of G. This method has the following drawbacks:

1. The search space can be huge, with many final configurations for non-maximal clique-partitions. For instance, in Example 1, the final configurations cfg_1, cfg_6 and cfg_6 represent non-maximal clique partitions.
2. Some maximal clique-partitions may be represented by more than one final configuration.

We wish to prune the search space as much as possible, to eliminate the computation of configurations for non-maximal clique-partitions, and to ensure the computation of exactly one configuration for every maximal clique-partition.

3 Avoiding the computation of nonmaximal clique-partitions

A final configuration $[C_1,\ldots,C_m]$ does not represent a maximal clique-partition if there exist $1 \le i \ne j \le m$ such that $C_i \ne \emptyset \ne C_j$ and $C_i \cup C_j$ is a clique. Therefore, it is useful to detect and stop searching below nodes of $\mathbb{T}^S(G)$ from where we reach only such final configurations.

Definition 1. *We say that a configuration $cfg = [C_1,\ldots,C_m] \in \mathbb{T}^S(G)$ with $\delta(cfg) = [i_1,\ldots,i_\ell]$ is a $T1$-**node**, or that it has property $T1$, if either*

(a) *$cliques(C_a) \cap Rgd \ne \emptyset$ for some clique index $a \in \{i_1,\ldots,i_\ell\} - Rgd$, or*

(b) *$cliques(C_a) \cap cliques(C_b) \ne \emptyset$ for distinct clique indices $a,b \in \{i_1,\ldots,i_\ell\}$.*

We let $\mathbb{T}^S_{!(T_1)}(G)$ be the result of pruning from $\mathbb{T}^S(G)$ all subtrees whose root is a $T1$-node.

Proposition 1. *If cfg is a $T1$-node of $\mathbb{T}^S(G)$ then there is no final configuration $[C_1,\ldots,C_m]$ below or equal to cfg such that $repr(cfg)$ is a maximal clique-partition.*

Proof. Let $[C_1,\ldots,C_m]$ be a final configuration below or equal to cfg in $\mathbb{T}^S(G)$.

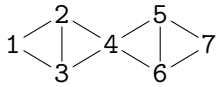
(a) If $cliques(C_a) \cap Rgd \ne \emptyset$ for some clique index $a \in \{i_1,\ldots,i_\ell\} - Rgd$ then $C_a \ne \emptyset$ by Lemma 2, and there exists $b \in Rgd$ such that $C_a \subseteq \overline{C}_b$. In this case we have: (1) $b \ne a$ because $b \in Rgd$ and $a \notin Rgd$; (b) $C_b \ne \emptyset$ by Lemma 2; and (3) $C_a \cup C_b$ is a clique included in $\overline{C}_b$ because $C_a \subseteq \overline{C}_b$ and $C_b \subseteq \overline{C}_b$. Therefore, $\mathscr{P}$ is not a maximal clique-partition.

(b) If $cliques(C_a) \cap cliques(C_b) \ne \emptyset$ for distinct $a,b \in \{i_1,\ldots,i_\ell\}$ then $C_a \ne \emptyset \ne C_b$ by Lemma 2, and there exists $p \in cliques(C_a) \cap cliques(C_b)$ such that $C_a \subseteq \overline{C}_p$ and $C_b \subseteq \overline{C}_p$. Thus $C_a \cup C_b \subseteq \overline{C}_p$, hence $C_a \cup C_b$ is a clique and $\mathscr{P}$ is not maximal clique-partition. □

Corollary 2. *$Leaf(\mathbb{T}^S_{!(T_1)}(G)) = \Pi_{mcp}(G)$.*

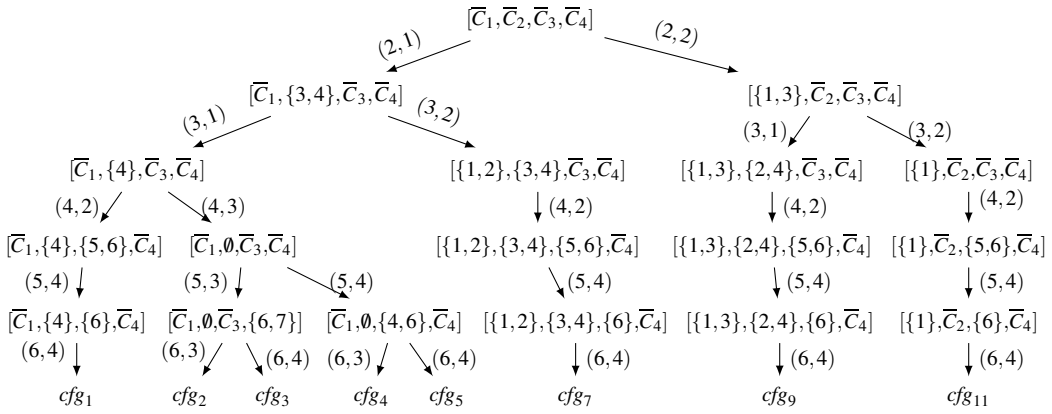
Proof. Immediate consequence of Lemma 3, Proposition 1 and the obvious observation that every $cfg \in \mathbb{T}^S(G)$ with $repr(cfg)$ non-maximal is not in $\mathbb{T}^S_{!(T1)}(G)$ because it is a $T1$-node. □

Example 2. *An enumeration of the maximal cliques of the undirected graph G:*



is $cfg_0 := [\overline{C}_1,\overline{C}_2,\overline{C}_3,\overline{C}_4]$ where $\overline{C}_1 = \{1,2,3\}$, $\overline{C}_2 = \{2,3,4\}$, $\overline{C}_3 = \{4,5,6\}$, $\overline{C}_4 = \{5,6,7\}$. Then $d[1] = d[7] = 1$ and $d[v] = 2$ for all vertices $v \in \{2,3,4,5,6\}$. Therefore $Rgd = \{1,4\}$ and we can choose $S = [2,3,4,5,6]$. The tree $\mathbb{T}^S(G)$ has $\sum_{i=1}^{5}\prod_{j=1}^{i} 2 = 62$ non-root configurations and $2^5 = 32$ final configurations, whereas $\mathbb{T}^S_{!(T1)}(G)$ has 25 non-root configurations, as shown in Fig. 1. The final configurations in $\mathbb{T}^S_{!(T1)}(G)$ are $cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_7, cfg_9, cfg_{11}$, and

$$\begin{aligned} repr(cfg_1) = repr(cfg_5) &= \{\{1,2,3\},\{4,5,6\},\{7\}\}, & repr(cfg_2) &= \{\{1,2,3\},\{4,5,6\},\{7\}\},\\ repr(cfg_3) &= \{\{1,2,3\},\{4,5\},\{6,7\}\}, & repr(cfg_4) &= \{\{1,2,3\},\{4,6\},\{5,7\}\},\\ repr(cfg_7) &= \{\{1,2\},\{3,4\},\{5,6,7\}\}, & repr(cfg_9) &= \{\{1,3\},\{2,4\},\{5,6,7\}\},\\ repr(cfg_{11}) &= \{\{1\},\{2,3,4\},\{5,6,7\}\}. \end{aligned}$$



where $cfg_1 = [\overline{C}_1,\{4\},\emptyset,\overline{C}_4]$, $cfg_2 = [\overline{C}_1,\emptyset,\overline{C}_3,\{7\}]$, $cfg_3 = [\overline{C}_1,\emptyset,\{4,5\},\{6,7\}]$, $cfg_4 = [\overline{C}_1,\emptyset,\{4,6\},\{5,7\}]$, $cfg_5 = [\overline{C}_1,\emptyset,\{4\},\overline{C}_4]$, $cfg_6 = [\{1,2\},\{3\},\overline{C}_3,\overline{C}_4]$, $cfg_7 = [\{1,2\},\{3,4\},\emptyset,\overline{C}_4]$, $cfg_8 = \{\{1,3\},\{2\},\overline{C}_3,,\overline{C}_4\}$, $cfg_9 = [\{1,3\},\{2,4\},\emptyset,\overline{C}_4]$, $cfg_{10} = [\{1\},\{2,3\},\overline{C}_3,\overline{C}_4]$, $cfg_{11} = [\{1\},\overline{C}_2,\emptyset,\overline{C}_4]$.

Figure 1: Search tree $\mathbb{T}^S_1(G)$ for the undirected graph from Example 2.

4 Avoiding repeated computations of the same maximal clique-partition

In Example 2, the final configurations $cfg_1 := [\overline{C}_1,\{4\},\emptyset,\overline{C}_4]$ and $cfg_5 := [\overline{C}_1,\emptyset,\{4\},\overline{C}_4]$ represent the same maximal clique-partition: $repr(cfg_1) = repr(cfg_5) = \{\{1,2,3\},\{4\},\{5,6,7\}\}$. Thus, there are situations when the search space $\mathbb{T}^S_{!(T1)}(G)$ has the following undesirable feature: different final configurations represents the same maximal clique-partition. This implies that some computations are redundant: some maximal clique-partitions will be generated more than once.

Note that, in Example 2, the configurations cfg_1 and cfg_5 which represent the same maximal clique-partition $\mathscr{P} = \{\overline{C}_1,\{4\},\overline{C}_4\}$ have the following property: $cfg_1 = [C_1,\ldots,C_m]$, $cfg_5 = [C'_1,\ldots,C'_m]$ and there exist $1 \leq j \neq i \leq m$ such that $C_i = C'_j \neq \emptyset$ and $C'_i = \emptyset = C_j$. The following lemma indicates that this is a general property of configurations which represent the same maximal clique-partition:

Lemma 4. *If the distinct final configurations $cfg = [C_1,\ldots,C_m], cfg' = [C'_1,\ldots,C'_m]$ in $\mathbb{T}^S_{!(T1)}(G)$ have $repr(cfg) = repr(cfg')$ then there exist $1 \leq j \neq i \leq m$ such that $C_i = C'_j \neq \emptyset$ and $C'_i = C_j = \emptyset$.*

Proof. Let $I := \{k \mid 1 \leq k \leq m \text{ and } C_k \neq \emptyset\}$. Then $repr(cfg) = \{C_k \mid k \in I\}$. Moreover,

- $repr(cfg) = repr(cfg')$ implies the existence of a permutation $\pi : \{1,\ldots,m\} \to \{1,\ldots,m\}$ such that $C'_k = C_{\pi(k)}$ for all $k \in [m]$,
- $cfg \neq cfg'$ implies the existence of $i \in I$ such that $C_i \neq C'_i$. This implies $i \neq \pi(i)$.

Let $j = \pi^{-1}(i)$. Then $j \neq i$ and $C_i = C_{\pi(j)} = C'_j$. Since $i \in I$, we have $C_i = C'_j \neq \emptyset$.

Note that $j \neq \pi(j)$ because $j = \pi(j)$ implies $j = \pi(\pi^{-1}(i)) = i$, contradiction.

It remains to prove that $C'_i = C_j = \emptyset$. From $\emptyset \neq C_i \subseteq \overline{C}_i$ and $C_{\pi(i)} = C'_i \subseteq \overline{C}_i$ we learn that $C_i \cup C_{\pi(i)}$ is a clique included in $\overline{C}_i$. We must have $C'_i = C_{\pi(i)} = \emptyset$ because otherwise C_i and $C_{\pi(i)}$ would be different cliques of $repr(cfg)$ with $C_j \cup C_{\pi(j)}$ a clique, which contradicts the assumption that $repr(cfg)$ is maximal clique-partition.

From $\emptyset \neq C'_j = C_{\pi(j)} \subseteq \overline{C}_j$ and $C_j \subseteq \overline{C}_j$ we learn that $C_{\pi(j)} \cup C_j$ is a clique included in $\overline{C}_j$. We must have $C_j = \emptyset$ because otherwise C_j and $C_{\pi(j)}$ would be different cliques of $repr(cfg)$ with $C_j \cup C_{\pi(j)}$ a clique, which contradicts the assumption that $repr(cfg)$ is maximal clique-partition. □

Lemma 4 allows us to define a criterion to eliminate from $\mathbb{T}^S_{!(T1)}(G)$ the redundant computations of maximal clique-partitions.

Definition 2. *We say that configuration $cfg = [C_1,\ldots,C_m] \in \mathbb{T}^S_{!(T1)}(G)$ with $\delta(cfg) = [i_1,\ldots,i_\ell]$ is a* $T2$**-node**, *or that it has property $T2$, if there exist $1 \le j < i \le m$ such that $j \in \{i_1,\ldots,i_\ell\} - Rgd$ and $i \in cliques(C_j)$. We let $\mathbb{T}^S_{!(T_1,T_2)}(G)$ be be the result of pruning from $\mathbb{T}^S_{!(T1)}(G)$ all subtrees whose root is a $T2$-node.*

From now on we write $Leaf(\mathbb{T}_{!(T1,T2)})^S(G))$ for the set of configurations in $\mathbb{T}^S_{!(T1,T2)}(G)$ at depth s. If we let $\Pi^!_{mcp}(G)$ be the set of configurations $[C_1,\ldots,C_m]$ from $\Pi_{mcp}(G)$ such that

$$\text{for all } 1 \le j < i \le m, \text{ if } C_j \neq \emptyset \text{ then } i \notin cliques(C_j)$$

then the following lemmas hold:

Lemma 5. *For every maximal clique-partition $\mathscr{P}$ of G there is exactly one configuration $cfg \in \Pi^!_{mcp}(G)$ with $repr(cfg) = \mathscr{P}$.*

Proof. For every configuration $cfg = [C_1,\ldots,C_m] \in Leaf(\mathbb{T}^S_{!(T1)}(G))$ we define the measure

$$m(cfg) := \sum_{\substack{1\le i\le m\\ C_i=\emptyset}} i.$$

First, we prove that $\Pi^!_{mcp}(G)$ has at least one configuration whose representation is $\mathscr{P}$. By Lemma 1, there exists $cfg'_0 = [C_1,\ldots,C_m] \in \Pi_{mcp}(G)$ with $repr(cfg) = \mathscr{P}$. If $cfg \in \Pi^!_{mcp}(G)$ then we are done. Otherwise there exists $1 \le j < i \le m$ such that $\emptyset \neq C_j \subseteq \overline{C}_i$. Then $C_i = \emptyset$ because otherwise C_i, C_j would be different components of $\mathscr{P}$ with $C_i \cup C_j \subseteq \overline{C}_i$ and this contradicts the assumption that the clique-partition $\mathscr{P}$ is maximal. Thus, we can define the configuration $cfg'_1 = [C'_1,\ldots,C'_m] \in \Pi_{mcp}(G)$ with

$$C'_k := \begin{cases} C_j & \text{if } k = i,\\ \emptyset & \text{if } k = j,\\ C_k & \text{otherwise} \end{cases}$$

If follows that $m(cfg'_0) - m(cfg'_1) = i - j > 0$. In this way we can build a sequence of configurations $cfg'_0, cfg'_1, \ldots \in \Pi_{mcp}(G)$ with $\mathscr{P} = repr(cfg'_0) = repr(cfg'_1) = \ldots$ and $m(cfg'_0) > m(cfg'_1) > \ldots$ Since the ordering $>$ on natural numbers is well-founded, this construction will eventually end with a configuration $cfg'_p \in \Pi^!_{mcp}(G)$ and $repr(cfg'_p) = \mathscr{P}$. Hence $\Pi^!_{mcp}(G)$ has at least one configuration whose representation is $\mathscr{P}$.

It remains to show that there are no two configurations $cfg = [C_1,\ldots,C_m], cfg' = [C'_1,\ldots,C'_m] \in \Pi^!_{mcp}(G)$ with $repr(cfg) = repr(cfg')$. If this were the case then, by Lemma 4, there exist $1 \le j \neq i \le m$ such that $C_i = C'_j \neq \emptyset$ and $C'_i = C_j = \emptyset$. If $j < i$ then $\emptyset \neq C'_j = C_i \subseteq \overline{C}_i$, which contradicts the assumption that $cfg' \in \Pi^!_{mcp}(G)$. If $j > i$ then $\emptyset \neq C_i = C'_j \subseteq \overline{C}_j$, which contradicts the assumption that $cfg \in \Pi^!_{mcp}(G)$. □

Lemma 6. $Leaf(\mathbb{T}^S_{!(T1,T2)}(G)) = \Pi^!_{mcp}(G)$.

Proof. First, we prove that $Leaf(\mathbb{T}^S_{!(T1,T2)}(G)) \subseteq \Pi^!_{mcp}(G)$. Let $cfg = [C_1,\ldots,C_m] \in Leaf(\mathbb{T}^S_{!(T1,T2)}(G))$ with $\delta(cfg) = [i_1,\ldots,i_s]$. Since $Leaf(\mathbb{T}^S_{!(T1,T2)}(G)) \subseteq Leaf(\mathbb{T}^S_{!(T1)}(G)) = \Pi_{mcp}(G)$, we only have to show that, for all $1 \le j < i \le m$, if $C_j \neq \emptyset$ then $i \notin cliques(C_j)$. If this were not the case, then there exist

$j \in \{i_1,\ldots,i_s\} \cup Rgd$ and $j < i \leq m$ such that $i \in cliques(C_j)$. We observe that $j \notin Rgd$ because otherwise $cliques(C_j) = \{j\}$, which contradicts the assumption $i \in cliques(C_j)$. This implies that $[C_1,\ldots,C_m]$ is $T2$-node of $\mathbb{T}^S_{!(T1)}(G)$, which contradicts the assumption that $[C_1,\ldots,C_m] \in Leaf(\mathbb{T}^S_{!(T1,T2)}(G))$.

To finish the proof, we must show that $\Pi^!_{mcp}(G) \subseteq Leaf(\mathbb{T}^S_{!(T1,T2)}(G))$. Since

$$\Pi^!_{mcp}(G) \subseteq \Pi_{mcp}(G) = Leaf(\mathbb{T}^S_{!(T1)}(G)),$$

it is sufficient to prove that every configuration $cfg \in \Pi_{mcp}(G) - \Pi^!_{mcp}(G)$ is below a $T2$-node of $\mathbb{T}^S_{!(T1)}(G)$. Let $cfg = [C_1,\ldots,C_m]$ with $\delta(cfg) = [i_1,\ldots,i_s]$. Then there exist $1 \leq j < i < m$ such that $\emptyset \neq C_j \subseteq \overline{C}_i$. We observe that $j \notin Rgd$ because otherwise the only maximal clique which contains C_j is $\overline{C}_j$. Therefore we must have $j \in \{i_1,\ldots,i_s\} - Rgd$. This implies that cfg is $T2$-node. □

The following corollary is an immediate consequence of the previous two lemmas.

Corollary 3. *Every maximal clique-partition is produced by a single configuration from* $Leaf(\mathbb{T}^S_{!(T1,T2)}(G))$.

Example 3. *The tree* $\mathbb{T}^S_{!(T1,T2)}(G)$ *for the graph G from Example 2 is shown in Figure 2.* $[\overline{C}_1,\{4\},\{5,6\},\overline{C}_4] \in \mathbb{T}^S_{!(T1)}(G)$ *is a* $T2$*-node because it is of the form* $[C_1,C_2,C_3,C_4]$ *and there exist* $j = 2 < 3 = i$ *such that* $j \in \{2,3\} - Rgd$ *and* $C_2 \subseteq \overline{C}_3$. *Compared to* $\mathbb{T}^S_{!(T1)}(G)$*, the total number of non-root nodes in* $\mathbb{T}^S_{!(T1,T2)}(G)$ *has dropped from 25 to 22, and*

$$Leaf(\mathbb{T}^S_{!(T1,T2)}(G)) = \Pi^!_{mcp}(G) = \{cfg_2, cfg_3, cfg_4, cfg_5, cfg_7, cfg_9, cfg_{11}\}.$$

Every maximal clique-partition is produced by a single final configuration in $\mathbb{T}^S_{!(T1,T2)}(G)$*):*

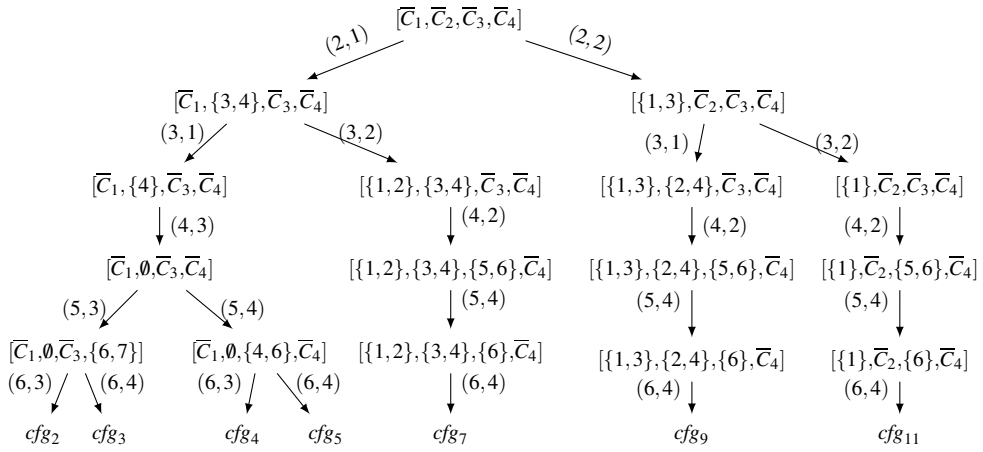
$$\begin{array}{ll} repr(cfg_2) = \{\{1,2,3\},\{4,5,6\},\{7\}\}, & repr(cfg_3) = \{\{1,2,3\},\{4,5\},\{6,7\}\}, \\ repr(cfg_4) = \{\{1,2,3\},\{4,6\},\{5,7\}\}, & repr(cfg_5) = \{\{1,2,3\},\{4,5,6\},\{7\}\}, \\ repr(cfg_7) = \{\{1,2\},\{3,4\},\{5,6,7\}\}, & repr(cfg_9) = \{\{1,3\},\{2,4\},\{5,6,7\}\}, \\ repr(cfg_{11}) = \{\{1\},\{2,3,4\},\{5,6,7\}\}. & \end{array}$$

5 The combined detection of $T1$-nodes and $T2$-nodes

Let $cfg \rightarrow_{(v_\ell,i_\ell)} cfg'$ be a branch of $\mathbb{T}^S_{!(T1,T2)}(G)$, and $\delta(cfg') = [i_1,\ldots,i_\ell]$. The only visible change from cfg to cfg' is that of the components with indices from the set $cliques(v_\ell) - \{i_\ell\}$. When we are about to check if cfg' has property $T1$ or $T2$, we know that cfg does not have property $T1$ or $T2$ because it has already passed this test. Therefore, it is sufficient to consider only the set of clique indices $\mathscr{J}_\ell := cliques(v_\ell) \cap \mathscr{I}_\ell$ where $\mathscr{I}_\ell := \{i_1,\ldots,i_\ell\} - Rgd$, and to check if cfg' is a configuration $[C_1,\ldots,C_m]$ which satisfies one of the following conditions for some $j \in \mathscr{J}_\ell$:

1. (a) $C_j \subseteq \overline{C}_i$ for an $i \in Rgd$ (in this case cfg' is $T1$-node), or (b) $i > j$ (in this case cfg' is $T2$-node); or
2. there exists $i \in \mathscr{I}_\ell - \{j\}$ such that $C_i \cup C_j$ is a clique. In this case cfg' is $T1$-node.

Condition 1.(a) is equivalent with $cliques(C_j) \cap Rgd \neq \emptyset$, and condition 2 is equivalent with $cliques(C_i) \cap cliques(C_j) \neq \emptyset$.



The configurations cfg_i for $2 \leq i \leq 11$ are the same as those from Figure 1.

Figure 2: The search tree $\mathbb{T}^S_{!(T1,T2)}(G)$ for the graph from Example 2.

6 Enumerating all maximal clique-partitions

In this section we describe an algorithm to compute the maximal clique-partitions one-by-one, on request. The following global data is assumed to be available:

- $\overline{C}_1,\ldots,\overline{C}_m$: the maximal cliques of G
- $cliques(v) = \{k \in [m] \mid v \in \overline{C}_k\}$ for all $v \in V$
- $Rgd = \{k \mid cliques(v) = \{k\}$ for some $v \in V\}$
- $S = [v_1,\ldots,v_s]$: an enumeration of all vertices $v \in V$ with $d(v) > 1$

During the computation we will keep track of the following information:

- ℓ: the depth of the search in tree $\mathbb{T}^S_{!(T1,T2)}(G)$
- $cfg[\ell]$: the current configuration of the search in tree $\mathbb{T}^S_{!(T1,T2)}(G)$
- $choice[i]$ for $1 \leq i \leq \ell$: the index of the clique where vertex $v_i \in S$ is assigned. If $choice[i] == 0$ then vertex v_i was not yet been assigned to any clique.

```
PROCEDURE initSearch()
  cfg[1] := [C̄_1,...,C̄_m];
  ℓ := 1;
  for i := 1 to s do
      choice[i] := 0;
  findNextClique();

PROCEDURE hasNext()
  return cfg[1] ≠ null

PROCEDURE next()
  if (cfg[1] == null) return null;
```

```
  else
    result := repr(cfg[ℓ]);
    findNextClique();
    return result;

PROCEDURE findNextClique()
  if s > 0
    while ℓ ≥ 1
      V_ℓ := {k ∈ cliques(v_ℓ) | k > choice[ℓ] and (k ∈ Rgd or cliques(C_k) ∩ Rgd = ∅)};
      if V_ℓ is empty
        choice[ℓ] := 0;
        ℓ := ℓ − 1;
      else
        i := min V_ℓ;
        // keep v_ℓ only in clique with index i
        choice[ℓ] := i;
        cfg[ℓ] := [C'_1, ..., C'_m] where C'_k := { C_k − {v_ℓ}   if k ∈ cliques(v_ℓ) − {i},
                                                  { C_k           otherwise
        if (isT1OrT2(cfg[ℓ])) continue;
        else
          if (ℓ == s)
            return;          // maximal clique-partition detected
          else
            cfg[ℓ + 1] := cfg[ℓ];
            ℓ := ℓ + 1;
  cfg[1] := null;

PROCEDURE isT1orT2([C_1, ..., C_m])
  I := {choice[k] | 1 ≤ k ≤ ℓ} − Rgd;
  J := I ∩ cliques(v_ℓ);
  for all j ∈ J
    if cliques(C_j) ∩ Rgd ≠ ∅ or cliques(C_j) ∩ {i | j < i ≤ m} ≠ ∅
      return true;
    for all i ∈ I
      if i ∈ J
        if (j < i) and cliques(C_i) ∩ cliques(C_j) ≠ ∅
          return true;
      else if cliques(C_i) ∩ cliques(C_j) ≠ ∅
        return true;
  return false;
```

7 Complexity

Theorem 1. *If the set of vertices of G is $\{v_1, \ldots, v_n\}$ then the number of maximal clique-partitions of G is at most $\prod_{i=1}^{n} d(v_i)$.*

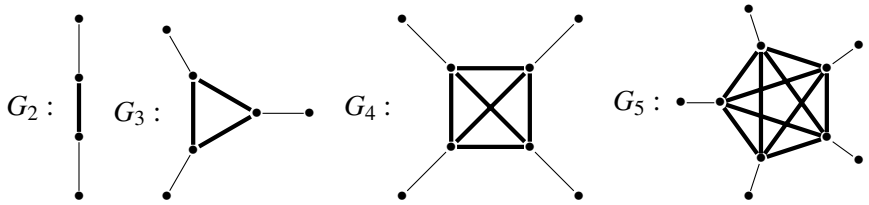
Proof. The result directly follows from the facts that (1) every maximal clique-partition is produced by some final configuration of $\mathbb{T}^S(G)$, and (2) $\mathbb{T}^S(G)$ has $\prod_{i=1}^{n} d(v_i)$ final configurations. □

It is easy to see that this upper bound can be reached. Just consider the graph with two maximal cliques: $C_1 = \{p_1, \ldots, p_n, true\}$ and $C_2 = \{p_1, \ldots, p_n, false\}$. The set of all maximal clique-partitions imitates the truth assignment in propositional logic, containing 2^n maximal clique-partitions.

This theorem implies that the algorithm is exponential in the number of vertices shared among multiple cliques. On the other hand, the length of each branch of the algorithm is polynomially bounded, since it requires at most as many steps as there are vertices shared among maximal cliques. Therefore, every single maximal clique-partition can be computed in polynomial time.

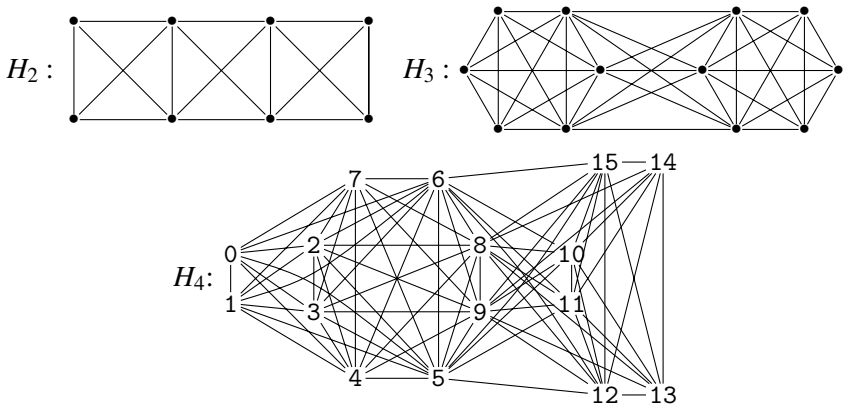
Experimental results To test the performance of our algorithm, we implemented it in Java and Mathematica [18], and ran it on a MacBook Air M2 with 8-core CPU and 8 GB RAM. We indicate the runtimes to enumerate all maximal clique-partitions of some graphs from the following families:

1. G_n, $n \geq 2$, obtained by extending the complete graph K_n with n new vertices, and connecting every vertex of K_n with a distinct new vertex. Examples of graphs G_n are:



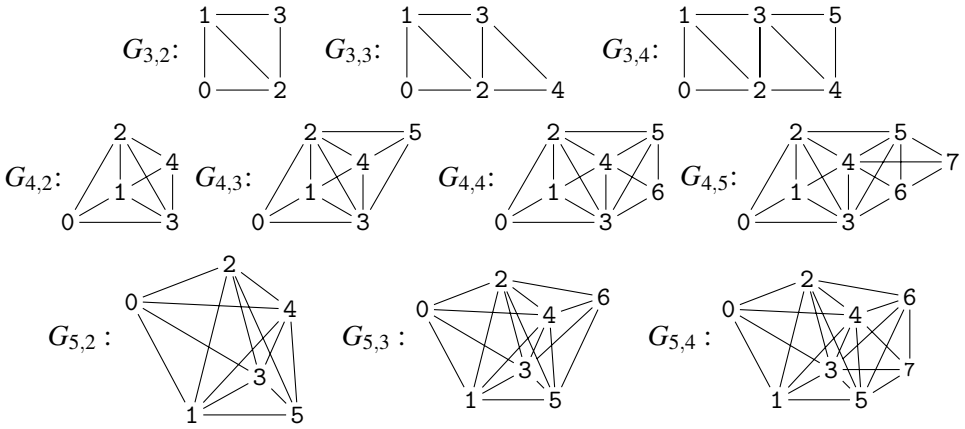
Graph	Order	Number of vertices in S	Number of maximal clique-partitions	Runtime
G_4	8	4	12	0.002
G_5	10	5	27	0.003
G_6	12	6	58	0.005
G_7	14	7	121	0.007
G_8	16	8	248	0.012
G_{10}	20	10	1014	0.270
G_{20}	40	20	1048556	2.462
G_{25}	50	25	33554407	61.706

2. H_n of order $4n$ ($n \geq 2$), with three maximal cliques of order $2n$: two are mutually disjoint, and the third one shares n vertices with each of the other two maximal cliques. For example:



Graph	Order	Number of vertices in S	Number of maximal clique-partitions	Runtime
H_4	16	8	226	0.018
H_5	20	10	962	0.028
H_6	24	12	3970	0.057
H_7	28	14	16130	0.122
H_8	32	16	65026	0.241
H_9	36	18	261122	0.723
H_{10}	40	20	1046530	2.280
H_{11}	44	22	2094082	3.944

3. Graphs $G_{m,n}$ with set of vertices $V = \{v_1, v_2, \ldots, v_{m+n-1}\}$ and n maximal cliques $\overline{C}_1, \ldots, \overline{C}_n$ such that $\overline{C}_i = \{v_j \mid i \leq j < i+m\}$ for all $1 \leq i \leq n$. Examples of graphs $G_{m,n}$ are:



Graph	Order	Number of vertices in S	Number of maximal clique-partitions	Runtime
$G_{3,2}$	4	2	4	0.001
$G_{3,3}$	5	3	5	0.001
$G_{3,4}$	6	4	7	0.002
$G_{4,5}$	8	6	33	0.008
$G_{5,3}$	7	5	35	0.003
$G_{5,4}$	8	6	65	0.006
$G_{5,5}$	9	7	114	0.015
$G_{5,6}$	10	8	200	0.031
$G_{6,6}$	11	9	781	0.037
$G_{7,5}$	11	9	1488	0.032
$G_{7,6}$	12	10	3135	0.082
$G_{8,6}$	13	11	12913	0.173
$G_{9,6}$	14	12	54495	0.408
$G_{11,9}$	19	17	20899403	121.135
$G_{12,8}$	19	17	40778092	202.608

The Mathematica implementation is slightly slower, but it enables a graphical visualization of the computed results. Some test examples can be seen online at

`staff.fmi.uvt.ro/~mircea.marin/software/MCP.html`

8 Conclusion

We developed an algorithm for computing all maximal clique-partitions in an undirected graph. The algorithm starts from the maximal clique cover of the graph and revises it, reducing the number of vertices shared among cliques by assigning them to one of the cliques they belong to. In this process, we avoid the computation of undesirable answers by detecting and discarding the search states which produce only non-maximal clique-partitions or duplicate answers. Our algorithm is optimal in the following sense: it enumerates all maximal clique-partitions, and each of them is computed only once. The set of computed partitions can be exponentially large with respect to the number of vertices shared among maximal cliques, but every answer can be computed in polynomial time (starting from all maximal cliques). Besides, the computations of different maximal clique-partitions corresponds to the exploration of different branches of the search tree for solutions, and they can be carried out independently, in parallel of each other. Our algorithm is iterative in the sense that it enumerates the computed answers one by one, on demand. This is highly desirable because the total number of maximal clique-partitions can be exponentially large and we may want to start analyzing and processing them as soon as possible.

In many practical applications, we are not interested to enumerate all maximal clique-partitions. Often, our algorithm can be easily adjusted to reduce the search space and compute only the preferred ones. For instance, we can impose the constraint to keep a group of vertices in the same clique by assigning them simultaneously (whenever possible) to the same single clique originating from a maximal clique of the graph.

References

[1] Hassan Aït-Kaci & Gabriella Pasi (2020): *Fuzzy lattice operations on first-order terms over signatures with similar constructors: A constraint-based approach*. *Fuzzy Sets Syst.* 391, pp. 1–46, doi:10.1016/j.fss.2019.03.019.

[2] María Alpuente, Santiago Escobar, José Meseguer & Julia Sapiña (2022): *Order-sorted equational generalization algorithm revisited*. *Ann. Math. Artif. Intell.* 90(5), pp. 499–522, doi:10.1007/s10472-021-09771-1.

[3] Jayaram Bhasker & Tariq Samad (1991): *The clique-partitioning problem*. *Computers and Mathematics with Applications* 22(6), pp. 1–11, doi:10.1016/0898-1221(91)90001-K.

[4] Coenraad Bron & Joep Kerbosch (1973): *Finding All Cliques of an Undirected Graph (Algorithm 457)*. *Commun. ACM* 16(9), pp. 575–576, doi:10.1145/362342.362367.

[5] Frédéric Cazals & Chinmay Karande (2008): *A note on the problem of reporting maximal cliques*. *Theor. Comput. Sci.* 407(1-3), pp. 564–568, doi:10.1016/j.tcs.2008.05.010.

[6] David M. Cerna & Temur Kutsia (2023): *Anti-unification and Generalization: A Survey*, doi:10.48550/ARXIV.2302.00277. Available at https://arxiv.org/abs/2302.00277.

[7] Pascual Julián Iranzo & Clemente Rubio-Manzano (2015): *Proximity-based unification theory*. *Fuzzy Sets and Systems* 262, pp. 21–43, doi:10.1016/j.fss.2014.07.006.

[8] Pascual Julián Iranzo & Fernando Sáenz-Pérez (2021): *Proximity-Based Unification: An Efficient Implementation Method*. *IEEE Trans. Fuzzy Syst.* 29(5), pp. 1238–1251, doi:10.1109/TFUZZ.2020.2973129.

[9] Richard M. Karp (1972): *Reducibility Among Combinatorial Problems*. In Raymond E. Miller & James W. Thatcher, editors: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, Plenum Press, New York, pp. 85–103, doi:10.1007/978-3-540-68279-0_8.

[10] Temur Kutsia & Cleopatra Pau (2019): *Matching and Generalization Modulo Proximity and Tolerance Relations*. In Aybüke Özgün & Yulia Zinova, editors: *Language, Logic, and Computation - 13th International Tbilisi Symposium, TbiLLC 2019, Batumi, Georgia, September 16-20, 2019, Revised Selected Papers*, *Lecture Notes in Computer Science* 13206, Springer, pp. 323–342, doi:10.1007/978-3-030-98479-3_16.

[11] Temur Kutsia & Cleopatra Pau (2022): *A Framework for Approximate Generalization in Quantitative Theories*. In Jasmin Blanchette, Laura Kovács & Dirk Pattinson, editors: *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, *Lecture Notes in Computer Science* 13385, Springer, pp. 578–596, doi:10.1007/978-3-031-10769-6_34.

[12] Cleo Pau (2022): *Symbolic Techniques for Approximate Reasoning*. Ph.D. thesis, RISC, Johannes Kepler University Linz.

[13] Gordon D. Plotkin (1970): *A note on inductive generalization*. *Machine Intel.* 5(1), pp. 153–163.

[14] John C. Reynolds (1970): *Transformational systems and the algebraic structure of atomic formulas*. *Machine Intel.* 5(1), pp. 135–151.

[15] Etsuji Tomita (2017): *Efficient Algorithms for Finding Maximum and Maximal Cliques and Their Applications*. In Sheung-Hung Poon, Md. Saidur Rahman & Hsu-Chun Yen, editors: *WALCOM: Algorithms and Computation, 11th International Conference and Workshops, WALCOM 2017, Hsinchu, Taiwan, March 29-31, 2017, Proceedings.*, *Lecture Notes in Computer Science* 10167, Springer, pp. 3–15, doi:10.1007/978-3-319-53925-6_1.

[16] Etsuji Tomita, Akira Tanaka & Haruhisa Takahashi (2006): *The worst-case time complexity for generating all maximal cliques and computational experiments*. *Theor. Comput. Sci.* 363(1), pp. 28–42, doi:10.1016/j.tcs.2006.06.015.

[17] Chia-Jeng Tseng (1984): *Automated synthesis of data paths in digital systems*. Ph.D. thesis, Dept. of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburg, PA, doi:10.1109/TCAD.1986.1270207.

[18] Stephen Wolfram (2003): *The Mathematica book, 5th Edition*. Wolfram-Media.

A Formalized Extension of the Substitution Lemma in Coq

Maria J. D. Lima
Departamento de Ciência da Computação
Universidade de Brasília, Brasília, Brazil
majuhdl@gmail.com

Flávio L. C. de Moura
Departamento de Ciência da Computação
Universidade de Brasília, Brasília, Brazil
flaviomoura@unb.br

The substitution lemma is a renowned theorem within the realm of λ-calculus theory and concerns the interactional behaviour of the metasubstitution operation. In this work, we augment the λ-calculus's grammar with an uninterpreted explicit substitution operator, which allows the use of our framework for different calculi with explicit substitutions. Our primary contribution lies in verifying that, despite these modifications, the substitution lemma continues to remain valid. This confirmation was achieved using the Coq proof assistant. Our formalization methodology employs a nominal approach, which provides a direct implementation of the α-equivalence concept. The strategy involved in variable renaming within the proofs presents a challenge, specially on ensuring an exploration of the implications of our extension to the grammar of the λ-calculus.

1 Introduction

In this work, we present a formalization of the substitution lemma [5] in a general framework that extends the λ-calculus with an explicit substitution operator using the Coq proof assistant [24]. The source code is publicly available at

https://flaviomoura.info/files/msubst.v

The substitution lemma is an important result concerning the composition of the substitution operation, and is usually presented as follows in the context of the λ-calculus:

> Let t, u and v be λ-terms, $x \neq y$ and $x \notin FV(v)$, where $FV(v)$ is the set of free variables of v. Then $\{y := v\}\{x := u\}t = \{x := \{y := v\}u\}\{y := v\}t$.

This is a well known result already formalized in the context of the λ-calculus [7]. Nevertheless, in the context of λ-calculi with explicit substitutions its formalization is not trivial due to the interaction between the metasubstitution and the explicit substitution operator. Our formalization is done in a nominal setting that uses the MetaLib[1] package of Coq, but no particular explicit substitution calculi is taken into account because the expected behaviour between the metasubstitution operation with the explicit substitutition constructor is the same regardless the calculus. The formalization was done with Coq (platform) version 8.15.2, which already comes with the Metalib package. The novel contributions of this work are twofold:

1. The formalization is modular in the sense that no particular calculi with explicit substitutions is taken into account. Therefore, we believe that this formalization could be seen as a generic framework for proving properties of these calculi that uses the substitution lemma in the nominal setting [16, 20, 21];

[1] https://github.com/plclub/metalib

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 80–95, doi:10.4204/EPTCS.389.7

2. A solution to a circularity problem in the proofs is given. It adds an axiom to the formalization that allow a rewrite step inside a let expression. Such a rewrite step is problematic and does not seem to have a trivial solution.

2 A syntactic extension of the λ-calculus

In this section, we present the framework of the formalization, which is based on a nominal approach [12] where variables use names. In the nominal setting, variables are represented by atoms that are structureless entities with a decidable equality:

```
Parameter eq_dec : forall x y : atom, {x = y} + {x <> y}.
```

therefore different names mean different atoms and different variables. The nominal approach is close to the usual paper and pencil notation used in λ-calculus, whose grammar of terms is given by:

$$t ::= x \mid \lambda_x.t \mid t\ t \tag{1}$$

where x represents a variable which is taken from an enumerable set, $\lambda_x.t$ is an abstraction, and $t\ t$ is an application. The abstraction is the only binding operator: in the expression $\lambda_x.t$, x binds in t, called the scope of the abstraction. This means that all free occurrence of x in t is bound in $\lambda_x.t$. A variable that is not in the scope of an abstraction is free. A variable in a term is either bound or free, but note that a varible can occur both bound and free in a term, as in $(\lambda_y.y)\ y$.

The main rule of the λ-calculus, named β-reduction, is given by:

$$(\lambda_x.t)\ u \to_\beta \{x := u\}t \tag{2}$$

where $\{x := u\}t$ represents the result of substituting all free occurrences of variable x in t with u in such a way that renaming of bound variable may be done in order to avoid the variable capture of free variables. We call t the body of the metasubstitution, and u its argument. In other words, $\{x := u\}t$ is a metanotation for a capture free substitution. For instance, the λ-term $(\lambda_x\lambda_y.x\ y)\ y$ has both bound and free occurrences of the variable y, and in order to β-reduce it, one has to replace (or substitute) the free variable y for all free occurrences of the variable x in the term $(\lambda_y.x\ y)$. But a straight substitution will capture the free variable y, *i.e.* this means that the free occurrence of y before the β-reduction will become bound after the β-reduction step. A renaming of bound variables may be done to avoid such a capture, so in this example, one can take an α-equivalent[2] term, say $(\lambda_z.x\ z)$, and perform the β-step correctly as $(\lambda_x\lambda_y.x\ y)\ y \to_\beta \lambda_z.y\ z$. Renaming of variables in the nominal setting is done via a name-swapping, which is formally defined as follows:

$$(\!(x\ y)\!)z := \begin{cases} y, & \text{if } z = x; \\ x, & \text{if } z = y; \\ z, & \text{otherwise.} \end{cases}$$

This notion can be extended to λ-terms in a straightfoward way:

$$(x\ y)t := \begin{cases} (\!(x\ y)\!)z, & \text{if } t = z; \\ \lambda_{(\!(x\ y)\!)z}.(x\ y)t_1, & \text{if } t = \lambda_z.t_1; \\ (x\ y)t_1\ (x\ y)t_2, & \text{if } t = t_1\ t_2 \end{cases} \tag{3}$$

[2] A formal definition of this notion will be given later in this section.

In the previous example, one could apply a swap to avoid the variable capture in a way that, a swap is applied to the body of the abstraction before applying the metasubstitution to it: $(\lambda_x\lambda_y.x\ y)\ y \to_\beta \{x := y\}((y\ z)(\lambda_y.x\ y)) = \{x := y\}(\lambda_z.x\ z) = \lambda_z.y\ z$. Could we have used a variable substitution instead of a swapping in the previous example? Absolutely. We could have done the reduction as $(\lambda_x\lambda_y.x\ y)\ y \to_\beta \{x := y\}(\{y := z\}(\lambda_y.x\ y)) = \{x := y\}(\lambda_z.x\ z) = \lambda_z.y\ z$, but as we will shortly see, variable substitution is not stable modulo α-equivalence, while the swapping is, thereby rendering it a more fitting choice when operating with α-classes.

In what follows, we will adopt a mixed-notation approach, intertwining metanotation with the equivalent Coq notation. This strategy aids in elucidating the proof steps of the upcoming lemmas, enabling a clearer and more detailed comprehension of each stage in the argumentation. The corresponding Coq code for the swapping of variables, named *vswap*, is defined as follows:

`Definition` *vswap* (*x*:*atom*) (*y*:*atom*) (*z*:*atom*) := `if` (*z* == *x*) `then` *y* `else if` (*z* == *y*) `then` *x* `else` *z*.

therefore, the swap $(\!(x\ y)\!)z$ is written in Coq as *vswap x y z*. As a short example to acquaint ourselves with the Coq notation, let us show how we will write the proofs:

`Lemma` *vswap_id*: $\forall$ *x y*, *vswap x x y* = *y*.
Proof. The proof is by case analysis, and it is straightforward in both cases, when $x = y$ and $x \neq y$. □

2.1 An explicit substitution operator

The extension of the swap operation to terms require an additional comment because we will not work with the grammar (1), but rather, we will extend it with an explicit substitution operator:

$$t ::= x \mid \lambda_x.t \mid t\ t \mid [x := u]t \tag{4}$$

where $[x := u]t$ represents a term with an operator that will be evaluated with specific rules of a substitution calculus. The intended meaning of the explicit substitution is that it will simulate the metasubstitution. This formalization aims to be a generic framework applicable to any calculi with explicit substitutions using a named notation for variables. Therefore, we will not specify rules about how one can simulate the metasubstitution, but it is important to be aware that this is not a trivial task as one can easily lose important properties of the original λ-calculus [18, 14].

Calculi with explicit substitutions are formalisms that deconstruct the metasubstitution operation into finer-grained steps, thereby functioning as an intermediary between the λ-calculus and its practical implementations. In other words, these calculi shed light on the execution models of higher-order languages. In fact, the development of a calculus with explicit substitutions faithful to the λ-calculus, in the sense of the preservation of some desired properties were the main motivation for such a long list of calculi with explicit substitutions invented in the last decades [1, 23, 6, 10, 19, 15, 8, 11, 17].

The following inductive definition corresponds to the grammar (4), where the explicit substitution constructor, named *n_sub*, has a special notation. Instead of writing *n_sub t x u*, we will write [*x* := *u*] *t* similarly to (4). Accordingly, *n_sexp* denotes the set of nominal λ-expressions equipped with an explicit substitution operator, which, for simplicity, we will refer to as just "terms".

`Inductive` *n_sexp* : `Set` :=
| *n_var* (*x*:*atom*)
| *n_abs* (*x*:*atom*) (*t*:*n_sexp*)
| *n_app* (*t1*:*n_sexp*) (*t2*:*n_sexp*)
| *n_sub* (*t1*:*n_sexp*) (*x*:*atom*) (*t2*:*n_sexp*).

The *size* of a term, also written as $|t|$, and the set *fv_nom* of the free variables of a term are defined as usual:

```
Fixpoint size (t : n_sexp) : nat :=
  match t with
  | n_var x ⇒ 1
  | n_abs x t ⇒ 1 + size t
  | n_app t1 t2 ⇒ 1 + size t1 + size t2
  | n_sub t1 x t2 ⇒ 1 + size t1 + size t2
  end.

Fixpoint fv_nom (t : n_sexp) : atoms :=
  match t with
  | n_var x ⇒ {{x}}
  | n_abs x t1 ⇒ remove x (fv_nom t1)
  | n_app t1 t2 ⇒ fv_nom t1 `union` fv_nom t2
  | n_sub t1 x t2 ⇒ (remove x (fv_nom t1)) `union` fv_nom t2
  end.
```

The action of a permutation on a term, written $(x\ y)t$, is inductively defined as in (3) with the additional case for the explicit substitution operator:

$$(x\ y)t := \begin{cases} (\!|x\ y|\!)v, & \text{if } t \text{ is the variable } v; \\ \lambda_{(\!|x\ y|\!)z}.(x\ y)t_1, & \text{if } t = \lambda_z.t_1; \\ (x\ y)t_1\ (x\ y)t_2, & \text{if } t = t_1\ t_2; \\ [(\!|x\ y|\!)z := (x\ y)t_2](x\ y)t_1, & \text{if } t = [z := t_2]t_1. \end{cases}$$

The corresponding Coq definition is given by the following recursive function:

```
Fixpoint swap (x:atom) (y:atom) (t:n_sexp) : n_sexp :=
  match t with
  | n_var z ⇒ n_var (vswap x y z)
  | n_abs z t1 ⇒ n_abs (vswap x y z) (swap x y t1)
  | n_app t1 t2 ⇒ n_app (swap x y t1) (swap x y t2)
  | n_sub t1 z t2 ⇒ n_sub (swap x y t1) (vswap x y z) (swap x y t2)
  end.
```

The *swap* function has many interesting properties, but we will focus on the ones that are more relevant to the proofs related to the substitution lemma. Nevertheless, all lemmas can be found in the source code of the formalization[3]. The next lemmas are simple properties that are all proved by induction on the structure of term *t*:

```
Lemma swap_neq: ∀ x y z w, z ≠ w → vswap x y z ≠ vswap x y w.

Lemma swap_size_eq : ∀ x y t, size (swap x y t) = size t.

Lemma swap_symmetric : ∀ t x y, swap x y t = swap y x t.

Lemma swap_involutive : ∀ t x y, swap x y (swap x y t) = t.
```

[3] `https://flaviomoura.info/files/msubst.v`

Lemma *shuffle_swap* : $\forall\ w\ y\ z\ t,\ w \neq z \rightarrow y \neq z \rightarrow (swap\ w\ y\ (swap\ y\ z\ t)) = (swap\ w\ z\ (swap\ w\ y\ t))$.

Lemma *swap_equivariance* : $\forall\ t\ x\ y\ z\ w,\ swap\ x\ y\ (swap\ z\ w\ t) = swap\ (vswap\ x\ y\ z)\ (vswap\ x\ y\ w)\ (swap\ x\ y\ t)$.

Lemma *fv_nom_swap* : $\forall\ z\ y\ t,\ z\ `notin`\ fv\_nom\ t \rightarrow y\ `notin`\ fv\_nom\ (swap\ y\ z\ t)$.

The standard proof strategy used so far is induction on the structure of terms. Nevertheless, the builtin induction principle automatically generated in Coq for the inductive definition *n_sexp* is not strong enough due to swappings:

```
forall P :n_sexp -> Prop,
 (forall x:atom, P(n_var x)) ->
 (forall (x:atom) (t:n_sexp), P t -> P(n_abs x t)) ->
 (forall t1:n_sexp, P t1 -> forall t2:n_sexp, P t2 -> P(n_app t1 t2)) ->
 (forall t1:n_sexp, P t1 -> forall (x:atom) (t2:n_sexp), P t2 -> P([x:=t2]t1)) ->
       forall t:n_sexp, P t
```

In fact, in general, the induction hypothesis in the abstraction case (resp. explicit substitution case) refers to the body *t* of the abstraction (resp. *t1* of the explicit substitution), while the goal involves a swap acting on the body of the abstraction (resp. explicit substitution). In order to circunvet this problem, we defined a customized induction principle based on the size of terms:

Lemma *n_sexp_induction*: $\forall\ P : n\_sexp \rightarrow \mathtt{Prop},\ (\forall\ x,\ P\ (n\_var\ x)) \rightarrow$
$(\forall\ t1\ z,\ (\forall\ t2\ x\ y,\ size\ t2 = size\ t1 \rightarrow P\ (swap\ x\ y\ t2)) \rightarrow P\ (n\_abs\ z\ t1)) \rightarrow$
$(\forall\ t1\ t2,\ P\ t1 \rightarrow P\ t2 \rightarrow P\ (n\_app\ t1\ t2)) \rightarrow$
$(\forall\ t1\ t3\ z,\ P\ t3 \rightarrow (\forall\ t2\ x\ y,\ size\ t2 = size\ t1 \rightarrow P\ (swap\ x\ y\ t2)) \rightarrow P\ (n\_sub\ t1\ z\ t3)) \rightarrow (\forall\ t,\ P\ t)$.

which states that in order to conclude that a certain property P holds for all terms, we need to prove that:

1. P must hold for any variable;
2. If P holds for the term $(x\ y)t_2$, where t_1 and t_2 have the same size, then it also holds for the abstraction $\lambda_z.t_1, \forall x, y, z, t_1$ and t_2;
3. If P holds for the terms t_1 and t_2 the it also holds for the application $t_1\ t_2$;
4. If P holds for the term t_3 and for the term $(x\ y)t_2$, where t_1 and t_2 have the same size, then it also holds for the explicit substitution $[z := t_3]t_1, \forall x, y, z, t_1, t_2$ and t_3.

The following lemma is a first example of the use of the *n_sexp_induction* principle:

Lemma *notin_fv_nom_equivariance*: $\forall\ t\ x'\ x\ y,\ x'\ `notin`\ fv\_nom\ t \rightarrow vswap\ x\ y\ x'\ `notin`\ fv\_nom\ (swap\ x\ y\ t)$.

Proof. Note that in the paper and pencil notation, this lemma states that:

If $x' \notin fv\_nom(t)$ then $(\!(x\ y)\!)x' \notin fv\_nom((x\ y)t)$.

The proof is by induction on the size of the term t.

1. If t is a variable, say z, then $x' \neq z$ by hypothesis, and we need to prove that $(\!(x\ y)\!)x' \neq (\!(x\ y)\!)z$. We conclude by lemma *swap_neq*.

2. If is an abstraction, say $t = \lambda_z.t_1$, then we have by induction hypothesis that if $x' \notin (x\ y)t_2$ then $(\!(x_0\ y_0)\!)x' \notin (x_0\ y_0)(x\ y)t_2$ for any term t_2 with the same size as t_1, and any variables x, y, x_0 and y_0. At this point is important to notice that an structural induction would generate an induction hypothesis with t_1 only, which is not strong enough to prove the goal $(\!(x\ y)\!)x' \notin fv\_nom((x\ y)\lambda_z.t_1)$ that has $(x\ y)t_1$ (and not t_1 alone!) after the propagation of the swap. In addition, we have by hypothesis that $x' \notin fv\_nom(t_1)\backslash\{z\}$. This means that either $x' = z$ or $x' \notin fv\_nom(t_1)$, and there are two subcases:
   (a) If $x' = z$ then the goal is $(\!(x\ y)\!)z \notin fv\_nom((x\ y)\lambda_z.t_1) \Leftrightarrow (\!(x\ y)\!)z \notin fv\_nom(\lambda_{(\!(x\ y)\!)z}.(x\ y)t_1) \Leftrightarrow (\!(x\ y)\!)z \notin fv\_nom((x\ y)t_1)\backslash\{(\!(x\ y)\!)z\}$ we are done by lemma *notin_remove_3*.[4]
   (b) Otherwise, $x' \notin fv\_nom(t_1)$, and we conclude using the induction hypothesis taking $x_0 = x$, $y_0 = y$ and the universally quantified variables x and y of the internal swap as the same variable (it does not matter which one).
3. The application case is straightforward from the induction hypothesis.
4. The case of the explicit substitution, *i.e.* when $t = [z := t_2]t_1$, we have to prove that $(\!(x\ y)\!)x' \notin fv\_nom((x\ y)([z := t_2]t_1))$. We then propagate the swap over the explicit substitution operator and show, by the definition of fv_nom, we have to prove that both $(\!(x\ y)\!)x' \notin (fv\_nom((x\ y)t_1))\backslash\{(\!(x\ y)\!)z\}$ and $(\!(x\ y)\!)x' \notin fv\_nom((x\ y)t_2)$.
   (a) In the former case, the hypothesis $x' \notin fv\_nom(t_1)\backslash\{z\}$ generates two subcases, either $x' = z$ or $x' \notin fv\_nom(t_1)$, and we conclude with the same strategy of the abstraction case.
   (b) The later case is straightforward by the induction hypothesis. □

The other direction is also true, but we skip the proof that is also by induction on the size of term t:

`Lemma` *notin_fv_nom_remove_swap*: ∀ *t x' x y*, *vswap x y x'* ‘*notin*‘ *fv_nom* (*swap x y t*) → *x'* ‘*notin*‘ *fv_nom t*.

2.2 α-equivalence

As usual in the standard presentations of the λ-calculus, we work with terms modulo α-equivalence. This means that λ-terms are identified up to renaming of bound variables. For instance, all terms $\lambda_x.x$, $\lambda_y.y$ and $\lambda_z.z$ are seen as the same term which corresponds to the identity function. Formally, the notion of α-equivalence is defined by the following inference rules:

$$\frac{}{x =_\alpha x}\ (aeq\_var) \qquad \frac{t_1 =_\alpha t_2}{\lambda_x.t_1 =_\alpha \lambda_x.t_2}\ (aeq\_abs\_same)$$

$$\frac{x \neq y \qquad x \notin fv(t_2) \qquad t_1 =_\alpha (y\ x)t_2}{\lambda_x.t_1 =_\alpha \lambda_y.t_2}\ (aeq\_abs\_diff)$$

$$\frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{t_1\ t_2 =_\alpha t_1'\ t_2'}\ (aeq\_app) \qquad \frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{[x := t_2]t_1 =_\alpha [x := t_2']t_1'}\ (aeq\_sub\_same)$$

[4] This is a lemma from Metalib library and it states that `forall (x y : atom) (s : atoms), x = y -> y ‘notin‘ remove x s`.

$$\frac{t_2 =_\alpha t_2' \qquad x \neq y \qquad x \notin fv(t_1') \qquad t_1 =_\alpha (y\ x)t_1'}{[x := t_2]t_1 =_\alpha [y := t_2']t_1'}\ (aeq\_sub\_diff)$$

Each of these rules correspond to a constructor in the *aeq* inductive definition below:

```
Inductive aeq : n_sexp → n_sexp → Prop :=
| aeq_var : ∀ x, aeq (n_var x) (n_var x)
| aeq_abs_same : ∀ x t1 t2, aeq t1 t2 → aeq (n_abs x t1)(n_abs x t2)
| aeq_abs_diff : ∀ x y t1 t2, x ≠ y → x ‘notin‘ fv_nom t2 → aeq t1 (swap y x t2) →
  aeq (n_abs x t1) (n_abs y t2)
| aeq_app : ∀ t1 t2 t1' t2', aeq t1 t1' → aeq t2 t2' → aeq (n_app t1 t2) (n_app t1' t2')
| aeq_sub_same : ∀ t1 t2 t1' t2' x, aeq t1 t1' → aeq t2 t2' → aeq ([x := t2] t1) ([x := t2'] t1')
| aeq_sub_diff : ∀ t1 t2 t1' t2' x y, aeq t2 t2' → x ≠ y → x ‘notin‘ fv_nom t1' → aeq t1 (swap y x t1') →
  aeq ([x := t2] t1) ([y := t2'] t1').
```

In what follows, we use a infix notation for α-equivalence in the Coq code. Therefore, we write *t =a u* instead of *aeq t u*. The above notion defines an equivalence relation over the set *n_sexp* of nominal expressions with explicit substitutions, *i.e.* the *aeq* relation is reflexive, symmetric and transitive (proofs in the source file[5]). In addition, α-equivalent terms have the same size, and the same set of free variables:

```
Lemma aeq_size: ∀ t1 t2, t1 =a t2 → size t1 = size t2.
```

```
Lemma aeq_fv_nom : ∀ t1 t2, t1 =a t2 → fv_nom t1 [=] fv_nom t2.
```

The key point of the nominal approach is that the swap operation is stable under α-equivalence in the sense that, $t_1 =_\alpha t_2$ if, and only if $(x\ y)t_1 =_\alpha (x\ y)t_2, \forall t_1, t_2, x, y$. Note that this is not true for renaming substitutions: in fact, $\lambda_x.z =_\alpha \lambda_y.z$, but $\{z := x\}(\lambda_x.z) = \lambda_x.x \neq_\alpha \{z := x\}\lambda_y.x(\lambda_y.z)$, assuming that $x \neq y$. This stability result is formalized as follows:

```
Corollary aeq_swap: ∀ t1 t2 x y, t1 =a t2 ↔ (swap x y t1) =a (swap x y t2).
```

When both variables in a swap do not occur free in a term, it eventually renames only bound variables, *i.e.* the action of this swap results in a term that is α-equivalent to the original term. This is the content of the following lemma:

```
Lemma swap_reduction: ∀ t x y, x ‘notin‘ fv_nom t → y ‘notin‘ fv_nom t → (swap x y t) =a t.
```

There are several other interesting auxiliary properties that need to be proved before achieving the substitution lemma. In what follows, we refer only to the tricky or challenging ones, but the interested reader can have a detailed look in the source file. Note that, swaps are introduced in proofs by the rules *aeq_abs_diff* and *aeq_sub_diff*. As we will see, the proof steps involving these rules are trick because a naïve strategy can easily get blocked in a branch without proof. We conclude this section, with a lemma that gives the conditions for two swaps with a common variable to be merged:

```
Lemma aeq_swap_swap: ∀ t x y z, z ‘notin‘ fv_nom t → x ‘notin‘ fv_nom t → (swap z x (swap x y t)) =a
(swap z y t).
```

Proof. Before commenting this proof, we state the lemma with the pencil and paper (meta)notation:

[5] `https://flaviomoura.info/files/msubst.v`

If $z \notin fv\_nom(t)$ and $x \notin fv\_nom(t)$ then $(z\,x)(x\,y)t =_\alpha (z\,y)t$.

Initially, observe the similarity of the left hand side (LHS) of the α-equation with the lemma *shuffle_swap*:

$\forall w\ y\ z\ t, w \neq z \to y \neq z \to (w\ y)((y\ z)t) = (w\ z)((w\ y)t)$

In order to use it, we need to have that both $z \neq y$ and $x \neq y$. We start comparing z and y:

1. If $z = y$ then the right hand side (RHS) reduces to t because the swap is trivial, and the LHS also reduces to t since swap is involutive.

2. When $z \neq y$ then we proceed by comparing x and y:

   (a) If $x = y$ then both sides of the α-equation reduces to $(z\,y)t$, and we are done.

   (b) Finally, when $x \neq y$, we can apply the lemma $shuffle\_swap$, and use lemma *aeq_swap* to reduce the current goal to $(z\,x)t =_\alpha t$, and we conclude by lemma $swap\_reduction$ since both z and x are not in the set of free variables of the term t. □

3 The metasubstitution operation of the λ-calculus

As presented in Section 2, the main operation of the λ-calculus is the β-reduction (2) that expresses how to evaluate a function applied to an argument. The β-contractum $\{x := u\}t$ represents a capture free in the sense that no free variable becomes bound by the application of the metasubstitution. This operation is in the meta level because it is outside the grammar of the λ-calculus (and hence its name). In [5], Barendregt defines it as follows:

$$\{x := u\}t = \begin{cases} u, & \text{if } t = x; \\ y, & \text{if } t = y \text{ and } x \neq y; \\ \{x := u\}t_1\ \{x := u\}t_2, & \text{if } t = t_1\ t_2; \\ \lambda_y.(\{x := u\}t_1), & \text{if } t = \lambda_y.t_1. \end{cases}$$

where it is assumed the so called "Barendregt's variable convention":

> If $t_1, t_2, \ldots, t_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

This means that we are assumming that both $x \neq y$ and $y \notin fv(u)$ in the case $t = \lambda_y.t_1$. This approach is very convenient in informal proofs because it avoids having to rename bound variables. In order to formalize the capture free substitution, *i.e.* the metasubstitution, there are different possible approaches. In our case, we perform a renaming of bound variables whenever the metasubstitution is propagated inside a binder. In our case, there are two binders: abstractions and explicit substitutions.

Let t and u be terms, and x a variable. The result of substituting u for the free ocurrences of x in t, written $\{x := u\}t$ is defined as follows:

$$\{x := u\}t = \begin{cases} u, & \text{if } t = x; \\ y, & \text{if } t = y \ (x \neq y); \\ \{x := u\}t_1 \ \{x := u\}t_2, & \text{if } t = t_1 \ t_2; \\ \lambda_x.t_1, & \text{if } t = \lambda_x.t_1; \\ \lambda_z.(\{x := u\}((y\ z)t_1)), & \text{if } t = \lambda_y.t_1, x \neq y, z \notin fv(t) \cup fv(u) \cup \{x\}; \\ [x := \{x := u\}t_2]t_1, & \text{if } t = [x := t_2]t_1; \\ [z := \{x := u\}t_2]\{x := u\}((y\ z)t_1), & \text{if } t = [y := t_2]t_1, x \neq y, z \notin fv(t) \cup fv(u) \cup \{x\}. \end{cases} \quad (5)$$

and the corresponding Coq code is as follows:

```
Function subst_rec_fun (t:n_sexp) (u :n_sexp) (x:atom) {measure size t} : n_sexp :=
  match t with
  | n_var y ⇒ if (x == y) then u else t
  | n_abs y t1 ⇒ if (x == y) then t else let (z,_) :=
    atom_fresh (fv_nom u ‘union‘ fv_nom t ‘union‘ {{x}}) in n_abs z (subst_rec_fun (swap y z t1) u x)
  | n_app t1 t2 ⇒ n_app (subst_rec_fun t1 u x) (subst_rec_fun t2 u x)
  | n_sub t1 y t2 ⇒ if (x == y) then n_sub t1 y (subst_rec_fun t2 u x) else let (z,_) :=
    atom_fresh (fv_nom u ‘union‘ fv_nom t ‘union‘ {{x}}) in
    n_sub (subst_rec_fun (swap y z t1) u x) z (subst_rec_fun t2 u x) end.
```

Note that this function is not structurally recursive due to the swaps in the recursive calls, and that's why we need to provide the size of the term t as the measure parameter. Alternatively, a structurally recursive version of the function *subst_rec_fun* can be found in the file *nominal.v* of the *Metalib* library[6]. It has the size of the term as an explicit parameter in which the substitution will be performed, and hence one has to deal with the size of the term in each recursive call. We write $\{x{:=}u\}t$ instead of *subst_rec_fun t u x*, and refer to it just as "metasubstitution".

The following lemma states that if $x \notin fv(t)$ then $\{x := u\}t =_\alpha t$. In informal proofs the conclusion of this lemma is usually stated as a syntactic equality, i.e. $\{x := u\}t = t$ instead of the α-equivalence, but the function *subst_rec_fun* renames bound variables whenever the metasubstitution is propagated inside an abstraction or an explicit substitution, even in the case that the metasubstitution has no effect in the subterm it is propagated, as long as the variables of the metasubstitution and the binder (abstraction or explicit substitution) are different of each other. That's why the syntactic equality does not hold here.

```
Lemma m_subst_notin: ∀ t u x, x ‘notin‘ fv_nom t → {x := u}t =a t.
```

Proof. The proof is done by induction on the size of the term t using *n_sexp_induction* defined above. The interesting cases are the abstraction and the explicit substituion. We focus in the abstraction case, *i.e.* when $t = \lambda_y.t_1$, where the goal to be proven is $\{x := u\}(\lambda_y.t_1) =_\alpha \lambda_y.t_1$. We consider two cases:

1. If $x = y$ the result is trivial because both LHS and RHS are equal to $\lambda_y.t_1$
2. If $x \neq y$, we have to prove that $\lambda_z.\{x := u\}(y\ z)t_1 =_\alpha \lambda_y.t_1$, where z is a fresh name not in the set $fv\_nom(u) \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$. The induction hypothesis express the fact that every term with the same size as the body t_1 of the abstraction satisfies the property to be proven: $\forall t', |t'| = |t_1| \rightarrow \forall u\ x'\ x_0\ y_0, x' \notin fv((x_0\ y_0)t') \rightarrow \{x' := u\}((x_0\ y_0)t') =_\alpha (x\ y)t'$. Therefore, according to the definition of the metasubstitution (function [subst_rec_fun]), the variable y will be renamed to z, and the metasubstitution is propagated inside the abstraction resulting in the following goal:

[6] https://github.com/plclub/metalib

$\lambda_z.\{x := u\}((z\ y)t_1) =_\alpha \lambda_y.t_1$. Since $z \notin fv\_nom(\lambda_y.t_1) = fv\_nom(t_1)\backslash\{y\}$, there are two cases to consider, either $z = y$ or $z \in fv(t_1)$:

(a) $z = y$: In this case, we are done by the induction hypothesis taking $x_0 = y_0 = y$, for instance.

(b) $z \neq y$: In this case, we can apply the rule *aeq_abs_diff*, resulting in the goal $\{x := u\}((y\ z)t_1) =_\alpha (y\ z)t_1$ which holds by the induction hypothesis, since $|(z\ y)t_1| = |t_1|$ and $x \notin fv\_nom((y\ z)t_1)$ because $x \neq z$, $x \neq y$ and $x \notin fv\_nom(t_1)$.

The explicit substitution case is also interesting, *i.e.* if $t = [y := t_2]t_1$, but it follows a similar strategy used in the abstraction case for t_1. For t_2 the result follows from the induction hypothesis. □

The following lemmas concern the expected behaviour of the metasubstitution when the metasubstitution's variable is equal to the abstraction's variable. Their proofs are straightforward from the definition *subst_rec_fun*. The corresponding version when the metasubstitution's variable is different from the abstraction's variable will be presented later.

`Lemma` *m_subst_abs_eq*: $\forall\ u\ x\ t$, $\{x := u\}(n\_abs\ x\ t) = n\_abs\ x\ t$.

`Lemma` *m_subst_sub_eq*: $\forall\ u\ x\ t1\ t2$, $\{x := u\}(n\_sub\ t1\ x\ t2) = n\_sub\ t1\ x\ (\{x := u\}t2)$.

We will now prove some stability results for the metasubstitution w.r.t. α-equivalence. More precisely, we will prove that if $t =_\alpha t'$ and $u =_\alpha u'$ then $\{x := u\}t =_\alpha \{x := u'\}t'$, where x is a variable and t, t', u and u' are terms. This proof is split in two cases: firstly, we prove that if $u =_\alpha u'$ then $\{x := u\}t =_\alpha \{x := u'\}t, \forall x, t, u, u'$; secondly, we prove that if $t =_\alpha t'$ then $\{x := u\}t =_\alpha \{x := u\}t', \forall x, t, t', u$. These two cases are then combined through the transitivity of the α-equivalence relation. Nevertheless, this task was not straighforward. Let's follow the steps of our first trial.

`Lemma` *aeq_m_subst_in_trial*: $\forall\ t\ u\ u'\ x$, $u =a\ u' \rightarrow (\{x := u\}t) =a (\{x := u'\}t)$.

Proof. The proof is done by induction on the size of term t, and we will focus on the abstraction case, *i.e.* $t = \lambda_y.t_1$. The goal in this case is $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u'\}(\lambda_y.t_1)$.

1. If $x = y$ then the result is trivial by lemma *m_subst_abs_eq*.
2. If $x \neq y$ then we need two fresh names in order to propagate the metasubstitution inside the abstractions on each side of the α-equation. Let x_0 be a fresh name not in the set $fv\_nom(u) \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$, and x_1 be a fresh name not in the set $fv\_nom(u') \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$. After propagating the metasubstitution we need to prove $\lambda_{x_0}.\{x := u\}((y\ x_0)t_1) =_\alpha \lambda_{x_1}.\{x := u'\}((y\ x_1)t_1)$, and we proceed by comparing x_0 and x_1:
   (a) If $x_0 = x_1$ then we are done by the induction hypothesis.
   (b) Otherwise, we need to apply the rule *aeq_abs_diff* and the goal is $\{x := u\}((y\ x_0)t_1) =_\alpha (x_0\ x_1)(\{x := u'\}((y\ x_1)t_1))$. But in order to proceed we need to know how to propagate the swap inside the metasubstitution, which is the content of the following lemma:

`Lemma` *swap_m_subst*: $\forall\ t\ u\ x\ y\ z$, *swap* $y\ z\ (\{x := u\}t) =a (\{(vswap\ y\ z\ x) := (swap\ y\ z\ u)\}(swap\ y\ z\ t))$.

Proof. We write the statement of the lemma in metanotation before starting the proof:

$$\forall t\ u\ x\ y\ z, (y\ z)(\{x := u\}t) =_\alpha \{(\!|y\ z|\!)x := (y\ z)u\}(y\ z)t$$

The proof is by induction on the size of the term t, and again we will focus only on the abstraction case, *i.e.* when $t = \lambda_w.t_1$. The goal in this case is $(y\ z)(\{x := u\}(\lambda_w.t_1)) =_\alpha \{(\!|y\ z|\!)x := (y\ z)u\}((y\ z)\lambda_w.t_1)$, and we proceed by comparing x and w.

1. If $x = w$ the α-equality is trivial.
2. If $x \neq w$ then we need a fresh name, say w_0, to be able to propagate the metasubstitution inside the abstraction on the LHS of the α-equation. The variable w_0 is taken such that it is not in the set $fv\_nom(u) \cup fv\_nom(\lambda_w.t_1) \cup \{x\}$, and we get the goal $\lambda_{(\!(y\ z)\!)w_0}.(y\ z)(\{x := u\}(w\ w_0)t_1) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}(\lambda_{(\!(y\ z)\!)w}.(y\ z)t_1)$. Now we propagate the metasubstitution over the abstraction in the RHS of the goal. Since $x \neq w$ implies $(\!(y\ z)\!)x \neq (\!(y\ z)\!)w$, we need another fresh name, say w_1, not in the set $fv\_nom((y\ z)u) \cup fv\_nom(\lambda_{(\!(y\ z)\!)w}.(y\ z)t_1) \cup \{(\!(y\ z)\!)x\}$, and after the propagation we need to prove that $\lambda_{(\!(y\ z)\!)w_0}.(y\ z)(\{x := u\}(w\ w_0)t_1) =_\alpha \lambda_{w_1}.\{(\!(y\ z)\!)x := (y\ z)u\}((w_1\ (\!(y\ z)\!)w)((y\ z)t_1))$. We consider two cases: either $w_1 = (\!(y\ z)\!)w_0$ or $w_1 \neq (\!(y\ z)\!)w_0$. In the former case, we apply the rule *aeq_abs_same* and we are done by the induction hypothesis. When $w_1 \neq (\!(y\ z)\!)w_0$, the application of the rule *aeq_abs_diff* generates the goal

$$(w_1\ (\!(y\ z)\!)w_0)(y\ z)(\{x := u\}(w\ w_0)t_1) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}((w_1\ (\!(y\ z)\!)w)((y\ z)t_1)) \quad (6)$$

   We can use the induction hypothesis to propagate the swap inside the metasubstitution, and then we get an α-equality with metasubstitution as main operation on both sides, whose corresponding components are α-equivalent. In a more abstract way, we have to prove an α-equality of the form $\{x := u\}t =_\alpha \{x := u'\}t'$, where $t =_\alpha t'$ and $u =_\alpha u'$, but this is exactly what we were trying to prove in the previous lemma.

Therefore, we are in a circular problem because both *aeq_m_subst_in_trial* and *swap_m_subst* depend on each other to be proved!

Our solution to this problem consists in taking advantage of the fact that α-equivalent terms have the same set of free variables (see lemma *aeq_fv_nom*), and noting that the external swap in the LHS of (6) was generated by the application of the rule *aeq_abs_diff* because the abstractions have different bindings. Let's go back to the proof of lemma *aeq_m_subst_in*: `Lemma` *aeq_m_subst_in*: $\forall$ *t u u' x*, *u* =*a u'* $\rightarrow$ ({*x* := *u*}*t*) =*a* ({*x* := *u'*}*t*).

Proof. We go directly to the abstraction case. When $t = \lambda_y.t_1$, the goal is $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u'\}(\lambda_y.t_1)$. If $x \neq y$ then the fresh name needed for the LHS must not belong to the set $fv\_nom(u) \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$, while the fresh name for the RHS must not belong to $fv\_nom(u') \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$. These sets differ only by the subsets $fv\_nom(u)$ and $fv\_nom(u')$. Nevertheless, these subsets are equal because u and u' are α-equivalent (see lemma *aeq_fv_nom*). Concretely, the current goal is as follows:

```
(let (z, _) := atom_fresh (union (fv_nom u) (union (fv_nom (n_abs y t1))
       (singleton x))) in n_abs z (subst_rec_fun (swap y z t1) u x)) =a
(let (z, _) := atom_fresh (union (fv_nom u') (union (fv_nom (n_abs y t1))
         (singleton x))) in n_abs z (subst_rec_fun (swap y z t1) u' x))
```

where the sets $fv\_nom(u)$ and $fv\_nom(u')$ appear in different *let* expressions, each one is responsible for generating one fresh name. But since these sets are equal, if one could replace $fv\_nom(u)$ by $fv\_nom(u')$ (or vice-versa) then only one fresh name is generated after evaluating the *atom_fresh* function. Nevertheless, the only way that we managed to do such replacement was by adding the following axiom:

```
Axiom Eq_implies_equality: forall t1 t2, t1 =a t2 -> fv_nom t1 = fv_nom t2.
```

This axiom is similar to lemma *aeq_fv_nom* where the set equality [=] was replaced by the syntactic (Leibniz) equality =. Now, we can generate just one fresh name and propagate the metasubstitution on both sides of the goal, and we are done by the induction hypothesis. The case of the explicit substitution is similar, and with this strategy we avoid both the rules *aeq_abs_diff* and *aeq_sub_diff* that introduce swappings. □

The next lemma, named *aeq_m_subst_out* will benefit the strategy used in the previous proof, but it is not straightfoward.

`Lemma` *aeq_m_subst_out*: $\forall\ t\ t'\ u\ x,\ t =a\ t' \rightarrow (\{x := u\}t) =a (\{x := u\}t')$.

Proof. The proof is by induction on the size of the term t. Note that induction on the hypothesis $t =a\ t'$ does not work due to a similar problem involving swaps that appears when structural induction on t is used. The abstraction and the explicit substitution are the interesting cases.

In the abstraction case, we need to prove that $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u\}t'$, where $\lambda_y.t_1 =_\alpha t'$ by hypothesis. Therefore, t' must be an abstraction, and according to our definition of α-equivalence there are two possible subcases:

1. In the first subcase, $t' = \lambda_y.t_2$, where $t_1 =_\alpha t_2$, and hence the current goal is $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u\}(\lambda_y.t_2)$. We proceed by comparing x and y:
   (a) If $x = y$ then, we are done by using twice lemma *m_subst_abs_eq*.
   (b) When $x \neq y$, then we need to propagate the metasubstitution on both sides of the goal. On the LHS, we need a fresh name that is not in the set $fv(u) \cup fv(\lambda_y.t_1) \cup \{x\}$, while for the RHS, the fresh name cannot belong to the set $fv(u) \cup fv(\lambda_y.t_2) \cup \{x\}$. From the hypothesis $t_1 =_\alpha t_2$, we know, by lemma *aeq_fv_nom*, that the sets $fv\_nom(t_1)$ and $fv\_nom(t_2)$ are equal. Therefore, we can take just one fresh name, say z, and propagate both metasubstitutions over abstractions with the same binding, and we conclude with the induction hypothesis.
2. In the second subcase, $t' = \lambda_{y_0}.t_2$, where $t_1 =_\alpha (y_0\ y)t_2$ and $y \neq y_0$. The current goal is
   $$\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u\}(\lambda_{y_0}.t_2)$$
   and we proceed by comparing x and y:
   (a) If $x = y$ then the goal simplifies to $\lambda_y.t_1 =_\alpha \{x := u\}(\lambda_{y_0}.t_2)$ by lemma *m_subst_abs_eq*, and we pick a fresh name x, that is not in the set $fv\_nom(u) \cup fv\_nom(\lambda_{y_0}.t_2) \cup \{y\}$, and propagate the metasubstitution on the RHS of the goal, resulting in the new goal $\lambda_y.t_1 =_\alpha \lambda_x.\{y := u\}((y_0\ x)t_2)$. Note that the metasubstitution on the RHS has no effect in the term $(y_0\ x)t_2$ because $y \neq y_0$, $y \neq x$ and y does not occur free in t_2 and we conclude by hypothesis.
   (b) If $x \neq y$ then we proceed by comparing x and y_0 on the RHS, and the proof, when $x = y_0$, is analogous to the previous subcase. When both $x \neq y$ and $x \neq y_0$ then we need to propagate the metasubstitution on both sides of the goal $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u\}(\lambda_{y_0}.t_2)$. We have that $\lambda_y.t_1 =_\alpha \lambda_{y_0}.t_2$ and hence the sets $fv\_nom(\lambda_y.t_1)$ and $fv\_nom(\lambda_{y_0}.t_2)$ are equal. Therefore, only one fresh name, say x_0, that is not in the set $x_0 \notin fv\_nom(u) \cup fv\_nom(\lambda_{y_0}.t_2) \cup \{x\}$ is enough to fulfill the conditions for propagating the metasubstitutions on both sides of the goal, and we are done by the induction hypothesis.
3. The explicit substitution operation is also interesting, but we will not comment because we are running out of space. □

As a corollary, one can join the lemmas *aeq_m_subst_in* and *aeq_m_subst_out* as follows:

`Corollary` *aeq_m_subst_eq*: $\forall\ t\ t'\ u\ u'\ x,\ t =a\ t' \rightarrow u =a\ u' \rightarrow (\{x := u\}t) =a\ (\{x := u'\}t')$.

Now, we show how to propagate a swap inside metasubstitutions using the decomposition of the metasubstitution provided by the corollary *aeq_m_subst_eq*.

`Lemma` *swap_subst_rec_fun*: $\forall\ x\ y\ z\ t\ u$, *swap x y* $(\{z := u\}t) =a\ (\{(vswap\ x\ y\ z) := (swap\ x\ y\ u)\}(swap\ x\ y\ t))$.

Proof. Firstly, we write the lemma in metanotation: $\forall x\,y\,z\,t\,u, (x\,y)\{z := u\}t =_\alpha \{(\!(x\,y)\!)z := (x\,y)u\}(x\,y)t$. Next, we compare x and y, since the case $x = y$ is trivial. When $x \neq y$, the proof proceeds by induction on the size of the term t. The tricky cases are the abstraction and explicit substitution, and we comment just the former case. If $t = \lambda_{y'}.t_1$ then we must prove that $(x\ y)\{z := u\}(\lambda_{y'}.t_1) =_\alpha \{(\!(x\ y)\!)z := (x\ y)u\}(x\ y)(\lambda y'.t_1)$. Firstly, we compare the variables y' and z according to the definition of the metasubstitution:

1. When $y' = z$ the metasubstitution is erased according to the definition (5) on both sides of the goal and we are done.
2. When $y' \neq z$ then the metasubstitutions on both sides of the goal need to be propagated inside the corresponding abstractions. In order to do so, a new name need to be created. Note that in this case, it is not possible to create a unique name for both sides because the two sets are different. In fact, in the LHS the fresh name cannot belong to the set $fv\_nom(\lambda'_y.t_1) \cup fv\_nom(u) \cup \{z\}$, while the name of the RHS cannot belong to the set $fv\_nom((x\ y)\lambda'_y.t_1) \cup fv\_nom((x\ y)u) \cup \{(\!(x\ y)\!)z\}$. Let x_0 be a fresh name that is not in the set $fv\_nom(\lambda'_y.t_1) \cup fv\_nom(u) \cup \{z\}$, and x_1 a fresh name that is not in the set $fv\_nom((x\ y)\lambda'_y.t_1) \cup fv\_nom((x\ y)u) \cup \{(\!(x\ y)\!)z\}$. After the propagation of the metasubstitutions, we have to prove that $\lambda_{(\!(x\ y)\!)x0}.((x\ y)(\{z := u\}((y'\ x_0)t_1)) =_\alpha \lambda_{x_1}.(\{(\!(x\ y)\!)z := (x\ y)u\}(((\!(x\ y)\!)y')\ x_1)((x\ y)t_1))$. We proceed by comparing x_1 with $(\!(x\ y)\!)x_0$.
   (a) If $x_1 = (\!(x\ y)\!)x_0$ then we use the induction hypothesis to propagate the swap inside the metasubstitution in the LHS, and we get the goal $\{(\!(x\ y)\!)z := (x\ y)u\}((x\ y)((y'\ x_0)t_1)) =_\alpha \{(\!(x\ y)\!)z := (x\ y)u\}((((\!(x\ y)\!)y')\ ((\!(x\ y)\!)x_0))((x\ y)t_1))$ that is proved by the swap equivariance lemma *swap_equivariance*.
   (b) If $x_1 \neq (\!(x\ y)\!)x_0$ then by the rule *aeq_abs_diff* we have to prove that the variable $(\!(x\ y)\!)x_0$ is not in the set of free variables of the term $\{(\!(x\ y)\!)z := (x\ y)u\}(((\!(x\ y)\!)y'\ x_1)(x\ y)t_1)$ and that $(x\ y)(\{z := u\}((y'\ x_0)t_1)) =_\alpha (x_1\ ((\!(x\ y)\!)x_0))(\{(\!(x\ y)\!)z := (x\ y)u\}((((\!(x\ y)\!)y')\ x_1)((x\ y)t_1))$. The former condition is routine. The later condition is proved using the induction hypothesis twice to propagate the swaps inside the metasubstitutions on each side of the α-equality. This swap has no effect on the variable z of the metasubstitution because x_1 is different from $(\!(x\ y)\!)z$, and x_0 is different from z. Therefore we can apply lemma *aeq_m_subst_eq*, and each generated case is proved by routine manipulation of swaps.

□

The following two lemmas toghether with lemmas *m_subst_abs_eq* and *m_subst_sub_eq* are essential in simplifying the propagations of metasubstitution. They are presented here because they depend on lemma *swap_subst_rec_fun*.

`Lemma` *m_subst_abs_neq*: $\forall\ t\ u\ x\ y\ z,\ x \neq y \rightarrow z$ ‘*notin*‘ *fv_nom u* ‘*union*‘ *fv_nom* (*n_abs y t*) ‘*union*‘ $\{\{x\}\} \rightarrow \{x := u\}(n\_abs\ y\ t) =a\ n\_abs\ z\ (\{x := u\}(swap\ y\ z\ t))$.

Lemma *m_subst_sub_neq* : $\forall$ *t1 t2 u x y z*, $x \neq y \rightarrow z$ ‘*notin*‘ *fv_nom u* ‘*union*‘ *fv_nom* ([$y := t2$]*t1*) ‘*union*‘ $\{\{x\}\} \rightarrow \{x := u\}([y := t2]t1) =_a ([z := (\{x := u\}t2)](\{x := u\}(swap\ y\ z\ t1)))$.

In the pure λ-calculus, the substitution lemma is probably the first non trivial property. In our framework, we have defined two different substitution operators, namely, the metasubstitution denoted by $\{x := u\}t$ and the explicit substitution, written as $[x := u]t$. In what follows, we present the main steps of our proof of the substitution lemma for *n_sexp* terms, *i.e.* for nominal terms with explicit substitutions.

Lemma *m_subst_lemma*: $\forall$ *t1 t2 t3 x y*, $x \neq y \rightarrow x$ ‘*notin*‘ (*fv_nom t3*) $\rightarrow$
$$(\{y := t3\}(\{x := t2\}t1)) =_a (\{x := (\{y := t3\}t2)\}(\{y := t3\}t1)).$$

Proof. The proof is by induction on the size of *t1*. The interesting cases are the abstraction and the explicit substitution. We focus on the former, *i.e.* $t1 = \lambda_z.t_1'$, whose initial goal is

$\{y := t_3\}(\{x := t_2\}(\lambda_z.t_1')) =_\alpha \{x := \{y := t_3\}t_2\}(\{y := t_3\}(\lambda_z.t_1'))$

assuming that $x \neq y$ and $x \notin fv\_nom(t_3)$. The induction hypothesis generated by this case states that the lemma holds for any term of the size of t_1', *i.e.* any term with the same size of the body of the abstraction. We start comparing z with x aiming to apply the definition of the metasubstitution on the LHS of the goal.

1. When $z = x$, the subterm $\{x := t_2\}\lambda_x.t_1'$ reduces to $\lambda_x.t_1'$ by lemma *m_subst_abs_eq*, and then the LHS reduces to $\{y := t_3\}\lambda_x.t_1'$. The RHS $\{x := \{y := t_3\}t_2\}\{y := t_3\}\lambda_x.t_1'$ also reduces to it because x does not occur free neither in $\lambda_x.t_1'$ nor in t_3, and we are done.
2. When $z \neq x$, then we compare y with z.
   (a) When $y = z$, the subterm $\{y := t_3\}(\lambda_z.t_1')$ can be simplified to $\lambda_z.t_1'$, by lemma *m_subst_abs_eq*. On the LHS, we propagate the internal metasubstitution over the abstraction taking a fresh name w not in the set $fv\_nom(\lambda_z.t_1') \cup fv\_nom(t_3) \cup fv\_nom(t_2) \cup \{x\}$, where the goal is $\{z := t_3\}(\lambda_w.(\{x := t_2\}(z\ w)t_1')) =_\alpha \{x := \{z := t_3\}t_2\}(\lambda_z.t_1')$. We proceed by comparing z and w:
      i. If $z = w$ then the current goal simplifies to
      $\{w := t_3\}(\lambda_w.(\{x := t_2\}t_1')) =_\alpha \{x := \{w := t_3\}t_2\}(\lambda_w.t_1')$
      We can propagate the metasubstitution on the RHS and there is no need for a fresh name since the variable w fullfil the condition required by lemma *m_subst_abs_neq*. We conclude with lemmas *aeq_m_subst_in* and *m_subst_notin*.
      ii. If $z \neq w$ then we can propagate the metasubstitutions on both sides of the goal taking w as the fresh name that fullfil the conditions of lemma *m_subst_abs_neq*. We proceed with *aeq_abs_same*, and conclude by the induction hypothesis.
   (b) If $y \neq z$ then we follow a similar strategy that avoids unnecessary generation of fresh names. In this way, we take a fresh w that is not in the set $fv\_nom(t_3) \cup fv\_nom(t_2) \cup fv\_nom(\lambda_z.t_1') \cup \{x\} \cup \{y\}$, and propagate the metasubstitution inside the abstraction resulting in the goal $\lambda_w.(\{y := t_3\}(\{x := t_2\}(z\ w)t_1')) =_\alpha \lambda_w.(\{x := \{y := t_3\}t_2\}(\{y := t_3\}(z\ w)t_1')$. We conclude by the induction hypothesis. □

4 Conclusion and Future work

In this work, we presented a formalization of the substitution lemma in a framework that extends the λ-calculus with an explicit substitution operator. Calculi with explicit substitutions are important frameworks to study properties of the λ-calculus and have been extensively studied in the last decades [1, 2, 3, 4, 9].

The formalization is modular in the sense that the explicit substitution operator is generic and could be instantiated with any calculi with explicit substitutions in a nominal setting. Despite the fact that our definition of metasubstitution, called *subst_rec_fun*, performs a renaming with a fresh name whenever it is propagated inside a binding structure (either an abstraction or an explicit substitution in our case), we showed how to avoid unnecessary generation of fresh names that could result in a circular problem in the proofs. Several auxiliary (minor) results were not included in this document, but they are numerous and can be found in the source file of the formalization that is publicly available at `https://flaviomoura.info/files/msubst.v`

As future work, we intend to get rid of the axiom *Eq_implies_equality*. The natural candidate for this would be the use of generalized rewriting, *i.e.* setoid rewriting, but it not clear whether generalized rewriting allows a rewrite step in a let expression. Another possibility is the implementation of the metasubstitution using recursors [22, 13]. In addition, we plan to integrate this formalization with another one related to the Z property[7] to prove confluence of calculi with explicit substitutions [20, 21], as well as other properties in the nominal framework [16].

References

[1] M. Abadi, L. Cardelli, P.-L. Curien & J.-J. Lévy (1991): *Explicit Substitutions*. Journal of Functional Programming 1(4), pp. 375–416, doi:10.1017/S0956796800000186.

[2] Beniamino Accattoli (2012): *An Abstract Factorization Theorem for Explicit Substitutions*, p. 16 pages. doi:10.4230/LIPICS.RTA.2012.6.

[3] Mauricio Ayala-Rincón, Flávio L.C. De Moura & Fairouz Kamareddine (2002): *Comparing Calculi of Explicit Substitutions with Eta-reduction*. Electronic Notes in Theoretical Computer Science 67, pp. 76–95, doi:10.1016/S1571-0661(04)80542-5.

[4] Mauricio Ayala-Rincón, Flávio L.C. De Moura & Fairouz Kamareddine (2005): *Comparing and Implementing Calculi of Explicit Substitutions with Eta-Reduction*. Annals of Pure and Applied Logic 134(1), pp. 5–41, doi:10.1016/j.apal.2004.06.009.

[5] H. P. Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*, rev. ed edition. Studies in Logic and the Foundations of Mathematics v. 103, North-Holland ; Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, Amsterdam ; New York : New York, N.Y.

[6] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne & Jocelyne Rouyer-Degli (1996): *Λv, a Calculus of Explicit Substitutions Which Preserves Strong Normalisation*. Journal of Functional Programming 6(5), pp. 699–722, doi:10.1017/S0956796800001945.

[7] Stefan Berghofer & Christian Urban (2007): *A Head-to-Head Comparison of de Bruijn Indices and Names*. Electronic Notes in Theoretical Computer Science 174(5), pp. 53–67, doi:10.1016/j.entcs.2007.01.018.

[8] Roel Bloo & Herman Geuvers (1999): *Explicit Substitution: On the Edge of Strong Normalization*. Theoretical Computer Science 211(1-2), pp. 375–395, doi:10.1016/s0304-3975(97)00183-7.

[9] E. Bonelli (2001): *Perpetuality in a Named Lambda Calculus With Explicit Substitutions*. Mathematical Structures in Computer Science 11(1), pp. 47–90, doi:10.1017/s0960129500003248.

[7] `https://cicm-conference.org/2021/cicm.php?event=fmm&menu=general`

[10] Pierre-Louis Curien, Thérèse Hardin & Jean-Jacques Lévy (1996): *Confluence Properties of Weak and Strong Calculi of Explicit Substitutions*. *Journal of the ACM* 43(2), pp. 362–397, doi:10.1145/226643.226675.

[11] R. David & B. Guillaume (2001): *A Lambda-Calculus with Explicit Weakening and Explicit Substitution*. *Mathematical Structures in Computer Science* 11(1), pp. 169–206, doi:10.1017/S0960129500003224.

[12] Murdoch J. Gabbay & Andrew M. Pitts (2002): *A New Approach to Abstract Syntax with Variable Binding*. *Formal Aspects of Computing* 13(3-5), pp. 341–363, doi:10.1007/s001650200016.

[13] Lorenzo Gheri & Andrei Popescu (2020): *A Formalized General Theory of Syntax with Bindings: Extended Version*. *Journal of Automated Reasoning* 64(4), pp. 641–675, doi:10.1007/s10817-019-09522-2.

[14] Bruno Guillaume (2000): *The λs_e-Calculus Does Not Preserve Strong Normalisation*. *Journal of Functional Programming* 10(4), pp. 321–325, doi:10.1017/S0956796800003695.

[15] Fairouz Kamareddine & Alejandro Ríos (1997): *Extending a λ-Calculus with Explicit Substitution Which Preserves Strong Normalisation into a Confluent Calculus on Open Terms*. *Journal of Functional Programming* 7(4), pp. 395–420, doi:10.1017/S0956796897002785.

[16] D. Kesner (2008): *Perpetuality for Full and Safe Composition (in a Constructive Setting)*. In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, pp. 311–322, doi:10.1007/978-3-540-70583-3_26.

[17] Delia Kesner (2009): *A Theory of Explicit Substitutions with Safe and Full Composition*. *Logical Methods in Computer Science* Volume 5, Issue 3, p. 816, doi:10.2168/LMCS-5(3:1)2009.

[18] Paul-André Mellies (1995): *Typed λ-Calculi with Explicit Substitutions May Not Terminate*. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Mariangiola Dezani-Ciancaglini & Gordon Plotkin, editors: *Typed Lambda Calculi and Applications*, 902, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 328–334, doi:10.1007/BFb0014062.

[19] C. A. Muñoz (1996): *Confluence and Preservation of Strong Normalisation in an Explicit Substitutions Calculus*. In: *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pp. 440–447, doi:10.1109/LICS.1996.561460.

[20] Koji Nakazawa & Ken-etsu Fujita (2016): *Compositional Z: Confluence Proofs for Permutative Conversion*. *Studia Logica* 104(6), pp. 1205–1224, doi:10.1007/s11225-016-9673-0.

[21] Koji Nakazawa, Ken-etsu Fujita & Yuta Imagawa (2023): *Z Property for the Shuffling Calculus*. *Mathematical Structures in Computer Science*, pp. 1–13, doi:10.1017/S0960129522000408.

[22] Andrei Popescu (2023): *Nominal Recursors as Epi-Recursors*. arXiv:2301.00894.

[23] K. H. Rose, R. Bloo & F. Lang (2011): *On Explicit Substitution With Names*. *J Autom Reasoning* 49(2), pp. 275–300, doi:10.1007/s10817-011-9222-5.

[24] The Coq Development Team (2021): *The Coq Proof Assistant*. Zenodo, doi:10.5281/ZENODO.5704840.

While Loops in Coq

David Nowak, CNRS* Vlad Rusu, Inria†

While loops are present in virtually all imperative programming languages. They are important both for practical reasons (performing a number of iterations not known in advance) and theoretical reasons (achieving Turing completeness). In this paper we propose an approach for incorporating while loops in an imperative language shallowly embedded in the Coq proof assistant. The main difficulty is that proving the termination of while loops is nontrivial, or impossible in the case of non-termination, whereas Coq only accepts programs endowed with termination proofs. Our solution is based on a new, general method for defining possibly non-terminating recursive functions in Coq. We illustrate the approach by proving termination and partial correctness of a program on linked lists.

1 Introduction

The definition of recursive functions in the Coq proof assistant [4] is subject to certain restrictions to ensure their termination, which is essential for the consistency of Coq's underlying logic. Specifically, recursive calls must be made on strict subterms, effectively ensuring that the computation eventually reaches a base case. Alternatively, users have the option to prove that a specific quantity strictly decreases according to a well-founded order. In such cases, Coq can automatically transform the recursive calls into strict subterm calls, using a so-called accessibility proof to guarantee termination. Adhering to these constraints eliminates the risk of infinitely many calls, thereby ensuring that functions terminate.

An alternative, somewhat ad-hoc strategy is to introduce an additional natural-number argument called the *fuel*. The fuel's value is decremented with each recursive call, thereby guaranteeing finitely many recursive calls, hence, termination. However, a crucial concern arises as one must supply enough fuel so that termination does not disrupt the intended computation of the program by occurring too early.

In this paper we present a novel approach to defining possibly partial recursive functions in Coq while achieving separation of concerns: write the program first, and prove its properties (including termination) later. In broad terms our technique consists in providing an infinite amount of fuel for recursive functions. By doing so the function can proceed with its computations without risk of exhausting its fuel.

As a result, this approach empowers developers to focus on the core logic of the recursive function, separate from the termination concern, streamlining the development process and enhancing the modularity and readability of the code.

A key property guaranteed by our technique is that, given the *functional* of the recursive function under definition (i.e., an abstract description of the function's body), the resulting function is the *least fixpoint* of its functional. We prove this general result under mild constraints on the functional - it must be monotonic and, in some sense described precisely in the paper, must preserve continuity.

The method is applied to while loops in an imperative shallowly embedded in Coq. By proving that the functional of while loops is monotonic and continuity-preserving we obtain while loops as least fixpoints of their functionals. This enables programmers to construct imperative programs featuring

*Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
†Univ. Lille, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 96–109, doi:10.4204/EPTCS.389.8

arbitrary while loops and provides them with tools for reasoning about loops. Specifically, the least-fixpoint property is used for proving that a while loop terminates if and only if there exists some finite amount of fuel for which it returns the desired result; hence, a termination proof can proceed by induction on the fuel value, once an adequate instantiation for this known-to-exists value is chosen.

Subsequently, we proceed to establish a Hoare logic system, which serves as a formal tool for proving the partial correctness of programs. In essence the Hoare logic provides a systematic and rigorous approach to program verification, where one defines preconditions and postconditions that govern the state of the program before and after its execution. Through these assertions one can verify that the program's execution leads to the desired outcomes, establishing its partial correctness. Here, again, the property of being a fixpoint is used for proving the soundness of the while loop's *Hoare triple*.

Finally, the least-fixpoint property of the partial functions being defined ensures that the functions, defined abstractly using order theory, are, as mathematical functions, the same as the ones that Coq would have generated, had it not been constrained by its logic into rejecting the functions as partial.

Outline In Section 2 we introduce the reader, termination, and state monads, which serve the purpose of writing imperative programs in the Coq proof assistant. Moving forward to Section 3 we present our method for defining possibly nonterminating recursive functions and demonstrate its application to the definition of while loops. In Section 4 we define a monadic Hoare logic and illustrate its effectiveness by applying it to a program that computes the length of a linked list. In Section 5 we address the issue of proving termination. We compare with related work in Section 6 and conclude in Section 7.

The Coq development corresponding to this paper is available at `https://tinyurl.com/2p93uwdj`.

2 Monads for Possibly Nonterminating Stateful Computation

We consider a subset of Gallina (the programming language of Coq) expressive enough for shallowly embedding possibly nonterminating imperative programs. In purely functional languages such as Gallina the usual approach is to encode imperative features with monads [11]. We use a combination of the termination, state, and reader monads. The first one is used for possibly nonterminating computations, the second one for stateful computations where the state may change, and the third one for state-aware computations that do not change the state, such as checking the condition of while loops.

A monad consists of a type constructor equipped with an operation usually called `ret` for trivial computation, and another one, usually called `bind`, for sequencing computations. Each particular monad also comes with specific operations. Assume a context where a type `T` is declared. The type constructor for the reader monad is defined as follows:

```
Definition reader (A: Type): Type := T -> A.
```

where `A` is a type for values returned by a computation. Intuitively, `reader` is a side-effect free function from states to computed values. Unlike the state monad, shown below, it only reads from states, without modifying them; hence the monad's name `reader`.

Trivial computation `ret` consists in ignoring the state:

```
Definition ret {A: Type}(a: A) : reader A := fun _ => a.
```

Placing the parameter `A` between curly brackets marks it as *implicit* - its value can therafter be automatically inferred, relieving users from the burden of having to instantiate it.

Sequencing of computations consists in passing the result of the first computation to the second one:

```
Definition rbind {A B: Type}(m: reader A)(f: A -> reader B): reader B :=
fun s => f (m s) s.
```

Finally, the reader monad is equipped with a primitive for reading the state:

```
Definition get: reader T := fun s => s.
```

The other monad used in the shallow embedding of an imperative language in Coq is a combination of termination and state monads, for which we use the `option` type constructor:

```
Inductive option (X: Type): Type := None: option X | Some: X -> option X.
```

where `None` encodes nontermination. Termination state monads are the programs of our imperative language, which is reflected in the name of its type constructor:

```
Definition program (A: Type): Type := T -> option (A * T).
```

The reader monad is a special case of the termination state monad. Thus we introduce a *coercion* that enables Coq to automatically convert a reader monad into a termination state monad when needed:

```
Coercion reader_to_program {A: Type}(m: reader A): program A :=
fun s => Some (m s, s).
```

The `ret` and the `get` primitive of the reader monad are thus automatically converted. But the sequencing of computations needs to be redefined in order the take into account the fact that the first of the sequenced computations might change the state, or might not terminate:

```
Definition bind {A B: Type}(m: program A)(f: A -> program B): program B :=
fun s => match m s with None => None | Some (a, s') => f a s' end.
```

The termination state monad comes with an additional primitive to change the state:

```
Definition put (s: T) : program unit := fun _ => Some (tt, s).
```

The `unit` type (inhabited by one term - the constant `tt`) is used in our functional-laaguage setting for modelling imperative programs that do not return anything, encoded by returning the dummy constant `tt`.

As a running example we consider an imperative program that computes the length of a linked list. First we need to specify the `State` on which the program works. We assume a machine with two positive integer registers and an unbounded memory whose adresses and stored values are also positive integers:

```
Record State: Type := {reg1: nat; reg2: nat; memory: nat -> nat}.
```

Then, using the primitives of our monad we write operations to read/write in registers and memory:

- `read_reg1`, `write_reg1` for reading, resp. writing in the first register, and similar functions for reading/writing in the second register. The second register is increased by one using `incr_reg2`;
- `do next <- read_addr curr` assigns to `next` the content of the address `curr` in memory. More generally, reading operations return a value that can be bound to an identifier using the standard "do" notation of monads, i.e., `do x <- m; f x` is a shortcut for `rbind m (fun x => f x)`;
- operations are sequenced by double semicolons: `m;;f` is a notation for `bind m (fun _ => f)`.

This almost allows us to write a program computing the length of a linked list. Not completely, because the program uses a `while` loop, which is not defined until later in the paper:

```
Definition length (addr: nat): program State nat :=
write_reg1 addr;;
write_reg2 0;;
while (do curr <- read_reg1; ret  (curr != 0))
  (
   incr_reg2;;
   do curr <- read_reg1;
   do next <- read_addr curr;
   write_reg1 next
  );;
do res <- read_reg2; ret res.
```

It is assumed that the linked list[1] of interest starts in memory at address `addr`. This address is written into the first register, then the second register (which is to contain the length of the list) is initialized to zero. Next, while the current address of the first register is not `null` (also encoded by zero), the second register is incremented and the first register is updated to point to the next element of the linked list. Finally, at the end of the `while` loop (if the end is ever reached), the value in the second register is the length of the list of interest, hence, it is the value returned by our function whenever it terminates.

What is still missing is, of course, the definition of the `while` loop. A first attempt uses recursion:

```
Fixpoint
 while{T: Type}(cond: reader T bool)(body: program T unit): program T unit:=
  do c <- cond;if c then body;;while cond body else reader_to_program (ret tt)
```

That is, a recursive function (introduced by the keyword `Fixpoint`) attempts to define a `while` loop (with condition `cond` and body `body` of appropriate types) by first checking the condition, and if the condition holds, executing the body then recursively the `while` loop; otherwise, doing nothing (which is encoded by `reader_to_program (ret tt)`). However, this function does not always terminate. For example, if the `while` loop is used to navigate a linked list, like in the case of the `length` function above, and the list is badly linked, i.e., it contains a loop, then the `while` loop does not terminate. Coq rejects this definition attempt, as it rejects any recursive function whose termination it cannot infer.

In the rest of the paper we show how possibly infinite while loops (and, in general, partial recursive function) can be accepted by Coq by encoding nontermination as evaluation to a special value.

3 Partial Recursive Functions in Coq

The `while` loop is a particular case of a partial recursive function. We first sketch how partial recursive functions can be encoded in Coq before providing details.

[1] For simplicity we consider linked lists where each element only contains the next element's address.

3.1 Outline of the Approach

Assume we want to define a partial recursive function `f` from type `A` to type `B`. A natural way to proceed is to give `f` the type `A -> option B`, where for any `a:A`, `f a = None` encodes the fact that `f` is undefined for the input `a`. In order to define a function we further need its *functional*, an abstract representation of the body of the function being defined. Let `F : (A -> option B)-> A -> option B` be the functional for `f`. We say that `f := F f` is the *fixpoint definition* of `f`. The interesting case we here solve is when fixpoint definitions are not accepted by Coq - just like in the case of the `while` function above.

We proceed as follows. We define an auxiliary function `f_fuel : nat -> A -> option B` with an additional natural-number parameter called the *fuel*, as the following recursive function, which is accepted by Coq because Coq "sees" that the `fuel` parameter strictly decreases at each recursive call:

```
Fixpoint f_fuel (fuel: nat) (a: A) : option B:=
match fuel with
|S fuel' => F(f_fuel fuel') a
    (*S is the successor function on natural numbers*)
|0 => None
end
```

If the functional `F` is *monotonic* then, based on results in order theory explained later in this section, the function `f_fuel` can be *lifted* to a *continuous* function in `conat -> option B` where `conat` is the type

```
Inductive conat: Type:= finite: nat -> conat | infinity
```

That is, the inhabitants of `conat` are natural numbers wrapped with the `finite` constructor, together with the constant `infinity`. Putting back the parameter `a:A` in the type we obtain a function `f_inf` of the type `conat -> A -> option B`. The results later in the section also ensure that, under an additional condition on `F` (preservation of continuity), the function `(f_inf infinity)` is the *least fixpoint* of `F`.

Recapitulating, we started with the intention of defining a function `f : A -> option B`, using its functional `F : (A -> option B)-> A -> option B`, via the fixpoint definition `f := F f`. We have assumed this is rejected by Coq. Per the results below we define `f:= f_inf infinity` and prove that is the least solution of the fixpoint equation `f = F f` - precisely the solution that Coq would have constructed had it accepted the definition `f := F f` - with the advantage that our definition *is* accepted.

3.2 Elements of Order Theory

The results in this subsection have been adapted from the textbook [2]. We have formalized them in Coq, hence, hereafter proofs are only sketched or omitted altogether. The examples are not only used for illustration purposes: they also serve as building blocks in our approach to partial recursive functions.

Definition 1 *A* pointed partial order *(PPO)* $(S, \preceq, \bot)$ *is a partially ordered set* $(S, \preceq)$ *together with a distinguished element* $\bot \in S$ *such that for all* $s \in S$, $\bot \preceq s$.

Example 1 The triple $(\mathbb{N}, \leq, 0)$ consisting of natural numbers $\mathbb{N}$, their usual order $\leq$, and the least natural number 0 form a PPO.

Example 2 For any set A, the triple $(A \cup \{\bot\}, \preceq, \bot)$ with $\bot \notin A$, and $\preceq$ being defined as the smallest relation on $A \cup \{\bot\}$ such that $\bot \preceq a$ and $a \preceq a$ for all $a \in A$, is a PPO called the *flat PPO of A*.

In Coq the flat PPO of a type `A` is encoded using the type `option A` where `None` plays the role of $\bot$.

Definition 2 *Given a PPO $(S, \preceq, \bot)$, a set $S' \subseteq S$ is* directed *if $S \neq \emptyset$ and for all $x, y \in S$ there exists $z \in S$ such that $x, y \preceq z$.*

Example 3 Any nonempty set of natural numbers in the PPO $(\mathbb{N}, \leq, 0)$ is directed.

The above example is a consequence of the more general fact that any nonempty sequence, i.e., totally ordered subset of elements in a PPO, is directed. Indeed, directed sets are generalizations of sequences.

Example 4 In a flat PPO $(A \cup \{\bot\}, \preceq, \bot)$, the directed sets are exactly: the singletons $\{x\}$ with $x \in A \cup \{\bot\}$, and the pairs of elements of the form $\{a, \bot\}$ with $a \in A$.

Definition 3 *A* Complete Partial Order *(CPO) is a PPO $(S, \preceq, \bot)$ with the additional property that any directed set $T \subseteq S$ has a* least upper bound*, denoted by $lub\, T$.*

Least upper bounds of directed sets are generalizations of limits of sequences.

Example 5 Consider the PPO $(\mathbb{N} \cup \{\infty\}, \leq, 0)$ with the order $\leq$ on natural numbers extended such that $\infty \leq \infty$ and $n \leq \infty$ for all $n \in \mathbb{N}$. Then, $(\mathbb{N} \cup \{\infty\}, \leq, 0)$ is a CPO. Indeed, in this totally ordered set all subsets $T \subseteq \mathbb{N} \cup \{\infty\}$ are directed, and $lub\, T$ is either:

- the maximum of T, if it exists
- ∞, if the maximum of T does not exist.

In Coq the set $\mathbb{N} \cup \{\infty\}$ shall be encoded as the type `conat` seen earlier in this section.

Example 6 In a PPO $(A \cup \{\bot\}, \preceq, \bot)$, the least upper bound of a singleton $\{x\}$ is x and the least upper bound of a pair $\{\bot, a\}$ is a. Those are the only directed sets in this PPO; hence, $(A \cup \{\bot\}, \preceq, \bot)$ is a CPO.

Informally, the notion of compactness below captures what it means for an element to be finite.

Definition 4 *In a CPO $(S, \preceq, \bot)$, an element $s^\circ \in S$ is* compact *whenever for all directed sets $T \subseteq S$, if $s^\circ \preceq lub\, T$ then there exists $t \in T$ such that $s^\circ \preceq t$.*

Example 7 In the CPO $(\mathbb{N} \cup \{\infty\}, \leq, 0)$ the compact elements are exactly the (finite) natural numbers.

Example 8 In the CPO $(A \cup \{\bot\}, \preceq, \bot)$ all the elements are compact.

Some CPOs are, in the following sense, completely determined by their compact elements:

Definition 5 *A CPO $(S, \preceq, \bot)$ having the set $S^\circ \subseteq S$ of compacts is* algebraic *if for all $s \in S$, the set $\{s^\circ \in S^\circ \mid s^\circ \preceq s\}$ is directed, and $s = lub\, \{s^\circ \in S^\circ \mid s^\circ \preceq s\}$.*

Example 9 The CPO $(\mathbb{N} \cup \{\infty\}, \leq, 0)$ is algebraic. Indeed, for all $n \in \mathbb{N} \cup \{\infty\}$, the set of compacts (natural numbers) $\{m \in \mathbb{N} \mid m \leq n\}$ is directed (as is any nonempty subset of $\mathbb{N} \cup \{\infty\}$). Moreover,

- if $n = \infty$, then the set $\{m \in \mathbb{N} \mid m \leq n\}$ coincides with $\mathbb{N}$, and $lub\, \mathbb{N} = \infty$;
- if $n \in \mathbb{N}$, then $lub\, \{m \in \mathbb{N} \mid m \leq n\} = n$.

Example 10 The flat CPO $(A \cup \{\bot\}, \preceq, \bot)$ is algebraic. Indeed:

- for all $a \in A$, the set $\{\bot, a\}$ of compacts in the $\preceq$ relation with a is directed, and $a = lub\, \{\bot, a\}$
- the set $\{\bot\}$ of compacts in the $\preceq$ relation with $\bot$ is directed and $lub\, \{\bot\} = \bot$.

We shall use the following notion, which relates a PPO to the compact elements of an algebraic CPO:

Definition 6 *Consider a PPO* $(S^\circ, \preceq^\circ, \bot^\circ)$ *and an algebraic CPO* $(T, \preceq, \bot)$ *whose set of compacts is* T°. *We say that* $(T, \preceq, \bot)$ *is an* embedding *of* $(S^\circ, \preceq^\circ, \bot^\circ)$ *if there exists an injection* $\iota : S^\circ \to T$ *such that:*

- $\iota \bot^\circ = \bot$*;*
- ι *is monotonic;*
- ι *maps* S° *to* T°*, written* $\iota S^\circ = T^\circ$.

The embedding, including the injection involved in it, is denoted by $\iota : (S^\circ, \preceq^\circ, \bot^\circ) \to (T, \preceq, \bot)$.

Example 11 The embedding $\kappa : (\mathbb{N}, \leq, 0) \to (\mathbb{N} \cup \{\infty\}, \leq, 0)$ is induced by the canonical inclusion κ of $\mathbb{N}$ into $\mathbb{N} \cup \{\infty\}$. The embedding of $(A \cup \{\bot\}, \preceq, \bot)$ into itself is also induced by the canonical inclusion.

Remark: not all embeddings are induced by canonical inclusions. In Coq, in general, they are not. This is because Coq is based on type theory, hence, one cannot just add a new element to a type; to do this one must create a new type and wrap the old type in a constructor (which, in Coq, is always an injection). An example of this is the representation of $\mathbb{N} \cup \{\infty\}$ as the type `conat` with a constructor `finite : nat -> conat`. With the appropriate order and bottom element, `conat` is an embedding of `nat` induced by the constructor `finite`. The only situations when canonical inclusion induces an embedding in Coq is when it coincides with the identity function, like in the embedding of $(A \cup \{\bot\}, \preceq, \bot)$ into itself.

Definition 7 *Assume an embedding* $\iota : (S^\circ, \preceq^\circ, \bot^\circ) \to (T, \preceq, \bot)$. *We denote by* $\iota^{-1} : T \to S^\circ$ *the (unique) function such that* $\iota^{-1}(\iota\, s^\circ) = s^\circ$ *for all* $s^\circ \in S^\circ$*, and* $\iota^{-1}\, t = \bot^\circ$ *for* $t \in T \setminus (\iota S^\circ)$.

Hence ι^{-1} is the inverse of ι on the compacts ιS° of T, and elsewhere it is given the (arbitrary) value $\bot$.

The next theorem is our main ingredient for defining and reasoning about partial recursive functions. It uses the following notion of continuity:

Definition 8 *Given two CPOs* $(T, \preceq, \bot)$ *and* $(T', \preceq', \bot')$*, a function* $f : T \to T'$ *is* continuous *if for any directed set* $S \subseteq T$*, its image* $(f\ S) \subseteq T'$ *is directed, and* $f\ (lub\ S) = lub(f\ S)$.

Theorem 1 *Assume two embeddings* $\iota_1 : (S_1^\circ, \preceq_1^\circ, \bot_1^\circ) \to (T_1, \preceq_1, \bot_1)$ *and* $\iota_2 : (S_2^\circ, \preceq_2^\circ, \bot_2^\circ) \to (T_2, \preceq_2, \bot_2)$ *and a monotonic function* $f^\circ : S_1^\circ \to S_2^\circ$*. Then there exists a unique continuous function* $f : T_1 \to T_2$ *such that* $f = \iota_2 \circ f^\circ \circ \iota_1^{-1}$ *— where* $\circ$ *is the standard notation for function composition.*

If the embeddings in Theorem 1 are canonical inclusions we have a simpler version of the above result:

Corollary 1 *Assume two embeddings* $\iota_1 : (S_1^\circ, \preceq_1^\circ, \bot_1^\circ) \to (T_1, \preceq_1, \bot_1)$ *and* $\iota_2 : (S_2^\circ, \preceq_2^\circ, \bot_2^\circ) \to (T_2, \preceq_2, \bot_2)$ *where* ι_1, ι_2 *are canonical inclusions. Then, for any any monotonic function* $f^\circ : S_1^\circ \to S_2^\circ$ *there exists a unique continuous function* $f : T_1 \to T_2$ *such that for all* $s^\circ \in S_1^\circ$*,* $f\ s^\circ = f^\circ s^\circ$.

3.3 Application to Partial Recursive Functions

We use the existence part of Theorem 1 in order to define partial recursive functions and the uniqueness part in order to prove that the defined functions are least fixpoints of their respective fixpoint equations.

The method has been formalized in Coq; we sketch it below in mathematical notation. Assume that we want to define a partial function $f : A \to (B \cup \{\bot\})$. We have have at our disposal the functional $F : (A \to (B \cup \{\bot\})) \to A \to (B \cup \{\bot\})$. The following assumptions on F are required:

- monotonicity: for all $f, f' : A \to (B \cup \{\bot\})$, if *for all* $a \in A$, $f\ a \preceq f'\ a$ then *for all* $a \in A$, $F\ f\ a \preceq F\ f'\ a$, where $\preceq$ is the flat order on $(B \cup \{\bot\})$.

- preservation of continuity: assume an arbitary function $g : (\mathbb{N} \cup \{\infty\}) \to A \to (B \cup \{\bot\})$. If, for each $a \in A$, the function $\lambda n \to g\ n\ a$ is continuous (as a function between $(\mathbb{N} \cup \{\infty\})$ and $(B \cup \{\bot\})$ organized as CPOs) then, for each $a' \in A$ the function $\lambda n \to F\ (g\ n)\ a'$ is continuous as well.

The method proceeds as a series of steps, grounded in the results from the previous subsection:

1. A function $f^\circ : \mathbb{N} \to A \to (B \cup \bot)$ is recursively defined by the equations: for all $a \in A$ and $m \in \mathbb{N}$, $f^\circ\ 0\ a = \bot$ and $f^\circ\ (m+1)\ a = F\ (f^\circ\ m)\ a$; intuitively, for all $m \in \mathbb{N}$, $(f^\circ\ m)$ constitute approximations of the function that we want to define, constrained by the finite amount of fuel $m \in \mathbb{N}$;
2. the *monotonicity* requirement on F ensures that, for all $a \in A$, the function $f^\circ_a = \lambda n \to (f^\circ\ n\ a)$ is monotonic as a function between $\mathbb{N}$ and $(B \cup \{\bot\})$ organized as PPOs;
3. the *existence* result of Theorem 1 ensures that, for all $a \in A$ that there exists a continuous function $f_a : (\mathbb{N} \cup \{\infty\}) \to (B \cup \{\bot\})$, satisfying $f_a\ m = f^\circ\ m\ a$ for all $m \in \mathbb{N}$; here, we have used the fact that, in the embeddings involved in our application of Theorem 1, the injections are the canonical inclusions (cf. Example 11), hence, one can apply the simpler version of the theorem — Corollary 1;
4. using the *uniqueness* result of the corollary, for all $a \in A$, any continuous function $f'_a : (\mathbb{N} \cup \{\infty\}) \to (B \cup \{\bot\})$ satisfying $f'_a\ m = f^\circ\ (m+1)\ a$ for all $m \in \mathbb{N}$, also satisfies $f'_a = f_a \circ (\lambda n \to n+1)$; here we have used the fact that the function $\lambda\ n \to n+1 : (\mathbb{N} \cup \{\infty\}) \to (\mathbb{N} \cup \{\infty\})$ is continuous, and that the composition of continuous functions is a continuous function as well;
5. for all $a \in A$, let $f'_a : (\mathbb{N} \cup \{\infty\}) \to (B \cup \{\bot\})$ be defined by $f'_a = \lambda n \to F\ (\lambda (x : A) \to f_x\ n)\ a$. Using the *continuity preservation* requirement on F, the continuity of f'_a reduces to the continuity of $\lambda n \to (\lambda (x : A) \to f_x\ n)\ a = \lambda n \to f_a\ n = f_a$; since f_a is continuous, f'_a is continuous as well;
6. moreover, using the definitions of f° (first item in this list), of f_a (item 2), and of f'_a (item 5): for all $m \in \mathbb{N}$, $f'_a\ m = F\ (\lambda x \to f_x\ m)\ a = F(\lambda x \to f^\circ\ m\ x)\ a = F\ (f^\circ\ m)\ a = f^\circ\ (m+1)\ a$; hence, (cf. item 4), $f'_a = f_a \circ (\lambda n \to n+1)$. This implies that for all $n \in (\mathbb{N} \cup \{\infty\})$, $f_a\ (n+1) = F\ (\lambda x \to f_x\ n)$;
7. let now $f : A \to (B \cup \{\bot\})$ defined, for all $a \in A$, by $f\ a = f_a\ \infty$. Then, using item 6 and $\infty = \infty + 1$: for all $a \in A$, $f\ a = f_a\ \infty = f_a\ (\infty + 1) = F\ (\lambda x \to f_x\ \infty)\ a = F\ (\lambda x \to f\ x)\ a = F\ f\ a$; that is, we have obtained the fixpoint equation $f = F\ f$. What remains to be proved is that f is its least solution;
8. for this, we inductively define a sequence of functions in $A \to (B \cup \{\bot\})$ by $F^0 = \lambda x \to \bot$ and, for all $m \in \mathbb{N}$, $F^{m+1} = F(F^m)$. Using the definition of f we prove the equality $f = lub\{F^n \mid n \in \mathbb{N}\}$, where the least upper bound is taken in the CPO of functions $A \to (B \cup \{\bot\})$ ordered pointwise. Finally, we use a result that says that if F is monotonic on a CPO (it is, in our case, by the *monotonicity* assumption) and $lub\{F^n \mid n \in \mathbb{N}\}$ is a fixpoint of F (it is, in our case, since $lub\{F^n \mid n \in \mathbb{N}\} = f$ and $f = F\ f$) then $lub\{F^n \mid n \in \mathbb{N}\}$ is the least fixpoint of F; i.e., f is the least fixpoint of F.

A comparison between the proposed approach for defining partial recursive functions and the standard one based on Kleene's fixpoint theorem is discussed in Section 6 dedicated to related works.

3.4 Instantiation to While Loops

The results from the previous subsection are now instantiated to `while` loops.

Recall from subsection 3.1 the failed attempt at defining `while` loops in Coq, and notice their type:

```
Fixpoint while{T:Type}(cond:reader T bool)(body:program T unit):program T unit:=
  do c <- cond; if c then body ;; while cond body else reader_to_program (ret tt)
```

In order to instantiate the method described in the previous subsection to `while` loops we first need to change their type to `A->option B` for appropriate `A`, `B`. Remembering that `program T unit` is defined as `T->option(unit*T)`, once the implicit parameter `T` is chosen, the type of `while` becomes

```
(reader T bool) -> (program T unit) -> T -> option(unit*T)
```

In order to obtain a type of the form `A->option B` we *uncurry* the above type to

```
((reader T bool)*(program T unit)*T) -> option(unit*T)
```

where * builds products between types. Next, we define a function `while':A->option B` where `A=(reader T bool)*(program T unit)*T` and `B=unit*T`. For this we first write the *functional* for the `while'` function as follows

```
Definition While'{T:Type}(W:((reader T bool)*(program T unit)*T)->option unit*T)
                         (p:(reader T bool)*(program T unit)*T): option unit*T :=
let (cond,body,s)  := decompose p in
(
 do c <- cond;
 if c then
 body;;(fun (s':T)=>W (cond, body, s')) (*after ;; a function on T is expected*)
 else reader_to_program (ret tt)
) s
```

(Notice how the parameter `p` was decomposed into three components.) After proving that `While'` is monotonic and preserves continuity, we obtain using the method in Subsection 3.3 the function `while': (reader T bool)*(program T unit)*T -> option unit*T` as the least fixpoint of `While'`.

What remains to be done is to *curry* the type `(reader T bool)*(program T unit)*T->option unit*T` to the expected type of the `while` function. When this is done, we obtain `while{T:Type}: (reader T bool)->(program T unit)->T->option unit*T` as the least fixpoint of the functional

```
Definition While{T:Type}(W:(reader T bool)->(program T unit)->T->option unit*T)
     (cond :(reader T bool))(body : (program T unit))(s :T)) : option unit*T :=
(do c <- cond; if c then body;;(W cond body) else reader_to_program (ret tt)) s
```

which concludes our construction of `while` loops in Coq. The next step is to provide users with means to reason about programs that contain such loops. This is the object of the next two sections. They shall be using the two following facts, which are consequences of `while` being the least fixpoint of its functional:

- an unfolding lemma, which is just another form of the fixpoint equation:

  ```
  Lemma while_unfold{T:Type}: forall(c: reader T bool)(b: program T unit),
  while c b =
        (do c' <- c; if c' then b;;while c b else reader_to_program (ret tt))
  ```

- a lemma stating that the `while` loop evaluates to `Some x` in a state if and only if there exists a fuel-constrained version of the loop that also evaluates to the same `Some x` in the same state:

  ```
  Lemma while_iff_while_fuel{T:Type}:
  forall (c:reader T bool)(b:program T unit)(s:T)(x:unit*T),
  while c b s = Some x <-> exists (fuel:nat), while_fuel fuel c b s = Some x.
  ```

  Since evaluation to `Some x` models termination, the lemma can also be read as"if a loop terminates, then it terminates in finitely many steps" - where the number of steps is upper-bounded by `fuel`.

4 Partial Correctness

In this section we define a monadic Hoare logic for partial correctness [7, 14]. Roughly speaking, partial correctness expresses the fact that a program returns the right answer whenever it terminates. In this paper, a program is a monadic computation. Remembering that `Prop` is the Coq type for logical statements, one writes the Hoare triple `{{P}} m {{Q}}` for the proposition `hoare_triple P m Q` defined in Coq by

```
Definition hoare_triple
  {T A : Type} (P : T -> Prop) (m : program T A) (Q : A -> T -> Prop) : Prop :=
forall s s' a, P s -> m s = Some (a, s') -> Q a s'.
```

That is, if the program `m: (program T A)` is in a state `s: T` such that the pre-condition `P: T -> Prop` holds for `s`, if the program terminates (encoded by the fact that the program returns `Some (a, s')`) then the pair `(a, s')` satisfies the postcondition `Q: A -> T -> Prop`.

There are Hoare triples for all monadic instructions, but the triple of interest is this paper is the one for the `while` loops. It states that: if the `body` of the loop preserves an invariant `I` as long as the condition `cond` of the loop is true, then the loop preserves the invariant whenever it terminates.

```
Lemma while_triple
  {T: Type}(cond: reader T bool)(body: program T unit)(I: T -> Prop):
{{ fun s => I s /\ cond s = true }} body {{ fun _ s' => I s' }} ->
{{ I }} while cond  body {{ fun _ s' => cond s' = false /\ I s' }}.
```

In order to prove `while_triple` we first prove a triple for fuel-constrained loops by induction on `fuel`:

```
Lemma while_fuel_triple
 {T:Type}(fuel:nat)(cond: reader T bool)(body: program T unit)(I: T -> Prop):
{{ fun s => I s /\ cond s = true }}body{{ fun _ s' => I s' }} ->
{{ I }} while_fuel fuel cond body  {{ fun _ s' => cond s' = false /\ I s' }}.
```

The lemma `while_fuel_triple` is used in the proof of `while_triple`, together with the lemmas `while_iff_while_fuel` and `while_unfold` shown at the end of the previous subsection.

Then, `while_triple` is used to prove the *weakest precondition* triple for our running example:

```
Lemma length_wp (addr: nat)(P: nat -> State -> Prop):
{{ fun s => forall len,
     Length s addr len ->
     P len {| reg1 := 0; reg2 := len; memory := s.(memory) |} }}
length addr
{{ P }}.
```

where `Length` is an inductively defined relation that says that in a state `s`, the list starting at address `addr` has length `len`. Having this predicate gives us an abstract manner of defining a linked list's length:

```
Inductive Length (s:State): nat -> nat -> Prop :=
| length_nil : forall addr, addr = 0 -> Length s addr 0
| length_cons :
  forall addr len,
  addr <> 0 -> Length s (s.(memory) addr) len -> Length s addr (S len).
```

In accordance to the laws of weakest precondition, `length_wp` says that the postcondition `P` must hold in the precondition when applied to the expected return value upon termination (the length `len`) and to the expected state upon termination `{|reg1 := 0; reg2 := len; memory := s.(memory)|}`.

As usual with Hoare logic, the crux of the proof is to find the right loop invariant to be fed to the lemma `while_triple`. Here, it is a generalization of the precondition in `length_wp`:

```
fun _ s => forall len,
  Length s s.(reg1) len ->
  P (len+s.(reg2)) {| reg1:= 0; reg2:= len + s.(reg2) ; memory:= s.(memory) |}).
```

With this choice of invariant the proof of `length_wp` is just a matter of unfolding definitions.

The interest of having proved a weakest precondition lies in its generality. As immediate corollaries of `length_wp` we obtain a first lemma that states that whenever `length addr` terminates, the register `reg2` contains the length of the linked list starting at address `addr`:

```
Lemma length_correct1 (s0: State)(addr: nat) :
{{ fun s => s = s0 }} length addr {{ fun _ s' => Length s0 addr s'.(reg2) }}.
```

And another lemma stating that: if the linked list starting at address `addr` has length `len`, then this is the value that will be returned by `length addr` whenever it terminates.

```
Lemma length_correct2 (len: nat)(addr: nat) :
{{ fun s => Length s addr len }} length addr {{ fun n _ => n = len }}.
```

5 Termination

In our approach, termination is modeled by evaluation to `Some` value. In order to succesfully prove termination we need to specify quite precisely the value to which a program evaluates. The key for the termination of the `length` program is the termination of its `while` loop, which is expressed as follows:

```
Lemma while_terminates:
forall len s addr, Length s addr len ->
forall n,
while do curr <- read_reg1; ret  (curr != 0)
(incr_reg2;; do curr <- read_reg1; do next <- read_addr curr; write_reg1 next)
{| reg1 := addr; reg2 := n; memory := s.(memory) |} =
Some (tt, {| reg1 := 0; reg2 := n + len; memory := s.(memory) |}).
```

That is, when called in a state where the first register `reg1` points to the beginning of the list and the second register `reg2` is initialized with some value `n`, the `while` loop ends in a state where the first register is `null` and the second register contains `n` plus the length of the list. The `memory` field of the state remains unchanged because the loop does not write in it. The return value `tt` is the unique inhabitant of `unit` and encodes the fact that `while` loops do not actually return anything relevant.

Lemma `while_terminates` is proved by induction on `n` and uses the lemma `while_unfold` to unfold the loop in the inductive step. It is then used to prove the termination of the `length` program:

```
Lemma length_terminates:
forall  s len addr,
Length s addr len ->
length addr s =
 Some (len,{|reg1 := 0; reg2 := len; memory := s.(memory)|}).
```

This says that the function call `length addr s` terminates with value `len` whenever, according to the inductive relation `Length`, the state `s` has a `memory` field where there is a well-formed linked list of length `len` starting at address `addr`. If the list were not well-formed, i.e. its links would form a loop, then the function would evaluate to `None`, which, in our method denotes non-termination.

6 Related Work

In domain theory [15] partial recursive functions are typically defined as least fixpoints of their functionals using *Kleene's fixpoint theorem*, which states that a functional has a least fixpoint whenever it is *continuous*, and that the least fixpoint is obtained by infinitely many iterations of the functional starting from a function defined nowhere. This theorem is very elegant and easy to prove. Our own version of a fixpoint theorem (perhaps less elegant than Kleene's theorem, and definitely harder to prove) has the same conclusion but requires that the functional be monotonic and *continuity-preserving*, which roughly means that, given a certain continuous function, the functional produces a continuous function. From our own experience and that of other authors (e.g., [3]) it appears that *using* Kleene's theorem, which requires proving continuity, is difficult in practice, even in the simplest cases. For example, it took us hundreds of lines of Coq code just to prove the continuity of the successor function on natural numbers extended with infinity. By contrast, using our version of the theorem requires a proof of preservation of continuity, which appear to be more manageable - for while-loops in a shallow embedding of an imperative programming language in Coq the proof of continuity-preservation is remarkably simple: ten lines of Coq code. One possible reason for why continuity-preservation is easier to prove for functionals than continuity is that that the latter refers to the *higher-order* functional itself, whereas the former concerns the argument of the functional, which is a simpler function, one order below the order of the functional.

Other authors have explored partial recursive functions in Coq. In [5] a partial recursive function's codomain is a *thunk* - a parameterized coinductive type that "promises" an answer as a value of its parameter, but may postpone this answer forever, yielding nontermination. However, the functions being defined now become *corecursive* functions, which are restricted in the Coq proof assistant. As a result, only *tail-recursive* functions can be defined with this approach. This was also noted in [6, Chapter 7.3].

The same author [6, Chapter 7.2] proposes an alternative for the codomain of a partial function: a *computation* is a type that associates to a natural-number approximation level an approximation of the intended function. Like us the author uses the Coq `option` type, where `None` stands for nontermination and `Some` for termination with a return value; and a monad of computations is designed so that computations can encode imperative features. However, [6, Chapter 7.2] requires that the functional of the function being defined be continuous, for a definition of continuity equivalent to that employed in domain theory; hence this approach is subject to the general difficulty of proving continuity of functionals.

Regarding the embedding of imperative languages in proof assistants for the purpose of program verification, an alternative to the shallow embedding that we here use is *deep* embedding, which consist in defining the syntax, operational semantics [9, 10], and program logic for the guest language in the logic of the host. Among the many projects we here cite the Iris project [1], a rich environment built around a concurrent programming language and the corresponding program logic - concurrent separation logic.

7 Conclusion and Future Work

Recursive functions in Coq need to terminate for the underlying logic to be sound. Coq typically ensures termination via an automatically-checkable structural decreasing of terms to which recursive calls apply. In more complex cases Coq can be helped by the user with termination proofs. The termination proof becomes a part of a function definition; a function is not defined until the termination proof is completed.

For various reasons falling under the general notion of separation of concerns it is desirable to separate function definitions from termination proofs. It is also useful to have functions that do not terminate on some inputs. This paper proposes a new approach that achieves these desirable features. Non-termination is simulated as evaluation to a special value interpreted as "undefined". Under mild conditions on the function's body encoded as a higher-order functional, a possibly non-terminating function is defined and proved to be the least fixpoint of its functional according to a certain definition order.

We instantiate the general approach to while-loops in an imperative language shallowly embedded in Coq. The shallow embedding is based on a combination of monads. The least-fixpoint property of the resulting while loops is a key property enabling termination and partial-correctness proofs on imperative programs containing them. The practicality of the approach is illustrated by proving partial correctness and termination properties on a program computing the length of linked lists. Partial correctness is expressed in Hoare logic and is proved in the standard manner, by having users provide a strong-enough invariant; and termination is proved by having users provide an upper bound for the number of iterations.

Future Work A promising line of future work is to extend our approach to defining partial *co*recursive function in Coq. The idea is that codomains of such functions would be encodings of coinductive types organized as algebraic CPOs, generalizing the option types that we here used for recursive functions. Initial experiments with defining some difficult corecursive functions that go beyond Coq's builtin corecursion mechanisms (a filter function on streams, a mirror function on Rose trees) are promising.

The function-definition mechanism presented in this paper critically depends on functionals preserving continuity. A deeper understanding of the relationship between continuity-preservation and continuity, and of the perimeter where the continuity-preservation property holds, is also left for future work.

A more practical future work direction is to apply the instance of our approach to imperative-program definition and verification. Our intention is to build on our experience with proving low-level programs manipulating linked lists [8]. An interesting logic to consider in this setting is separation logic [12], perhaps by drawing inspiration from the shallow embedding of separation logic in Coq presented in [13].

References

[1] *The Iris Project*. `https://iris-project.org/`.

[2] Roberto M. Amadio & Pierre-Louis Curien (1998): *Domains and lambda-calculi*. *Cambridge tracts in theoretical computer science* 46, Cambridge University Press, doi:10.1017/CBO9780511983504. Also available as an Inria research report: `https://hal.inria.fr/inria-00070008`.

[3] Y. Bertot & V. Komendantsky (2008): *Fixed point semantics and partial recursion in Coq*. In: *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pp. 89–96, doi:10.1145/1389449.1389461.

[4] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-662-07964-5.

[5] Venanzio Capretta (2005): *General recursion via coinductive types*. *Log. Methods Comput. Sci.* 1(2), pp. 1–18, doi:10.2168/LMCS-1(2:1)2005.

[6] Adam Chlipala (2013): *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. Available at `http://mitpress.mit.edu/books/certified-programming-dependent-types`.

[7] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *Commun. ACM* 12(10), pp. 576–580, doi:10.1145/363235.363259.

[8] Narjes Jomaa, Paolo Torrini, David Nowak, Gilles Grimaud & Samuel Hym (2018): *Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base*. In: *18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018)*, Oxford, United Kingdom, doi:10.14279/tuj.eceasst.76.1080. Available at `https://hal.science/hal-01816830`.

[9] Robbert Krebbers, Xavier Leroy & Freek Wiedijk (2014): *Formal C Semantics: CompCert and the C Standard*. In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, *Lecture Notes in Computer Science* 8558, Springer, pp. 543–548, doi:10.1007/978-3-319-08970-6_36.

[10] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Commun. ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[11] Eugenio Moggi (1989): *Computational Lambda-Calculus and Monads*. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, IEEE Computer Society, pp. 14–23, doi:10.1109/LICS.1989.39155.

[12] John C. Reynolds (2005): *An Overview of Separation Logic*. In Bertrand Meyer & Jim Woodcock, editors: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, *Lecture Notes in Computer Science* 4171, Springer, pp. 460–469, doi:10.1007/978-3-540-69149-5_49.

[13] Ilya Sergey: *Programs and Proofs: Mechanizing Mathematics with Dependent Types*, doi:10.5281/zenodo.4996238. Lecture notes with exercises.

[14] Wouter Swierstra (2009): *A Hoare Logic for the State Monad*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, *Lecture Notes in Computer Science* 5674, Springer, pp. 440–451, doi:10.1007/978-3-642-03359-9_30.

[15] Glynn Winskel (1993): *The formal semantics of programming languages - an introduction*. Foundation of computing series, MIT Press, doi:10.7551/mitpress/3054.001.0001.

Privacy-preserving Linear Computations in Spiking Neural P Systems

Mihail-Iulian Plesa
University of Bucharest
Bucharest, Romania
Department of Computer Science
mihail-iulian.plesa@s.unibuc.ro

Marian Gheorghe
University of Bradford
Bradford, UK
School of Electrical Engineering and Computer Science
M.Gheorghe@bradford.ac.uk

Florentin Ipate
University of Bucharest
Bucharest, Romania
Department of Computer Science
florentin.ipate@unibuc.ro

Spiking Neural P systems are a class of membrane computing models inspired directly by biological neurons. Besides the theoretical progress made in this new computational model, there are also numerous applications of P systems in fields like formal verification, artificial intelligence, or cryptography. Motivated by all the use cases of SN P systems, in this paper, we present a new privacy-preserving protocol that enables a client to compute a linear function using an SN P system hosted on a remote server. Our protocol allows the client to use the server to evaluate functions of the form t_1k+t_2 without revealing t_1, t_2 or k and without the server knowing the result. We also present an SN P system to implement any linear function over natural numbers and some security considerations of our protocol in the honest-but-curious security model.

1 Introduction

Membrane computing (or P systems) is a new model of computation inspired by how membranes work and interact in living cells [17]. There are several variants of the model e.g. neural P systems, cell P systems, tissue P systems, etc., [29, 15, 12]. P systems have generated new perspectives on the P vs NP problem, being used to efficiently solve hard problems [27, 3, 7, 28]. There are also multiple applications of P systems in various fields like formal verification, artificial intelligence, or cryptography [30].

In this work we used a special type of P systems called Spiking Neural P systems (SN P systems for short) [12]. SN P systems are inspired by biological neurons. There are also numerous variants of SN P systems: SN P systems with astrocytes, SN P systems with communication on request, SN P systems with polarization, SN P systems with colored spikes, etc., [16, 14, 25, 23].

Although there are many theoretical aspects and simulations in the literature, to gain the maximum efficiency of these systems, they must be implemented on dedicated hardware [1]. If these systems are implemented at a large scale, they will have to be accessed remotely in the cloud. This raises privacy concerns about data uploaded to the server that hosts the P system. This paper approaches the problem of confidentiality in SN P systems by describing a protocol that allows a client to perform a simple linear computation using an SN P system that is served remotely without revealing private information.

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 110–119, doi:10.4204/EPTCS.389.9

1.1 Related work

Besides the theoretical work, there are also many applications of P systems. In [22] the authors propose a new key agreement protocol based on SN P systems. In [8, 10, 11] the authors describe how to implement the RSA algorithm in the framework of membrane computing. One ingenious way of applying P systems is shown in [24] which presents an algorithm to break the RSA encryption. There are also applications in artificial intelligence. In [2] the authors present a survey of the learning aspects in SN P systems. Clustering algorithms have also been developed in the framework of membrane computing [21, 20, 19]. Image processing is another common application of P systems [4, 26, 5].

1.2 Our contribution

In this paper, we present a protocol that allows a client to perform a linear computation using an SN P system hosted on a server without revealing any private data. The SN P system computes functions of the form t_1k+t_2 over natural numbers. The client must retrieve from the server the result of the computation without the server knowing t_1, t_2 or k. Also, the server must not learn the value t_1k+t_2. To enable privacy-preserving computations on the server side, we use the ElGamal cryptosystem and its homomorphic properties [6]. We also provide an SN P system that computes any linear function over natural numbers and some security considerations of our protocol. The paper is organized as follows: in Section 2 we present the background on the SN P system and homomorphic encryption. In Section 3, we show an SN P system that computes linear functions over the natural numbers. In Section 4, we introduce our protocol and some security considerations. Section 5 is left for the conclusions and further directions.

2 Preliminaries

In this section, we briefly present the Spiking Neural P systems (SN P systems) and the cryptographic algorithm used in our protocol. We stress some useful properties of the encryption scheme.

2.1 Spiking Neural P systems

A Spiking Neural P system (SN P system) of degree $m \geq 1$ is defined as the following construct:

Definition 2.1. $\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, i_0)$ where:

- $O = \{a\}$ is the alphabet. The symbol a denotes a spike.
- $\sigma_i,\ 1 \leq i \leq m$ represents a neuron. Each neuron is characterized by the initial number of spikes denoted by $n_i \geq 0$ and the finite set of rules denoted by R_i: $\sigma_i = (n_i, R_i)$.
- Each rule can be of the following two forms:
  1. $E/a^r \rightarrow a;t$ where E is a regular expression over the alphabet O, $r \in \mathbf{N}^*$ represents the current number of spikes in the neuron and $t \geq 0$ is the refractory period. This type of rule is called a firing rule.
  2. $a^s \rightarrow \lambda$ where $s \geq 1$ is the current number of spikes in the neuron and λ is a special symbol that denotes an empty set of spikes. This type of rule is called a forgetting rule.
- $syn \subseteq \{1,2,\ldots,m\} \times \{1,2,\ldots,m\}$ is the set of synapses between neurons. No neuron can have a synapse to it i.e. $(i,i) \notin syn\ \forall\ 1 \leq i \leq m$.

- i_0 represents the output neuron.

A neuron can fire using the firing rule $E/a^r \to a;t$ only if it contains n spikes such that $a^n \in L(E)$ and $n \geq r$ where $L(E)$ is a language defined in the following way:

- $L(\lambda) = \{\lambda\}$
- $L(a) = \{a\}\ \forall a \in O$
- $L((E_1) \cup (E_2)) = L(E_1) \cup L(E_2)$
- $L((E_1)(E_2)) = L(E_1)L(E_2)$
- $L\left((E_1)^+\right) = L(E_1)^+$

for all regular expressions over the alphabet O.

After firing, r spikes are consumed. A firing rule that is applied when the neuron contains exactly r spikes i.e. $L(E) = \{a^r\}$, is simply denoted as $a^r \to a;t$.

At the neuron level, all rules are applied sequentially, but the system as a whole evolves with maximum parallelism i.e. if a rule can be applied in a neuron then that rule will be applied.

At a certain point, a neuron can be firing, spiking, or closed. If a neuron applies the firing rule $E/a^r \to a;t$ at moment q then the neuron will send a spike to all the neurons to which it is connected by synapses at moment $q+t$. At times $q+1, q+2, \ldots q+t-1$ the neuron will be in the refractory period i.e. the neuron will not receive or send any spikes. When neuron σ_i is spiking, the spikes are replicated in such a way that each neuron σ_j with $(i, j) \in syn$ receives one spike although the number of spikes consumed by σ_i is exactly r.

When a forgetting rule $a^s \to \lambda$ is applied in a neuron, s spikes are removed from that neuron. A neuron can apply a forgetting rule only if the number of spikes is exactly s.

There are several ways in which we can record the output of an SN P system:

- The moments of time at which the output neuron i_0 sends a spike i.e. if the neuron i_0 releases spikes at the moments $q_1, q_2 \ldots$ then the output of Π is the sequence $q_1, q_2 \ldots$.
- The interval between the moments at which the output neuron i_0 sends a spike i.e. if the neuron i_0 releases spikes at the moments $q_1, q_2 \ldots$ then the output of Π is the sequence $q_2 - q_1, q_3 - q_2, \ldots$

An SN P system is constructed using the principle of minimal determinism i.e. at a certain moment in time, either a firing or a forgetting rule is applied without being able to choose which of the two types of rules is applied [12].

2.2 Homomorphic encryption

In this work, we use the ElGamal cryptosystem [6]. The security of the encryption scheme is based on the computational Diffie-Hellman assumption (CDH). Moreover, the scheme achieves semantic security based on the decisional Diffie-Hellman assumption (DDH) i.e. the scheme is randomized. Randomization implies that when encrypting the same message multiple times, each resulting ciphertext will be different. A consequence of this property is the fact that an attacker cannot distinguish two plaintexts by analyzing the corresponding ciphertext with non-negligible probability. The scheme works over a group G of order q with a generator g. We now proceed to the description of the cryptosystem:

- The key generation algorithm denoted by **KeyGen** generates a key pair i.e. a private key and the corresponding public key. The algorithm takes the following steps:

1. Generate a random integer $x \in \{1,2,\ldots q-1\}$.
2. Compute $h := g^x$.
3. Output the public key h and the corresponding private key x.

- The encryption algorithm denoted by $\mathbf{Enc}_h^y$ encrypts a plaintext $m \in G$ using the public key h and a random number $y \in \{1,2,\ldots,q-1\}$. The algorithm performs the following steps:
  1. Computes $s := h^y$.
  2. Computes $c_1 := g^y$ and $c_2 := m \cdot s$.
  3. Outputs the ciphertext $c := (c_1, c_2)$.
- The decryption algorithm denoted by $\mathbf{Dec}_x$ takes as input a ciphertext $c = (c_1, c_2)$ and decrypts it under the private key x. The algorithm is composed of the following steps:
  1. Computes $s := c_1^x$.
  2. Computes s^{-1}, the inverse of s in the group G.
  3. Computes the plaintext $m := c_2 \cdot s^{-1}$.
  4. Outputs the plaintext m.

The proof of correctness is straightforward:

$$c_2 \cdot s^{-1} = c_2 \cdot c_1^{-x} = m \cdot h^y \cdot g^{-xy} = m \cdot g^{-xy} \cdot g^{xy} = m \tag{1}$$

The encryption of a message $m \in G$ can be summarized by the following two equations:

$$c_1 = g^y \tag{2}$$

$$c_2 = m \cdot h^y \tag{3}$$

The scheme is homomorphic with respect to multiplication. Let $c = (c_1, c_2)$ and $c' = (c_1', c_2')$ be the encryptions of two plaintexts m and m' under the same public key h i.e. $c = \mathbf{Enc}_h^y(m)$ and $c' = \mathbf{Enc}_h^y(m')$. We define the following two operations:

1. Let $c \odot c' = (c_1 \cdot c_1', c_2 \cdot c_2')$ be the multiplication of two ciphertexts. The result of this operation is another ciphertext that encrypts the sum between m and m':

$$c_1 \cdot c_1' = g^y \cdot g^{y'} = g^{y+y'} \tag{4}$$

$$c_2 \cdot c_2' = m \cdot s \cdot m' \cdot s' = m \cdot m' \cdot h^y \cdot h^{y'} = m \cdot m' \cdot h^{y+y'} \tag{5}$$

   From 4 and 5 we can see that $c \odot c' = (c_1 \cdot c_1', c_2 \cdot c_2')$ is a ciphertext that encrypts $m \cdot m'$ under the public key h.

2. Let $c \otimes k = (c_1, c_2 \cdot k)$ be the multiplication between a ciphertext and a constant k. The result of this operation is a ciphertext that encrypts the product between m and k as it can be seen from 6 and 7:

$$c_1 = g^y \tag{6}$$

$$c_2 \cdot k = m \cdot s \cdot k = (m \cdot k) \cdot s = (m \cdot k)\, h^y \tag{7}$$

3. When $c_1 = c_1'$ i.e. $y = y'$, we can also define the addition between two ciphertexts as follows: $c \oplus c' = (c_1, c_2 + c_2')$. The result of this operation is a ciphertext that encrypts the sum between m and m':

$$c_1 = g^y \tag{8}$$

$$c_2 + c_2' = m \cdot s + m' \cdot s = \left(m + m'\right) \cdot s = \left(m + m'\right) \cdot h^y \tag{9}$$

It is important to notice that when $y = y'$ the scheme is no longer semantically secure. Although an attacker who observes the two ciphertexts c and c' could not recover any of the plaintexts, it could determine additional information about them e.g. whether they are different. In many scenarios, this is not acceptable but in this work, we will use the $\oplus$ operation. All the messages encrypted with the same random y are not critical i.e. the impact of the lack of semantic security does not affect the security of the protocol in which the encryption scheme is used.

3 SN P system to compute linear functions

In this section, we describe an SN P system that computes functions of the form $t_1 k + t_2$ over natural numbers.

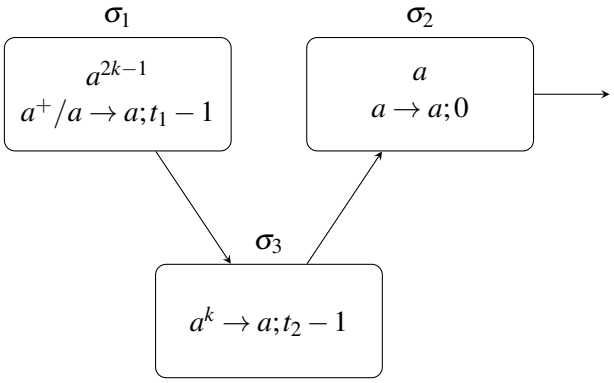


Figure 1: SN P system to compute linear functions

Let $\Pi_{Add}(t_1, t_2, k) = \{\{a\}, \sigma_1, \sigma_2, \sigma_3, syn_{add}, \sigma_2\}$ be the SN P system that computes the linear function $t_1 k + t_2$, $t_1, t_2 \in \mathbf{N}$ with the following components:

- The alphabet is made from a single symbol $\{a\}$ that denotes a spike.
- There are three neurons: σ_1, σ_2 and σ_3 with the following firing rules:
  1. For σ_1 the firing rule is $a^+/a \to a; t_1 - 1$.
  2. For σ_2 the firing rule is $a \to a; 0$.
  3. For σ_3 the firing rule is $a^k \to a; t_2 - 1$.
- The set of synapses syn_{add} is the set $\{(1,3),(3,2)\}$.
- The output neuron is σ_2.

Initially σ_1 has $2k-1$ spikes, σ_2 has 1 spike and σ_3 has no spikes. Since σ_2 has no refractory time and one spike, it will release it in the first step of the computation. During the first step, σ_1 will be firing. Since its refractory period is $t_1 - 1$ during the time steps $2, 3, \ldots t_1 - 1$ the neuron will be closed i.e. it

will not receive or send any spikes. In step t_1 the neuron will send one spike to σ_3 and fire again. Thus σ_1 fires every t_1 steps consuming one spike and sending one spike to σ_3. σ_3 will fire when it acumulates k spikes from neuron σ_1. Since σ_1 fires one spikes every t_1 steps, at time step, $k \cdot t_1$ σ_3 will receive the k^{th} spike. At the moment $t_1 \cdot k+1$, σ_3 will fire. The refractory period of this neuron is t_2-1 thus at the steps $t_1 \cdot k+2, t_1 \cdot k+3, \ldots, t_1 \cdot k+t_2-1$ it will be closed and it will release one spike to σ_2 at $t_1 \cdot k+t_2$. Since σ_3 has no refractory period, it will release the spike at the moment $t_1 \cdot k+t_2+1$. At this point, the number of spikes left in σ_1 is $k-1$ because it already sent k spikes to neuron σ_3 and the initial number of spikes was $2k-1$. The neuron will continue to send spikes to σ_3 at the appropriate time steps until the spikes run out. The neuron σ_3 will never fire again since it can no longer accumulate k spikes. σ_2 will never fire again either since it will no longer receive the spike from σ_3. Thus, after σ_1 exhausts all the spikes, the computation will stop. There are two moments when the output neuron σ_2 fires: 1 and $t_1 \cdot k+t_2+1$. Thus, the result of the computation i.e. the difference between the time points at which the output neuron fires, is $t_1 \cdot k+t_2$. The $\Pi_{Add}(t_1,t_2,k)$ system is depicted in Figure 1.

4 Privacy-preserving computations in SN P systems

In this section, we describe our protocol which enables the running of an SN P system to compute linear functions in a privacy-preserving way. We also make some remarks regarding security.

4.1 The protocol

There are two actors in the protocol:

1. The Server: it can instantiate and run an SN P system of the form $\Pi_{Add}(t_1,t_2,k)$ for any integers t_1,t_2 and k.
2. The Client: it wants to evaluate the linear function $t_1 \cdot k+t_2$ using the system hosted by the Server without revealing any of inputs t_1,t_2 or k.

The protocol uses the holomorphic properties of the ElGamal cryptosystem to allow the client to use the server without revealing the inputs of the SN P system. There are 7 steps:

1. The Client will use the **KeyGen** algorithm to generate a key pair: h and x.
2. The Client will use the encryption algorithm $\mathbf{Enc}_h^y$ to encrypt t_1,t_2 and k. We denote by $c^{t_1} = \left(c_1^{t_1},c_2^{t_1}\right), c^{t_2} = \left(c_1^{t_2},c_2^{t_2}\right)$ and $c^k = \left(c_1^{t_k},c_2^k\right)$ the encryptions of t_1,t_2 and k:
   - $c^{t_1} = \left(c_1^{t_1},c_2^{t_1}\right) = \mathbf{Enc}_h^{y_1}(t_1)$
   - $c^k = \left(c_1^k,c_2^k\right) = \mathbf{Enc}_h^{y_2}(k)$
   - $c^{t_2} = \left(c_1^{t_2},c_2^{t_2}\right) = \mathbf{Enc}_h^{y_1+y_2}(t_2)$
3. The Client will store locally $c_1^{t_1}, c_1^{t_2}$ and c_1^k and send to the server $c_2^{t_1}, c_2^{t_2}$ and c_2^k.
4. The Server will instantiate and run an SN P system of the form $\Pi_{Add}\left(c_2^{t_1},c_2^{t_2},c_2^k\right)$.
5. After the computation stops, the Server will return to the client the result of the computation i.e. $c_2 = c_2^{t_1} \cdot c_2^k + c_2^{t_2}$.
6. The Client will compose a new ciphertext $c = (c_1,c_2)$ where $c_1 = c_1^{t_1} \cdot c_1^k$.
7. The Client will decrypt the ciphertext c using the algorithm $\mathbf{Dec}_x$ and retrived the result of the computation i.e. $t_1 \cdot k+t_2$.

We now prove that the ciphertext computed in step 6 of the protocol is a valid ElGamal ciphertext that correctly decrypts to the final result of the computation i.e. $t_1 \cdot k + t_2$. Let c' be the following ciphertext:

$$c' = \left(c_1, c_2^{t_1} \cdot c_2^k\right) \tag{10}$$

Since $c_1 = c_1^{t_1} \cdot c_1^k$ we can write c' as:

$$c' = \left(c_1^{t_1} \cdot c_1^k, c_2^{t_1} \cdot c_2^k\right) \tag{11}$$

The ciphertext c' represents the multiplication between the ciphertexts c^{t_1} and c^k:

$$c' = c^{t_1} \odot c^k \tag{12}$$

Since c and c' use the same randomess i.e. c_1, we can write c as the sum between the ciphertext c' and c^{t_2}:

$$c = c' \oplus c^{t_2} \tag{13}$$

In conclusion, we can express c as a composition of valid ElGamal ciphertexts which is also a valid ElGamal ciphertext:

$$c = \left(c^{t_1} \odot c^k\right) \oplus c^{t_2} \tag{14}$$

The ciphertext $\left(c^{t_1} \odot c^k\right)$ represents the encryption of $t_1 \cdot k$. When we add this ciphertext with c^{t_2} using the $\oplus$ operation, the resulting ciphertext will be the encryption of $t_1 \cdot k + t_2$ which is the result of the computation performed over plaintext data. Figure 2 depicts our protocol.

Since the SN P system works over natural numbers, we can use $G = \mathbf{Z}_q$ for a large prime q.

4.2 Security considerations

We analyze our protocol in the honest-but-curious security model [18]. In this model, we assume that the adversary is the Server. There are two properties of the adversary:

1. Curious: the Server will try to find information about the underlying plaintexts. In our case, these are the parameters of the SN P system: t_1, t_2, and k.
2. Honest: the Server will respect the protocol and it will complete every step. It will not modify in any way the messages received or sent to the Client.

The underlying encryption scheme i.e. the ElGamal cryptosystem is semantically secure given the DDH assumption as long as each ciphertext is generated using different randomness [9]. The security of the scheme can be illustrated using the following game. We suppose that the attacker runs in probabilistic polynomial time and has access to an encryption oracle that receives as input a plaintext and returns the corresponding ciphertext:

- The attacker chooses as many plaintexts as it wants and encrypts them using the oracle. For each ciphertext, the attacker knows the corresponding plaintext.
- The attacker chooses two plaintext m_0 and m_1 and send them to the oracle.
- The oracle will generate a random bit b and return the encryption of m_b.
- The attacker outputs the bit b.

The scheme is secure as long as the attacker cannot output the correct bit with non-negligible probability.

Given the fact that the Client encrypts each parameter of the SN P system using the ElGamal cryptosystem with different randomness, the Server cannot learn any information about them with non-negligible probability.

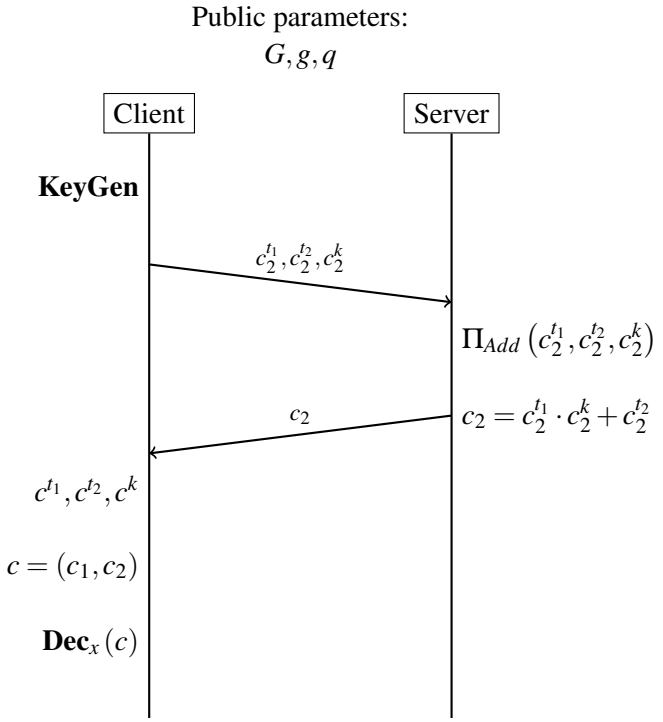


Figure 2: Privacy-preserving linear function computation using SN P systems

5 Conclusions and further directions of research

In this paper, we presented a protocol for performing privacy-preserving computation over SN P systems. There are two actors involved: the client and the server. The server hosts an SN P system that computes linear function over natural numbers i.e. t_1k+t_2. The client uses the protocol to retrieve the result of the computation without revealing t_1, t_2 or k and without the server knowing the result of the calculation. We presented an SN P system that computes any linear function over natural numbers and also some security considerations about our protocol which is based on the ElGamal cryptosystem.

There are several directions of research. The first one is to give formal proof of the security of the protocol. Although the protocol is secure at first sight, we must prove it by reducing the security of it to the security of the underlying cryptosystem. The second direction of research is to enable complex computation on SN P systems in a privacy-preserving way. The third direction is to use dedicated cryptosystems e.g. fully homomorphic encryption schemes which are created to enable privacy-preserving computations [13]. The challenge here is to map the operations performed over encrypted data to the operations performed by the SN P system. We should also consider using other privacy-enhancing technologies e.g. secure multi-party computation or differential privacy to enable private computations over SN P systems. Another direction of research is to implement the protocol and perform the appropriate benchmarking to study its efficiency and communication overhead in practice.

5.0.1 Acknowledgements

This research was supported by the European Regional Development Fund, Competitiveness Operational Program 2014-2020 through project IDBC (code SMIS 2014+: 121512).

References

[1] Alberto Arteta Albert, Ernesto Díaz-Flores, Luis Fernando de Mingo López & Nuria Gómez Blas (2021): *An in vivo proposal of cell computing inspired by membrane computing*. Processes 9(3), p. 511, doi:10.3390/pr9030511.

[2] Yunhui Chen, Ying Chen, Gexiang Zhang, Prithwineel Paul, Tianbao Wu, Xihai Zhang, Haina Rong & Xiaomin Ma (2021): *A Survey of Learning Spiking Neural P Systems and A Novel Instance*. International Journal of Unconventional Computing 16.

[3] Erzsébet Csuhaj-Varjú, Marian Gheorghe, Alberto Leporati, Miguel Ángel Martínez-del Amor, Linqiang Pan, Prithwineel Paul, Andrei Păun, Ignacio Pérez-Hurtado, Mario J Pérez-Jiménez, Bosheng Song et al. (2022): *Membrane computing concepts, theoretical developments and applications*. In: Handbook of Unconventional Computing: VOLUME 1: Theory, World Scientific, pp. 261–339, doi:10.1142/9789811235726_0008.

[4] Daniel Díaz-Pernil, Miguel A Gutiérrez-Naranjo & Hong Peng (2019): *Membrane computing and image processing: a short survey*. Journal of Membrane Computing 1, pp. 58–73, doi:10.1007/s41965-018-00002-x.

[5] Daniel Díaz-Pernil, Francisco Pena-Cantillana & Miguel A Gutiérrez-Naranjo (2013): *A parallel algorithm for skeletonizing images by using spiking neural P systems*. Neurocomputing 115, pp. 81–91, doi:10.1016/j.neucom.2012.12.032.

[6] Taher ElGamal (1985): *A public key cryptosystem and a signature scheme based on discrete logarithms*. IEEE transactions on information theory 31(4), pp. 469–472, doi:10.1109/TIT.1985.1057074.

[7] Songhai Fan, Yiyu Gong, Gexiang Zhang, Yun Xiao, Haina Rong, Prithwineel Paul, Xiaomin Ma, Han Huang & Marian Gheorghe (2021): *Implementation of Kernel P Systems in CUDA for Solving NP-hard Problems*. International Journal of Unconventional Computing 16.

[8] Ganbat Ganbaatar, Dugar Nyamdorj, Gordon Cichon & Tseren-Onolt Ishdorj (2021): *Implementation of RSA cryptographic algorithm using SN P systems based on HP/LP neurons*. Journal of Membrane Computing 3, pp. 22–34, doi:10.1007/s41965-021-00073-3.

[9] Oded Goldreich (2004): *Foundations of Cryptography, Volume 2*. Cambridge university press Cambridge, doi:10.1017/CBO9780511721656.

[10] Ping Guo & Wei Xu (2016): *A family P system of realizing RSA algorithm*. In: Bio-inspired Computing–Theories and Applications: 11th International Conference, BIC-TA 2016, Xi'an, China, October 28-30, 2016, Revised Selected Papers, Part I 11, Springer, pp. 155–167, doi:10.1007/978-981-10-3611-8_16.

[11] Ping Guo & Wei Xu (2017): *Implementation of RSA algorithm based on P system*. Journal of Computational and Theoretical Nanoscience 14(9), pp. 4227–4235, doi:10.1166/jctn.2017.6723.

[12] Mihai Ionescu, Gheorghe Păun & Takashi Yokomori (2006): *Spiking neural P systems*. Fundamenta informaticae 71(2-3), pp. 279–308.

[13] Paulo Martins, Leonel Sousa & Artur Mariano (2017): *A survey on fully homomorphic encryption: An engineering perspective*. ACM Computing Surveys (CSUR) 50(6), pp. 1–33, doi:10.1145/3124441.

[14] Linqiang Pan, Gheorghe Păun, Gexiang Zhang & Ferrante Neri (2017): *Spiking neural P systems with communication on request*. International journal of neural systems 27(08), p. 1750042, doi:10.1142/S0129065717500423.

[15] Linqiang Pan & Mario J Pérez-Jiménez (2010): *Computational complexity of tissue-like P systems*. Journal of Complexity 26(3), pp. 296–315, doi:10.1016/j.jco.2010.03.001.

[16] Linqiang Pan, Jun Wang & Hendrik Jan Hoogeboom (2012): *Spiking neural P systems with astrocytes*. Neural Computation 24(3), pp. 805–825, doi:10.1162/NECO_a_00238.

[17] Gheorghe Păun (2000): *Computing with membranes*. Journal of Computer and System Sciences 61(1), pp. 108–143, doi:10.1006/jcss.1999.1693.

[18] Andrew Paverd, Andrew Martin & Ian Brown (2014): *Modelling and automatically analysing privacy properties for honest-but-curious adversaries*. Tech. Rep.

[19] Hong Peng, Xiaohui Luo, Zhisheng Gao, Jun Wang, Zheng Pei et al. (2015): *A novel clustering algorithm inspired by membrane computing*. The Scientific World Journal 2015, doi:10.1155/2015/929471.

[20] Hong Peng, Jun Wang, Mario J Pérez-Jiménez & Agustín Riscos-Núñez (2015): *An unsupervised learning algorithm for membrane computing*. Information Sciences 304, pp. 80–91, doi:10.1016/j.ins.2015.01.019.

[21] Hong Peng, Jun Wang, Peng Shi, Agustín Riscos-Núñez & Mario J Pérez-Jiménez (2015): *An automatic clustering algorithm inspired by membrane computing*. Pattern Recognition Letters 68, pp. 34–40, doi:10.1016/j.patrec.2015.08.008.

[22] Mihail-Iulian Plesa, Marian Gheoghe, Florentin Ipate & Gexiang Zhang (2022): *A key agreement protocol based on spiking neural P systems with anti-spikes*. Journal of Membrane Computing 4(4), pp. 341–351, doi:10.1007/s41965-022-00110-9.

[23] Tao Song, Alfonso Rodríguez-Patón, Pan Zheng & Xiangxiang Zeng (2017): *Spiking neural P systems with colored spikes*. IEEE Transactions on Cognitive and Developmental Systems 10(4), pp. 1106–1115, doi:10.1109/TCDS.2017.2785332.

[24] Răzvan Vasile, Marian Gheorghe & Ionuț Mihai Niculescu (2023): *Breaking RSA Encryption Protocol with Kernel P Systems*. doi:10.21203/rs.3.rs-2684530/v1.

[25] Tingfang Wu, Andrei Păun, Zhiqiang Zhang & Linqiang Pan (2017): *Spiking neural P systems with polarizations*. IEEE transactions on neural networks and learning systems 29(8), pp. 3349–3360, doi:10.1109/TNNLS.2017.2726119.

[26] Rafaa I Yahya, Siti Mariyam Shamsuddin, Salah I Yahya, Shafatnnur Hasan, Bisan Al-Salibi & Ghada Al-Khafaji (2016): *Image segmentation using membrane computing: a literature survey*. In: Bio-inspired Computing–Theories and Applications: 11th International Conference, BIC-TA 2016, Xi'an, China, October 28-30, 2016, Revised Selected Papers, Part I 11, Springer, pp. 314–335, doi:10.1007/978-981-10-3611-8_26.

[27] Claudio Zandron, Claudio Ferretti & Giancarlo Mauri (2001): *Solving NP-complete problems using P systems with active membranes*. In: Unconventional Models of Computation, UMC'2K: Proceedings of the Second International Conference on Unconventional Models of Computation,(UMC'2K), Springer, pp. 289–301, doi:10.1007/978-1-4471-0313-4_21.

[28] Ge-Xiang Zhang, Marian Gheorghe & Chao-Zhong Wu (2008): *A quantum-inspired evolutionary algorithm based on P systems for knapsack problem*. Fundamenta Informaticae 87(1), pp. 93–116.

[29] Ge-Xiang Zhang & Lin-Qiang Pan (2010): *A survey of membrane computing as a new branch of natural computing*. Chinese journal of computers 33(2), pp. 208–214, doi:10.3724/SP.J.1016.2010.00208.

[30] Gexiang Zhang, Mario J Pérez-Jiménez & Marian Gheorghe (2017): *Real-life applications with membrane computing*. 25, Springer, doi:10.1007/978-3-319-55989-6.

Benchmarking Local Robustness of High-Accuracy Binary Neural Networks for Enhanced Traffic Sign Recognition

Andreea Postovan Mădălina Eraşcu
Faculty of Mathematics and Informatics, West University of Timisoara
4 blvd. V. Parvan, 300223, Romania
{andreea.postovan99, madalina.erascu}@e-uvt.ro

Traffic signs play a critical role in road safety and traffic management for autonomous driving systems. Accurate traffic sign classification is essential but challenging due to real-world complexities like adversarial examples and occlusions. To address these issues, binary neural networks offer promise in constructing classifiers suitable for resource-constrained devices.

In our previous work, we proposed high-accuracy BNN models for traffic sign recognition, focusing on compact size for limited computation and energy resources. To evaluate their local robustness, this paper introduces a set of benchmark problems featuring layers that challenge state-of-the-art verification tools. These layers include binarized convolutions, max pooling, batch normalization, fully connected. The difficulty of the verification problem is given by the high number of network parameters (905k - 1.7 M), of the input dimension (2.7k-12k), and of the number of regions (43) as well by the fact that the neural networks are not sparse.

The proposed BNN models and local robustness properties can be checked at `https://github.com/ChristopherBrix/vnncomp2023_benchmarks/tree/main/benchmarks/traffic_signs_recognition`.

The results of the 4th International Verification of Neural Networks Competition (VNN-COMP'23) revealed the fact that 4, out of 7, solvers can handle many of our benchmarks randomly selected (minimum is 6, maximum is 36, out of 45). Surprisingly, tools output also wrong results or missing counterexample (ranging from 1 to 4). Currently, our focus lies in exploring the possibility of achieving a greater count of solved instances by extending the allotted time (previously set at 8 minutes). Furthermore, we are intrigued by the reasons behind the erroneous outcomes provided by the tools for certain benchmarks.

1 Introduction

Traffic signs play a crucial role in ensuring road safety and managing traffic flow, both in urban and highway driving. For autonomous driving systems, the accurate recognition and classification of traffic signs, known as *traffic sign classification (recognition)*, are essential components. This process involves two main tasks: firstly, isolating the traffic sign within a bounding box, and secondly, classifying the sign into a specific traffic category. The focus of this work lies on the latter task.

Creating a robust traffic sign classifier is challenging due to the complexity of real-world traffic scenes. Common issues faced by classifiers include a lack of *robustness* against *adversarial examples* [19] and occlusions [21]. *Adversarial examples* are inputs that cause classifiers to produce erroneous outputs, and *occlusions* occur naturally due to various factors like weather conditions, lighting, and aging, which make traffic scenes unique and diverse.

To address the lack of robustness, one approach is to formally verify that the trained classifier can handle both adversarial and occluded examples. Binary neural networks (BNNs) have shown promise

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 120–130, doi:10.4204/EPTCS.389.10

in constructing traffic sign classifiers, even in devices with limited computational resources and energy constraints, often encountered in autonomous driving systems. BNNs are neural networks (NNs) with binarized weights and/or activations constrained to ± 1, reducing model size and simplifying image recognition tasks.

The long-term goal of this work is to provide formal guarantees of specific properties, like robustness, that hold for a trained classifier. This objective leads to the formulation of the *verification problem*: given a trained model and a property to be verified, does the model satisfy that property? The verification problem is translated into a constrained satisfaction problem, and existing verification tools can be employed to solve it. However, due to its NP-complete nature [14], this problem is experimentally challenging for state-of-the-art tools.

In our previous work [16], we proposed high-accuracy BNN models explicitly for traffic sign recognition, with a thorough exploration of accuracy, model size, and parameter variations for the produced architectures. The focus was on BNNs with high accuracy and compact model size, making them suitable for devices with limited computation and energy resources, while also reducing the number of parameters to facilitate the verification task. The German Traffic Sign Recognition Benchmark (GTSRB) [5] was used for training, and testing involved similar images from GTSRB, as well as Belgian [1] and Chinese [4] datasets. This paper builds upon the models with the best accuracy from the previous study [16] and presents a set of benchmark problems to verify local robustness properties of these models.

The novelty of the proposed benchmarks lies in the fact that traffic signs recognition is done using binarized neural networks. To the best of our knowledge this was not done before [8, 18]. Compared to existing benchmarks. The types of layers used determine a complex verification problem and include *binarized convolution layers* to capture advanced features from the image dataset, *max pooling layers* for model size reduction while retaining relevant features, *batch normalization layers* for scaling, and *fully connected (dense) layers*. The difficulty of the verification problem is given by the high number of network parameters (905k - 1.7 M), of the input dimension (2.7k-12k), and of the number of regions (42) as well by the fact that the neural networks are not sparse. Discussions with organizers and competitors in the Verification of Neural Network Competition (VNN-COMP)[1] revealed that no tool competing in 2022 could handle the proposed benchmark. Additionally, in VNN-COMP 2023 [3], the benchmark was considered fairly complex by the main developer of the winning solver α, β-CROWN[2].

We publicly released our bechmark in May 2023. In the VNN-COMP 2023, which took place in July 2023, our benchmark was used in scoring, being nominated by at least 2 competing tools. 4, out of 7, tools were able to find an answer for the randomly generated instances. Most instances were solved by α, β-CROWN (39 out of 45) but it received penalties for 3 results due to either incorrect answer or missing counterexample. Most correct answers were given by Marabou[3] (18) with only 1 incorrect answer.

Currently, we are investigating the reasons why the tools were not able to solve all instances and why incorrect answers were given. Additionally, more tests will be performed on randomly generated answers and we will examine the particularities of the input images and of the trained networks which can not be handled by solvers due to timeout or incorrect answer.

The rest of the paper is organized as follows. In Section 2 we present related work focusing on comparing the proposed benchmark with others competing in VNN-COMP. Section 3 briefly describes deep neural networks, binarized neural networks and formulates the robustness property. In Section 4 we

[1] https://github.com/stanleybak/vnncomp2023/issues/2
[2] https://github.com/Verified-Intelligence/alpha-beta-CROWN
[3] https://github.com/NeuralNetworkVerification/Marabou

describe the anatomy of the trained neural networks whose local robustness is checked. In Section 5.1 we introduce the verification problem and its canonical representation (VNN-LIB and ONNX formats). Section 6 presents the methodology for benchmarks generation and the results of the VNN-COMP 2023.

2 Related Work

There exist many approaches for the verification of neural networks, see [20] for a survey, however few are tackling the verification of binarized neural networks.

Verifying properties using boolean encoding [15] is an alternative approach to validate characteristics of a specific category of neural networks, known as binarized neural networks. These networks possess binary weights and activations. The proposed technique involves reducing the verification problem from a mixed integer linear programming problem to a Boolean satisfiability. By encoding the problem in Boolean logic, they exploit the capabilities of modern SAT solvers, combined with a counterexample-guided search method, to verify various properties of these networks. A primary focus of their research is assessing the networks' resilience against adversarial perturbations. The experimental outcomes demonstrate the scalability of this approach when applied to medium-sized deep neural networks employed in image classification tasks. However their neural networks do not have convolution layers and can handle only a simple dataset like MNIST where images are black and white and there are just 10 classes to classify. Also, no tool implementing the approach was realeased to be tested.

Paper [6] focuses on verification of binarized neural network, extended the Marabou [14] tool to support *Sign Constrains* and verified a network that uses both binarized and non-binarized layers. For testing they used Fashion-MNIST dataset which was trained using XNOR-NET architecture and obtained the accuracy of only 70.97%. This extension could not be used in our case due to the fact that we have binarized convolution layers which the tool can not handle.

In the verification of neural networks competition (VNN-COMP), in 2022, there are various benchmarks subject to verification [2], however, there is none involving traffic signs. To the best of our knowledge there is only one paper which deals with traffic signs datasets [11] that is GTSRB. However, they considered only subsets of the dataset and their trained models consist of only fully connected (FC) layers with ReLU activation functions, not convolutions, ranging from 70 to 1300 neurons. Furthermore they do not mention the accuracy of their trained models to be able to compare it with ours. Moreover, the benchmarks from VNN-COMP 2022 [**?**] used for image classification tasks have are in Table 1. As one could observe, no benchmarks use binarized convolutions and batch normalization layers. Discussions with competition organizers revealed the fact that no tool from 2022 competition could handle our benchmark[4].

Table 1: Benchmarks proposed in the VNN-COMP 2022 for image classification tasks

Category	Benchmark	Network Types	#Neurons	Input Dimension
CNN & ResNet	Cifar Bias Field	Conv. + ReLU	45k	16
	Large ResNets	ResNet (Conv. + ReLU)	55k - 286k	3.1k - 12k
	Oval21	Conv. + ReLU	3.1k - 6.2k	3.1k
	SRI ResNet A/B	ResNet (Conv. + ReLU)	11k	3.1k
	VGGNet16	Conv. + ReLU + MaxPool	13.6M	1 - 95k
Fully-Connected	MNIST FC	FC. + ReLU	512 - 1.5k	784

The report of this year neural networks verification competition (VNN-COMP 2023) is in the draft version, but we present here the differences between our benchmark and the others. Table 2 taken

[4]See https://github.com/stanleybak/vnncomp2023/issues/2 intervention from user stanleybak on May 17, 2023

from the draft report presents all the scored benchmarks, i.e. benchmarks which were nominated by at least 2 competing tools and are used in their ranking. The column Network Type presents the types of layers of the trained neural network, the column # of Params represent the number of parameters of the trained neural network, the column Input Dimension represents the dimension of the input (for example, for an image with dimension 30x30 pixels and RGB channel the dimension is 30x30x3 which means that the verification problem contains 30x30x3 variables), the Sparsity column represents the degree of sparsity of the trained neural network and, finally, the column # of Regions represents the number of regions determined by the verification problem (for example, for our German Traffic Sign Recognition Benchmark there are 43 traffic signs classes). Our proposed benchmark, Traffic Signs Recognition, is more complex as the others as it involves cumulatively a high number of parameters, input dimension, number of regious and no sparsity.

Table 2: Benchmarks proposed in the VNN-COMP 2023

Name	Network Type	# of Params	Input Dimension	Sparsity	# of Regions
nn4sys	Conv, FC, Residual + ReLU, Sigmoid	33k - 37M	1-308	0-66%	1 - 11k
VGGNet16	Conv + ReLU + MaxPool	138M	150k	0-99%	1
Collins Rul CNN	Conv + ReLU, Dropout	60k - 262k	400-800	50-99%	2
TLL Verify Bench	FC + ReLU	17k - 67M	2	0%	1
Acas XU	FC + ReLU	13k	5	0-20%	1-4
cGAN	FC, Conv, ConvTranspose, Residual + ReLU, BatchNorm, AvgPool	500k-68M	5	0-40%	2
Dist Shift	FC + ReLU, Sigmoid	342k-855k	792	98.9%	1
ml4acopf	FC, Residual + ReLU, Sigmoid	4k-680k	22-402	0-7%	1-600
Traffic Signs Recogn	Conv+Sign+MaxPool+BatchNorm, FC,	905k-1.7M	2.7k-12k	0%	43
ViT	Conv, FC, Residual + ReLU, Softmax, BatchNorm	68k-76k	3072	0%	9

3 Theoretical Background

3.1 Deep Neural Networks

Neural networks, inspired by the human brain, are computational models composed of interconnected nodes called artificial neurons. These networks have gained attention for their ability to learn and perform complex tasks. The nodes compute outputs using *activation functions*, and synaptic *weights* determine the strength of connections between nodes. Training is achieved through optimization algorithms, such as *backpropagation*, which adjust the weights iteratively to minimize the network's error.

A *deep neural network (DNN)* [6] can be conceptualized as a directed graph, where the nodes, also known as neurons, are organized in *layers*. The input layer is responsible for receiving initial values, such as pixel intensities in the case of image inputs, while the output layer generates the final predictions or results. Hidden layers, positioned between the input and output layers, play a crucial role in extracting and transforming information. During the evaluation or inference process, the input values propagate through the network, layer by layer, using connections between neurons. Each neuron applies a specific mathematical operation to the inputs it receives, followed by the *activation function* that introduces *non-linearity* to the network. The activation function determines the neuron's output based on the weighted sum of its inputs and an optional bias term.

Different layer types are employed in neural networks to compute the values of neurons based on the preceding layer's neuron values. Those relevant for our work are introduced in Section 3.2.

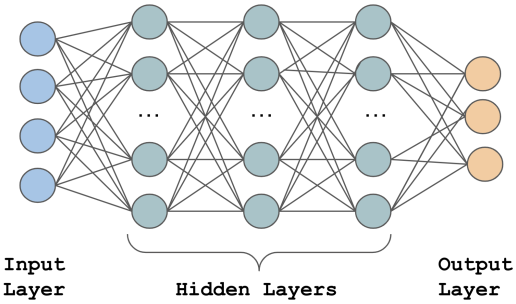


Figure 1: A fully connected DNN with 4 input nodes, 3 output nodes and 3 hidden layers

3.2 Binarized Neural Networks

A BNN [12] is a feedforward network where weights and activations are mainly binary. [15] describes BNNs as sequential composition of blocks, each block consisting of linear and non-linear transformations. One could distinguish between *internal* and *output blocks*.

There are typically several *internal blocks*. The layers of the blocks are chosen in such a way that the resulting architecture fulfills the requirements of accuracy, model size, number of parameters, for example. Typical layers in an internal block are: *1)* linear transformation (LIN) *2)* binarization (BIN) *3)* max pooling (MP) *4)* batch normalization (BN). A linear transformation of the input vector can be based on a fully connected layer or a convolutional layer. In our case is a convolution layer since our experiments have shown that a fully connected layer can not synthesize well the features of traffic signs, therefore, the accuracy is low. The linear transformation is followed either by a binarization or a max pooling operation. Max pooling helps in reducing the number of parameters. One can swap binarization with max pooling, the result would be the same. We use this sequence as Larq [9], the library we used in our experiments, implements convolution and binarization in the same function. Finally, scaling is performed with a batch normalization operation [13].

There is *one output block* which produces the predictions for a given image. It consists of a dense layer that maps its input to a vector of integers, one for each output label class. It is followed by function which outputs the index of the largest entry in this vector as the predicted label.

We make the observation that, if the MP and BN layers are omitted, then the input and output of the internal blocks are binary, in which case, also the input to the output block. The input of the first block is never binarized as it drops down drastically the accuracy.

3.3 Properties of (Binarized) Neural Networks: Robustness

Robustness is a fundamental property of neural networks that refers to their ability to maintain stable and accurate outputs in the presence of perturbations or adversarial inputs. Adversarial inputs are intentionally crafted inputs designed to deceive or mislead the network's predictions.

As defined by [15], *local robustness* ensures that for a given input x from a set χ, the neural network F remains unchanged within a specified perturbation radius ε, implying that small variations in the input space do not result in different outputs. The output for the input x is represented by its label l_x. We consider L_∞ norm defined as $||x||_\infty = \sup_n |x_n|$, but also other norms can be used, e.g. L_0 [17].

Definition 3.1 (Local robustness.)**.** A feedforward neural network F is locally ε-robust for an input $x, x \in \chi$, if there does not exist $\tau, ||\tau||_\infty \leq \varepsilon$, such that $F(x+\tau) \neq l_x$.

Global robustness [15] is an extension of the local robustness and it is defined as the expected maximum safe radius over a given test dataset, representing a collection of inputs.

Definition 3.2 (Global robustness.)**.** A feed-forward neural network F is globally ε-robust if for any $x, x \in \chi$, and $\tau, ||\tau||_\infty \leq \varepsilon$, we have that $F(x+\tau) = l_x$.

The definitions above can not be used in a computational setting. Hence, [14] proposes Definition 3.3 for local robustness which is equivalent to Definition 3.1.

Definition 3.3 (Local robustness.)**.** A network is ε-locally robust in the input x if for every x', such that $||x - x'||_\infty \leq \varepsilon$, the network assigns the same label to x and x'.

For our setting, the input is an image represented as a vector with values represented by the pixels. Hence, the inputs are the vector x and the perturbation ε.

This formula can also be applied to all inputs simultaneously (all images from test set of the dataset), in that case *global robustness* is addressed. However, the number of parameters involved in checking *global robustness* property increases enormously. Hence, in this paper, the benchmarks propose verification of local robustness only.

4 Anatomy of the Binarized Neural Networks

For benchmarking, we propose the two BNNs architectures for which we obtained the best accuracy [16], as well an additional one. More precisely, the best accuracy for GTSRB and Belgium datasets is $96,45\%$ and $88,17\%$, respectively, and was obtained for the architecture from Figure 2, with input size 64×64 (see Table 3). The number of parameters is almost 2M and the model size $225,67$ KiB (for the binary model) compared to $6932,48$ KiB (for the Float-32 equivalent). The best accuracy for Chinese dataset

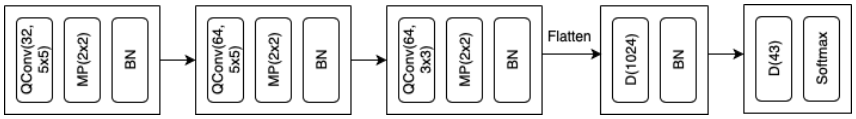


Figure 2: Accuracy Efficient Architecture for GTSRB and Belgium dataset

Table 3: Best results for the architecture from Figure 2. Dataset for train: GTSRB.

Input size	#Neur	Accuracy			#Params			Model Size (in KiB)	
		German	China	Belgium	Binary	Real	Total	Binary	Float-32
64px × 64px	1024	**96.45**	**81.50**	**88.17**	1772896	2368	1775264	225.67	6932.48

$(83,9\%)$ is obtained by another architecture, namely from Figure 3, with input size 48×48 (see Table 4). This architecture is more efficient from the point of view of computationally limited devices and formal verification having 900k parameters and $113,64$ KiB (for the binary model) and $3532,8$ KiB (for the Float-32 equivalent). Also, the second architecture gave the best average accuracy and the decrease in accuracy for GTSRB and Belgium is small, namely $1,17\%$ and $0,39\%$, respectively.

One could observe that the best architectures were obtained for input size images 48x48 and 64x64 pixels with max pooling and batch normalization layers which reduce the number of neurons, namely perform scaling which leads to good accuracy. We also propose for benchmarking an XNOR architecture, i.e. containing only binary parameters, (Figure 4) for which we obtained the best results for input size images 30x30 pixels and GTSRB (see Table 5).

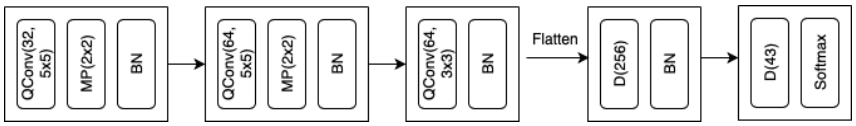


Figure 3: Accuracy Efficient Architecture for Chinese dataset

Table 4: Best results for the architecture from Figure 3. Dataset for train: GTSRB.

Input size	#Neur	Accuracy			#Params			Model Size (in KiB)	
		German	China	Belgium	Binary	Real	Total	Binary	Float-32
48px × 48px	256	**95.28**	**83.90**	**87.78**	904288	832	905120	113.64	3532.80

5 Model and Property Specification: VNN-LIB and ONNX Formats

The VNN-LIB (Verified Neural Network Library) format [10] is a widely used representation for encoding and exchanging information related to the verification of neural networks. It serves as a standardized format that facilitates the communication and interoperability of different tools and frameworks employed in the verification of neural networks.

The VNN-LIB format typically consists of two files that provide a detailed specification of the neural network model (see Section 5.1), along with relevant properties and constraints (see Section 5.2). These files encapsulate important information, including the network architecture, weights and biases, input and output ranges, and properties to be verified.

5.1 Model Representation

In machine learning, the representation of models plays a vital role in facilitating their deployment and interoperability across various frameworks and platforms. One commonly used format is the H5 format, which is an abbreviation for *Hierarchical Data Format version 5*. The H5 format provides a structured and efficient means of storing and organizing large amounts of data, including the parameters and architecture of machine learning models. It is widely supported by popular deep learning frameworks, such as TensorFlow and Keras, allowing models to be saved, loaded, and shared in a standardized manner.

However, while the H5 format serves as a convenient model representation for specific frameworks, it may lack compatibility when transferring models between different frameworks or performing model verification. This is where the *Open Neural Network Exchange* (ONNX) format comes into play. ONNX offers a vendor-neutral, open-source alternative that allows models to be represented in a standardized format, enabling seamless exchange and collaboration across multiple deep learning frameworks.

The VNN-LIB format, which is used for the formal verification of neural network models, leverages ONNX as its underlying model representation.

5.2 Property specification

For property specification, VNN-LIB standard uses the SMT-LIB format. The SMT-LIB (Satisfiability Modulo Theories-LIBrary) language [7] is a widely recognized formal language utilized for the formalization of Satisfiability Modulo Theories (SMT) problems.

A VNN-LIB file is structured as follows[5] and the elements involved have the following semantics for

[5]See, e.g. `https://github.com/apostovan21/vnncomp2023/blob/master/vnnlib/model_30_idx_1678_eps_1.00000.vnnlib`

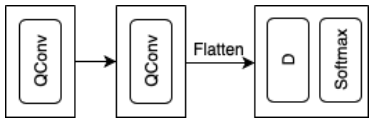


Figure 4: XNOR(QConv) architecture

Table 5: XNOR(QCONV) architecture. Image size: 30px × 30px. Dataset for train and test: GTSRB.

Model description	Acc	#Binary Params	Model Size (in KiB)	
			Binary	Float-32
QConv(16, 3×3), QConv(32, 2×2), D(43)	81.54	1005584	122.75	3932.16

the considered image classification task:

1. definition of input variables representing the values of the pixels X_i ($i = \overline{1,P}$, where P is the dimension of the input image: $N \times M \times 3$ pixels). For the file above, there are 2700 variables as the image has dimension 30×30 and the channel used is RGB.
2. definition of the output variables representing the values Y_j ($j = \overline{1,L}$, where L is the number of classes of the images in the dataset). For the file above, there are 43 variables as the GTSRB categorises the traffic signs images into 43 classes.
3. bounding constraints for the variables input variables. Definition 5.1 is used for generating the property taking into account that vector x (its elements are the values of the pixels of the image) and ε (perturbation) are known. For example, if $\varepsilon = 10$ and the value of the pixel X'_{2699} of the image with index 1678 from GTSRB is 24, the generated constraints for finding the values of the perturbed by ε pixel X_{2699} for which the predicted label still holds is:

```
(assert (<= X_2699 34.00000000))
(assert (>= X_2699 14.00000000))
```

4. constraints involving the output variables assessing the value of the output label. For example, if the verification problem is formulated as: *Given the image with index* 1678*, the perturbation* $\varepsilon = 10$ *and the trained model, find if the perturbed images are in class* 38, the generated constraints are as follows which actually represents the negation of the property to be checked:

```
(assert (or (>= Y_0 Y_38)
            ...
            (>= Y_37 Y_38)
            (>= Y_39 Y_38)
            ...
            (>= Y_42 Y_38)))
```

6 Benchmarks Proposal and Experimental Results of the VNN-COMP 2023

To meet the requirements of the VNN-COMP 2023, the benchmark datasets must conform to the ONNX format for defining the neural networks, while the problem specifications are expected to adhere to the

VNN-LIB format. Therefore, we have prepared a benchmark set comprising the BNNs introduced in Section 4 that have been encoded in the ONNX format. In order to assess the adversarial robustness of these networks, the problem specifications encompassed perturbations within the infinity norm around zero, with radius denoted as $\varepsilon = \{1, 3, 5, 10, 15\}$. To achieve this, we randomly selected three distinct images from the test set of the GTSRB dataset for each model and have generated the VNNLIB files for each epsilon in the set, in the way we ended up having 45 VNNLIB files in total. We were constrained to generate the small benchmark which includes just 45 VNNLIB files because of the total timeout which should not exceed 6 hour, this is the maximum timeout for a solver to address all instances, consequently a timeout of 480 seconds was allocated for each instance. For checking the generated VNNLIB specification files for submitted in the VNNCOMP 2023 as specified above as well as to generate new ones you can check `https://github.com/apostovan21/vnncomp2023`.

Our benchmark was used for scoring the competing tools. The results for our benchmark, as presented by the VNN-COMP 2023 organizers, are presented in Table 6.

Table 6: VNN-COMP 2023 Results for Traffic Signs Recognition Benchmark

#	Tool	Verified	Falsified	Fastest	Penalty	Score	Percent
1	Marabou	0	18	0	1	30	100%
2	PyRAT	0	7	0	1	-80	0%
3	NeuralSAT	0	31	0	4	-290	0%
4	alpha-beta-CROWN	0	39	0	3	-60	0%

The meaning of the columns is as follows. Verified is number of instances that were UNSAT (no counterexample) and proven by the tool. Falsifieid is number that were SAT (counterexample was found) and reported by the tool. Fastest is the number where the tool was fastest (this did not impact the scoring in this year competition). Penalty is the number where the tool gave the incorrect result or did not produce a valid counterexample. Score is the sum of scores (10 points for each correct answer and -150 for incorrect ones). Percent is the score of the tool divided by the best score for the benchmark (so the tool with the highest score for each benchmark gets 100) and was used to determine final scores across all benchmarks.

Currently, we are investigating if the number of solved instances could be higher if the time is increased (the deadline used was 8 minutes). Also, it is interesting why the tools gave incorrect results for some benchmarks.

7 Conclusions

Building upon our prior study that introduced precise binarized neural network models for traffic sign recognition, this study presents standardized challenges to gauge the resilience of these networks to local variations. These challenges were entered into the VNN-COMP 2023 evaluation, where 4 out of 7 tools produced results. Our current emphasis is on investigating the potential for solving more instances by extending the time limit (formerly set at 8 minutes). Additionally, we are keen to comprehend the factors contributing to incorrect outputs from the tools on specific benchmark tasks.

Acknowledgements

This work was supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI - UEFISCDI, project number PN-III-P1-1.1-TE-2021-0676, within PNCDI III.

References

[1] *Belgian Traffic Sign Database*. https://www.kaggle.com/datasets/shazaelmorsh/trafficsigns. Accessed: March 25th, 2023.

[2] *Benchmarks of the 3rd International Verification of Neural Networks Competition (VNN-COMP'22)*. https://github.com/ChristopherBrix/vnncomp2022_benchmarks. Accessed: February 22nd, 2023.

[3] *Benchmarks of the 4th International Verification of Neural Networks Competition (VNN-COMP'23)*. https://github.com/ChristopherBrix/vnncomp2023_benchmarks. Accessed: July 26th, 2023.

[4] *Chinese Traffic Sign Database*. https://www.kaggle.com/datasets/dmitryyemelyanov/chinese-traffic-signs. Accessed: March 25th, 2023.

[5] *German Traffic Sign Recognition Benchmark*. https://www.kaggle.com/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign?datasetId=82373&language=Python. Accessed: March 25th, 2023.

[6] Guy Amir, Haoze Wu, Clark Barrett & Guy Katz (2021): *An SMT-Based Approach for Verifying Binarized Neural Networks*. In Jan Friso Groote & Kim Guldstrand Larsen, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 203–222, doi:10.1007/978-3-030-72013-1_11.

[7] Clark Barrett, Aaron Stump, Cesare Tinelli et al. (2010): *The SMT-LIB Standard: Version 2.0*. In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 13, p. 14.

[8] Dan Ciregan, Ueli Meier & Jürgen Schmidhuber (2012): *Multi-Column Deep Neural Networks for Image Classification*. In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, pp. 3642–3649, doi:10.1109/CVPR.2012.6248110.

[9] Lukas Geiger & Plumerai Team (2020): *Larq: An Open-Source Library for Training Binarized Neural Networks*. *Journal of Open Source Software* 5(45), p. 1746, doi:10.21105/joss.01746.

[10] Dario Guidotti, Stefano Demarchi, Armando Tacchella & Luca Pulina (2023): *The Verification of Neural Networks Library (VNN-LIB)*. Available at https://www.vnnlib.org.

[11] Xingwu Guo, Ziwei Zhou, Yueling Zhang, Guy Katz & Min Zhang (2023): *OccRob: Efficient SMT-Based Occlusion Robustness Verification of Deep Neural Networks*. In Sriram Sankaranarayanan & Natasha Sharygina, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Nature Switzerland, Cham, pp. 208–226, doi:10.1007/978-3-031-30823-9_11.

[12] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv & Yoshua Bengio (2016): *Binarized Neural Networks*. *Advances in Neural Information Processing Systems* 29.

[13] Sergey Ioffe & Christian Szegedy (2015): *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. In: *International conference on machine learning*, PMLR, pp. 448–456, doi:10.48550/arXiv.1502.03167.

[14] Guy Katz, Clark Barrett, David L Dill, Kyle Julian & Mykel J Kochenderfer (2022): *Reluplex: A Calculus for Reasoning about Deep Neural Networks*. *Formal Methods in System Design* 60(1), pp. 87–116, doi:10.1007/s10703-021-00363-7.

[15] Nina Narodytska, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv & Toby Walsh (2018): *Verifying properties of binarized deep neural networks*. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, 32, doi:10.1609/aaai.v32i1.12206.

[16] Andreea Postovan & Mădălina Eraşcu (2023): *Architecturing Binarized Neural Networks for Traffic Sign Recognition*. *arXiv preprint arXiv:2303.15005*, doi:10.48550/arXiv.2303.15005.

[17] Wenjie Ruan, Min Wu, Youcheng Sun, Xiaowei Huang, Daniel Kroening & Marta Kwiatkowska (2019): *Global robustness evaluation of deep neural networks with provable guarantees for the hamming distance*. *IJCAI-19*, doi:10.24963/ijcai.2019/824.

[18] Johannes Stallkamp, Marc Schlipsing, Jan Salmen & Christian Igel (2012): *Man vs. Computer: Benchmarking Machine Learning Algorithms for Traffic Sign Recognition*. Neural networks 32, pp. 323–332, doi:10.1016/j.neunet.2012.02.016.

[19] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow & Rob Fergus (2013): *Intriguing Properties of Neural Networks*. arXiv preprint arXiv:1312.6199, doi:10.48550/arXiv.1312.6199.

[20] Huan Zhang, Kaidi Xu, Shiqi Wang & Cho-Jui Hsieh (2022): *Formal Verification of Deep Neural Networks: Theory and Practice*. https://neural-network-verification.com/. Tutorial at AAAI 2022.

[21] Jianming Zhang, Wei Wang, Chaoquan Lu, Jin Wang & Arun Kumar Sangaiah (2020): *Lightweight Deep Network for Traffic Sign Classification*. Annals of Telecommunications 75, pp. 369–379, doi:10.1007/s12243-019-00731-9.

Symmetric Functions over Finite Fields

Mihai Prunescu
Research Center for Logic, Optimization and Security (LOS),
Faculty of Mathematics and Computer Science,
University of Bucharest, Academiei 14, 010014 Bucharest, Romania
and Simion Stoilow Institute of Mathematics of the Romanian Academy,
Research unit 5, P. O. Box 1-764, RO-014700 Bucharest, Romania.
mihai.prunescu@imar.ro, mihai.prunescu@gmail.com

The number of linear independent algebraic relations among elementary symmetric polynomial functions over finite fields is computed. An algorithm able to find all such relations is described. It is proved that the basis of the ideal of algebraic relations found by the algorithm consists of polynomials having coefficients in the prime field $\mathbb{F}_p$.

A.M.S.-Classification: 14-04, 15A03.

1 Introduction

The interpolation problem for symmetric functions over finite fields does not have a unique solution in terms of elementary symmetric polynomials. That is why it is useful to know more about the set of solutions, as sometimes we need a solution which is easier to express or faster to evaluate. The general situation will be described in the lines below. Some steps are detailed in the following sections.

Let $\mathbb{F}_q$ be a finite field of characteristic p, a prime number. Let $\mathbb{F}_q[X_1,\ldots,X_n]$ be the $\mathbb{F}_q$-algebra of polynomials in variables $X_1,\ldots,X_n$ over $\mathbb{F}_q$. The symmetric group S_n acts on $\mathbb{F}_q[X_1,\ldots,X_n]$ as a group of automorphisms of $\mathbb{F}_q$-algebras and the subalgebra Q of fixed points is usually called the ring of symmetric polynomials. The algebra Q is again a polynomial ring freely generated by the elementary symmetric polynomials

$$e_i = \sum_{1\leq j_1<j_2<\cdots<j_i\leq n} X_{j_1}\ldots X_{j_i}$$

for $i = 1,\ldots,n$. So $Q = \mathbb{F}_q[e_1,\ldots,e_n]$.

The vector space Q has a basis given by the monomial symmetric functions $e_1^{k_1}\ldots e_n^{k_n}$.

There is an obvious homomorphism of $\mathbb{F}_q$-algebras from $\mathbb{F}_q[X_1,\ldots,X_n]$ to the $\mathbb{F}_q$-algebra $\mathscr{F}(q,n)$ of functions from $\mathbb{F}_q^n$ to $\mathbb{F}_q$. Considering the natural action of S_n on $\mathscr{F}(q,n)$ one can easily see that this homomorphism is S_n-equivariant, i. e. it commutes with the action of S_n. From this it follows that Q maps to the subring $\mathscr{S}(q,n)$ of the symmetric functions in $\mathscr{F}(q,n)$.

Since $x^q = x$ for all $x \in \mathbb{F}_q$, it is clear that the above homomorphism factors through the quotient:

$$\mathbb{F}_q\{X_1,\ldots,X_n\} = \mathbb{F}_q[X_1,\ldots,X_n]/ < X_1^q - X_1,\ldots,X_n^q - X_n >,$$

whose basis consists of those monomials whose exponents are smaller than q. So $\mathbb{F}_q\{X_1,\ldots,X_n\}$ is finite of dimension q^n and has q^{q^n} elements. By using interpolation, one can deduce that the above homomorphism gives an isomorphism $\mathbb{F}_q\{X_1,\ldots,X_n\} \simeq \mathscr{F}(q,n)$. This will follow by a cardinality argument.

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 131–143, doi:10.4204/EPTCS.389.11

The ideal $I =< X_1^q - X_1, \ldots, X_n^q - X_n >$ is obviously stable under the action of S_n. Hence $\mathbb{F}_q\{X_1, \ldots, X_n\}$ is an S_n-module for the previously given action and that the isomorphism $\mathbb{F}_q\{X_1, \ldots, X_n\} \simeq \mathscr{F}(q,n)$ is an isomorphism of S_n-modules. Under this map we see that the basis of Q made of monomial symmetric functions is mapped to a basis of the symmetric function algebra $\mathscr{S}(q,n)$ so that the image of Q in $\mathbb{F}_q\{x_1, \ldots, x_n\}$ is isomorphic to $\mathscr{S}(q,n)$.

If $E_1, \ldots, E_n$ are new variables, one can also consider the quotient

$$\mathbb{F}_q\{E_1, \ldots, E_n\} = \mathbb{F}_q[E_1, \ldots, E_n] / < E_1^q - E_1, \ldots, E_n^q - E_n >,$$

which is again finite of dimension q^n over $\mathbb{F}_q$. Consider the embedding

$$\mathbb{F}_q[E_1, \ldots, E_n] \to \mathbb{F}_q[X_1, \ldots, X_n]$$

generated by the substitutions $E_k \rightsquigarrow e_k(X_1, \ldots, X_n)$. This embedding descends to a homomorphism

$$\Psi : \mathbb{F}_q\{E_1, \ldots, E_n\} \to \mathbb{F}_q\{X_1, \ldots, X_n\} \simeq \mathscr{F}(q,n),$$

because the embedding transports $J =< E_1^q - E_1, \ldots, E_n^q - E_n >$ inside I.

Let $\vec{X}$ denote the tuple of variables $(X_1, \ldots, X_n)$. By diagram chasing one checks that $\Psi(\mathbb{F}_q\{E_1, \ldots, E_n\})$ is the image of $\mathbb{F}_q[e_1(\vec{X}), \ldots, e_n(\vec{X})]$ under the natural projection $\mathbb{F}_q[X_1, \ldots, X_n] \to \mathbb{F}_q\{X_1, \ldots, X_n\}$. By standard combinatorics one knows that the dimension of the space of the symmetric functions $\mathscr{S}(q,n)$ is:

$$\binom{n+q-1}{n}.$$

Summarising we have that $\Psi(\mathbb{F}_q\{E_1, \ldots, E_n\})$ is isomorphic with the ring of symmetric functions $\mathscr{S}(q,n)$ and therefore the dimension of the kernel of Ψ is:

$$q^n - \binom{n+q-1}{n}.$$

We call this kernel $\mathscr{I}(q,n)$. This kernel can be seen as the set of all algebraic relations between elementary symmetric functions over a finite field. For example, if we consider the symmetric functions with two variables $e_1 = X_1 + X_2$ and $e_2 = X_1 X_2$ as polynomial functions $\mathbb{F}_2 \to \mathbb{F}_2$, then:

$$e_1 e_2 = (X_1 + X_2) X_1 X_2 = X_1^2 X_2 + X_1 X_2^2 = 2 X_1 X_2 = 0,$$

so those functions fulfill the algebraic identity $e_1 e_2 = 0$. This doesn't happens when those symmetric functions are considered over $\mathbb{C}$, because it is known that there the corresponding functions are algebraically independent.

The scope of this article is to show an algorithm that computes a basis of the kernel $\mathscr{I}(q,n)$ as $\mathbb{F}_q$-vector space. The general Buchberger-Möller algorithm, see [2], finds a basis of the ideal in the sense of ring theory. The algorithm presented here finds a basis as a vector space. In practical applications, if someone finds an interpolation formula for some symmetric function, and looks for a shorter definition, the basis as a vector space is more useful. Moreover, the algorithm given here for this particular problem works faster then Buchberger-Möller algorithm.

2 Definitions and notations

Consider a finite field $\mathbb{F}_q$ of characteristic p. The elements of $\mathbb{F}_q$ are ordered in some way, and this order is fixed. For the rest of the paper we fix a natural number $n \geq 1$ and two sets of variables: $E_1, \ldots, E_n$ and $X_1, \ldots, X_n$.

Definition 2.1 The set $\mathrm{Mon}(q,n)$ is the set of all monomials $E_1^{\alpha_1} E_2^{\alpha_2} \ldots E_n^{\alpha_n}$ with $0 \leq \alpha_i < q$. There are q^n many such monomials.

Definition 2.2 Let $\mathbb{F}_q\{E_1, \ldots, E_n\}$ be the vector space over $\mathbb{F}_q$ freely generated by $\mathrm{Mon}(q,n)$.

$\mathbb{F}_q\{E_1, \ldots, E_n\}$ has dimension q^n over $\mathbb{F}_q$. It has also a canonical structure of finite ring induced by the epimorphism:

$$s : \mathbb{F}_q[E_1, \ldots, E_n] \longrightarrow \mathbb{F}_q\{E_1, \ldots, E_n\}$$

with $\mathrm{Ker}(s) = (E_1^q - E_1, \ldots, E_n^q - E_n)$ as ideal in $\mathbb{F}_q[E_1, \ldots, E_n]$. Observe that $\mathbb{F}_q \models \forall x\, x^q = x$.

Definition 2.3 For $f : \mathbb{F}_q^n \to \mathbb{F}_q$ and a permutation $\sigma \in S_n$ we define $f^\sigma(\vec{X}) = f(\sigma(\vec{X}))$ where $\sigma(\vec{X}) = (X_{\sigma(1)}, \ldots, X_{\sigma(n)})$. The function f is called symmetric if for all $\sigma \in S_n$, $f = f^\sigma$.

Definition 2.4 Let $\mathscr{F}(q,n)$ denote the set of all functions $f : \mathbb{F}_q^n \to \mathbb{F}_q$ and $\mathscr{S}(q,n) \subset \mathscr{F}(q,n)$ the subset of all symmetric functions. Both sets equipped with the point-wise operations are finite rings and finite vector spaces over $\mathbb{F}_q$.

Definition 2.5 For every $F \in \mathbb{F}_q[X_1, \ldots, X_n]$ and $\sigma \in S_n$ we define $F^\sigma(\vec{X}) = F(\sigma(\vec{X}))$, where $\sigma(\vec{X}) = (X_{\sigma(1)}, \ldots, X_{\sigma(n)})$. F is called symmetric if for all $\sigma \in S_n$, $F^\sigma = F$.

Definition 2.6 For $0 \leq k \leq n$ denote by $\mathscr{P}_k^n$ be the set of k-element subsets of $\{1, \ldots, n\}$. Recall that the elementary symmetric polynomials $e_k(X_1, \ldots, X_n)$ are defined as:

$$e_k(X_1, \ldots, X_n) = \sum_{J \in \mathscr{P}_k^n} \prod_{i \in J} X_i.$$

Definition 2.7 Consider the function

$$\Phi : \mathbb{F}_q\{E_1, \ldots, E_n\} \longrightarrow \mathscr{S}(q,n)$$

defined such that

$$\forall \vec{a} \in \mathbb{F}_q^n \;\; \Phi(f)(\vec{a}) = f(e_1(\vec{a}), \ldots, e_n(\vec{a})),$$

where $f \in \mathbb{F}_q\{E_1, \ldots, E_n\}$. Here we understand E_k as a symbol for the elementary symmetric polynomial e_k. Φ is a well defined homomorphism of finite rings and finite $\mathbb{F}_q$-vector spaces.

Definition 2.8 The ideal $\mathscr{I}(q,n) = \mathrm{Ker}(\Phi) \subset \mathbb{F}_q\{E_1, \ldots, E_n\}$ is the **ideal of algebraic relations** between elementary symmetric functions over $\mathbb{F}_q$.

$\mathscr{I}(q,n)$ is also a vector subspace of $\mathbb{F}_q\{e_1, \ldots, e_n\}$.

We also consider the following chain of homomorphisms:

$$\mathbb{F}_q\{E_1, \ldots, E_n\} \xrightarrow{\Psi} \mathbb{F}_q\{X_1, \ldots, X_n\} \xrightarrow{\Gamma} \mathscr{F}(q,n),$$

where $\Psi(P) = P(\vec{e}(\vec{X}))$ is the substitution homomorphism and Γ is the homomorphism associating to every polynomial Q its polynomial function. Of course $\Phi = \Gamma \circ \Psi$.

3 The number of algebraic relations

Definition 3.1 Let $\mathrm{WM}(q,n)$ be the set of all weakly monotone increasing tuples $(a_1,\ldots,a_n)$ in $\mathbb{F}_q$ according to the fixed order. Denote by $\mathrm{wm}(q,n)$ the cardinality of the set $\mathrm{WM}(q,n)$.

Lemma 3.2

$$\dim_{\mathbb{F}_q} \mathscr{S}(q,n) = \mathrm{wm}(q,n) = \binom{n+q-1}{n}.$$

Proof: In order to define an $f \in \mathscr{S}(q,n)$, it is enough to define its values for every $\vec{\imath} \in \mathrm{WM}(q,n)$. This number of sequences is the same as the number of possibilities to put n unsigned balls in q numbered urns, see [1]. As done there, the q urns can be represented by $q+1$ vertical bars and the n balls as n circles. A possible distribution as it follows:

$$|\circ||\circ\circ||$$

Because the first and the last symbols in distribution must be bars, we have to distribute n circles in $n+q-1$ positions and to complete the remaining positions with bars. There are

$$\binom{n+q-1}{n}$$

possibilities to do so. □

Definition 3.3 Consider the following matrix $M(q,n) \in \mathscr{M}(\mathrm{wm}(q,n) \times q^n, \mathbb{F}_q)$. The rows of $M(q,n)$ are indexed using the tuples $\vec{\imath} \in \mathrm{WM}(q,n)$, the columns are indexed using the monomials $m \in \mathrm{Mon}(q,n)$, and if $M(q,n) = (a(\vec{\imath},m) \mid \vec{\imath} \in \mathrm{WM}(q,n), m \in \mathrm{Mon}(q,n))$,

$$a(\vec{\imath},m) = [\Phi(m)](\vec{\imath}) = e_1^{\alpha_1}(\vec{\imath}) \ldots e_n^{\alpha_n}(\vec{\imath}),$$

where $m = E_1^{\alpha_1} \ldots E_n^{\alpha_n}$.

Lemma 3.4 *The morphism* $\Phi : \mathbb{F}_q\{E_1,\ldots,E_n\} \to \mathscr{S}(q,n)$ *with* $\mathrm{Ker}\,\Phi = \mathscr{I}(q,n)$ *is surjective.*

Proof: The proof consists of two steps.

Step 1: Let $f : \mathbb{F}_q^n \to \mathbb{F}_q$ be some function, for the moment not necessarily symmetric. For $a \in \mathbb{F}_q$ define the polynomial $h_a \in \mathbb{F}_q[X]$:

$$h_a(X) = \prod_{\lambda \in \mathbb{F}_q \setminus \{a\}} \frac{X-\lambda}{a-\lambda}.$$

Observe that $h_a(a) = 1$ and $h_a(\mathbb{F}_q \setminus \{a\}) = 0$. For a tuple $\vec{a} \in \mathbb{F}_q^n$ define $h_{\vec{a}} \in \mathbb{F}_q[X_1,\ldots,X_n]$:

$$h_{\vec{a}}(\vec{X}) = h_{a_1}(X_1) \ldots h_{a_n}(X_n).$$

A polynomial interpolating f is:

$$H(\vec{X}) = \sum_{\vec{a} \in \mathbb{F}_q^n} h_{\vec{a}}(\vec{X}) f(\vec{a}).$$

We observe that for all $\sigma \in S_n$, $(h_{\vec{a}})^\sigma = h_{\sigma^{-1}(\vec{a})}$. If the function f is symmetric, then $f^\sigma = f$ and it follows:

$$H^\sigma(\vec{X}) = \sum_{\vec{a}\in\mathbb{F}_q^n} h_{\vec{a}}^\sigma(\vec{X}) f(\vec{a}) = \sum_{\sigma^{-1}(\vec{a})\in\mathbb{F}_q^n} h_{\sigma^{-1}(\vec{a})}(\vec{X}) f(\sigma^{-1}(\vec{a})) = H(\vec{X}).$$

We proved that the interpolation algorithm applied to a symmetric function leads to a symmetric polynomial. Observe also that all exponents occurring in H are $< q$.

Step 2: We repeat the argument that a symmetric polynomial is a polynomial in elementary symmetric polynomials as given in [6] and reformulated in [5]. The following total order is defined over the set of monomials in $\vec{X}$: $X_1^{\alpha_1}\dots X_n^{\alpha_n} < X_1^{\beta_1}\dots X_n^{\beta_n}$ if and only if $\sum\alpha_i < \sum\beta_i$ or $\sum\alpha_i = \sum\beta_i$ but $(\alpha_i) < (\beta_i)$ lexicographically. This is the graded lexicographic order. For a polynomial $H(\vec{X}) \in \mathbb{F}_q[\vec{X}]$ define $\mathrm{Init}(H)$ to be the maximal monomial occurring in H, according to this order. It follows from symmetry that $\mathrm{Init}(H)$ has the form $cX_1^{\gamma_1}\dots X_n^{\gamma_n}$ with $\gamma_1 \geq \gamma_2 \geq \dots \geq \gamma_n$ with $c \in \mathbb{F}_q \setminus \{0\}$. Consider the polynomial:

$$H_1(\vec{X}) = H(\vec{X}) - ce_1(\vec{X})^{\gamma_1-\gamma_2} e_2(\vec{X})^{\gamma_2-\gamma_3} \dots e_{n-1}(\vec{X})^{\gamma_{n-1}-\gamma_n} e_n(\vec{X})^{\gamma_n}.$$

Observe that $\mathrm{Init}(H_1) < \mathrm{Init}(H)$. Continue by constructing in the same way a polynomial H_2 with $\mathrm{Init}(H_2) < \mathrm{Init}(H_1)$, and so on. This process ends in finitely many steps. Adding the S_i-monomials defined during the process, one gets a polynomial $F \in \mathbb{F}_q[e_1,\dots,e_n]$ with the property that for all $\vec{a} \in \mathbb{F}_q^n$, $F(e_1(\vec{a}),\dots,e_n(\vec{a})) = f(\vec{a})$. Finally, observe that all exponents occurring in F are again $< q$, so $F \in \mathbb{F}_q\{e_1,\dots,e_n\}$ and $\Phi(F) = f$.

□

Theorem 3.5 *The rank of the matrix $M(q,n)$ is maximal:*

$$\mathrm{rank}\, M(q,n) = \mathrm{wm}(q,n) = \binom{n+q-1}{n}.$$

The dimension of the ideal $\mathscr{I}(q,n)$ of algebraic relations as a vector space over $\mathbb{F}_q$ is:

$$\dim_{\mathbb{F}_q} \mathscr{I}(q,n) = q^n - \mathrm{wm}(q,n) = q^n - \binom{n+q-1}{n}.$$

Proof: Follows directly from Lemma 3.2 and the Lemma 3.4. □

Remark 3.6 If q is kept constant and $n \to \infty$,

$$\frac{\dim_{\mathbb{F}_q} \mathscr{I}(q,n)}{\dim_{\mathbb{F}_q} \mathbb{F}_q\{S_1,\dots,S_n\}} \longrightarrow 1.$$

If n is kept constant and $q \to \infty$,

$$\frac{\dim_{\mathbb{F}_q} \mathscr{I}(q,n)}{\dim_{\mathbb{F}_q} \mathbb{F}_q\{S_1,\dots,S_n\}} \longrightarrow 1 - \frac{1}{n!}.$$

Indeed, if q is kept constant and $n \to \infty$,

$$\frac{q^n - \binom{n+q-1}{n}}{q^n} = 1 - \frac{1}{(q-1)!}\frac{(n+1)\dots(n+q-1)}{q^n} \longrightarrow 1.$$

If n is kept constant and $q \to \infty$,

$$\frac{q^n - \binom{n+q-1}{n}}{q^n} = 1 - \frac{1}{n!}\frac{q(q+1)\dots(q+n-1)}{q^n} \longrightarrow 1 - \frac{1}{n!}.$$

Remark 3.7 Define the set of non-monotone tuples $\mathrm{NM}(q,p)$ as the set of tuples $(a_1,\dots,a_n)$ with $a_i \in \mathbb{F}_q$ such that there are $1 \le i < j \le n$ with $a_i > a_j$ according to the order fixed on $\mathbb{F}_q$. Let $\mathrm{nm}(q,n)$ be the cardinality of the set $\mathrm{NM}(q,n)$. According to Theorem 3.5, $\mathscr{I}(q,n)$ has dimension $\mathrm{nm}(q,n)$. But is there any natural correspondence between $\mathrm{NM}(q,n)$ and a basis of the vector space $\mathscr{I}(q,n)$? As far I know, this problem is open.

Remark 3.8 Recall the chain of homomorphisms:

$$\mathbb{F}_q\{E_1,\dots,E_n\} \xrightarrow{\Psi} \mathbb{F}_q\{X_1,\dots,X_n\} \xrightarrow{\Gamma} \mathscr{F}(q,n),$$

where $\Psi(P) = P(\vec{e}(\vec{X}))$ is the substitution homomorphism and Γ is the homomorphism associating to every polynomial Q its polynomial function, and $\Phi = \Gamma \circ \Psi$. Using the interpolation part of the proof of Lemma 3.4 one sees that Γ is an isomorphism of rings and vector spaces. Indeed, Γ is a surjective homomorphism and both rings have q^{q^n} elements.

Corollary 3.9 $\mathrm{Ker}\,\Psi = \mathscr{I}(q,n)$ *and* $\mathrm{Im}\,\Psi = \Gamma^{-1}(\mathscr{S}(q,n))$. *Consequently, the subring of symmetric polynomials in* $\mathbb{F}_q\{X_1,\dots,X_n\}$ *is a vector space of dimension* $\mathrm{wm}(q,n)$ *over* $\mathbb{F}_q$.

4 Deduction procedure

Before presenting the specific algorithm that generates a basis of the ideal of algebraic identities, I briefly recall the Gauss Algorithm over some field K.

Gauss Algorithm: Consider the extended matrix M of a system of e many linear equations in v unknowns. The matrix has e rows and $v+1$ columns. The algorithm finds out if the system is solvable. It also finds out on how many parameters the solution depends and generates a parametric solution of the system.

At the beginning one completes an array π with the values $\pi(0) = 0$, …, $\pi(v-1) = v-1$. This array will accumulate the total permutation of lines, produced by the algorithm.

Let r be a variable in which the algorithm computes the rank of the restricted matrix of the system. Let $m = \min(e,v)$. The Gauss Algorithm is a repetition of Gauss Steps according to a variable s running from 0 to $m-1$ inclusively. If the Gauss Step returns *false* at step s, the loop is broken. If not, r is increased by 1 and the corresponding Gauss Step is performed.

The Gauss Step of s works as follows. In every column c with $s \le c < v$ one looks for a row w with $s \le w < e$ such that $M(w,c) \neq 0$. If no such an element is found, the Gauss Step returns *false* and stops. Once such an element is found: if $c \neq s$, the columns c and s are inter-changed and also, the values of $\pi(s)$ and $\pi(c)$ are inter-changed. Further, if $s \neq w$, the rows w and s are inter-changed. Let now $a = M(s,s)$. Then the row s is divided by a. Further, from all the rows d with $d > s$ is subtracted the row s times a field value b, where $b = M(d,r)$. Because of this, all elements of the column s which are below $M(s,s)$ become 0. Finally, the Gauss Step returns *true*.

The decision on the number of solutions is made as follows. If the restricted rank r is equal with the number of unknowns v, there is only one solution. If $r < e$, there are two possibilities: (1) there are elements $M(v,i) \neq 0$ with $i \geq r$. In this case there are no solutions. Or (2), there are no such elements. Then the unknowns which corresponds now, after all permutations of columns (unknowns), to the columns r, $r+1, \ldots, v$, are independent parameters. The other unknowns are computed successively, started to the unknown corresponding to column $v-1$ after the permutation, continuing to the unknown corresponding to the column $v-2$, and finally ending with the unknown corresponding to the column 0.

□

The following algorithm is able to find a basis over $\mathbb{F}_q$ for the vector space $\mathscr{I}(q,n)$ of algebraic relations between elementary symmetric functions over $\mathbb{F}_q$.

1. Consider q^n many new unknowns Y_m indexed using the set $\mathrm{Mon}(q,n)$, and the following homogenous system Σ of $\mathrm{wm}(q,n)$ many linear equations indexed using the set $\mathrm{WM}(q,n)$:

$$(\vec{t}): \quad \sum_{m \in \mathrm{Mon}(q,n)} [\Phi(m)](\vec{t}) Y_m = 0.$$

   The matrix of this linear homogenous system is the matrix $M(q,n)$ defined in the previous section. One sees that for any polynomial $P \in \mathbb{F}_q\{e_1, \ldots, e_n\}$ following holds:

$$P(\vec{e}) = \sum_{m \in \mathrm{Mon}(q,n)} y_m \; m(\vec{e}) \;\; \in \;\; \mathscr{I}(q,n) \;\; \Leftrightarrow \;\; (y_m) \in (\mathbb{F}_q)^{q^n} \text{ satisfies } \Sigma.$$

2. Using Gauss' Algorithm over $\mathbb{F}_q$ transform $M(q,n)$ in an upper triangular matrix. Recall that $M(q,n)$ has maximal rank equal with $\mathrm{wm}(q,n)$.
3. Introduce a tuple $\vec{t}$ of $q^n - \mathrm{wm}(q,n)$ many new parameters and compute the parametric solution of Σ, consisting of linear functions in $\vec{t}$:

$$(Y_m(\vec{t}))_{m \in \mathrm{Mon}(q,n)}.$$

4. Using the equivalence from (1) one has that:

$$\mathscr{I}(q,n) = \left\{ \sum_{m \in \mathrm{Mon}(q,n)} Y_m(\vec{t}) \; m \;\; | \, \vec{t} \in (\mathbb{F}_q)^{q^n - \mathrm{wm}(q,n)} \right\}.$$

   For $i = 1, \ldots, q^n - \mathrm{wm}(q,n)$ set the parameter tuple $\vec{t_i} = (0,0,\ldots,t_i = 1,0,\ldots,0)$ in the general form

$$P_{\vec{t}} = \sum_{m \in \mathrm{Mon}(q,n)} Y_m(\vec{t}) m$$

   and call the result of the substitution $P_i = P_{\vec{t_i}}$. The mapping $\vec{t} \rightsquigarrow P_{\vec{t}}$ is an isomorphism of vector spaces over $\mathbb{F}_q$ because it is a surjective linear mapping between vector spaces of equal dimensions. As an isomorphism, this mapping transports basis to basis, so $\{ P_i \,|\, i = 1, \ldots, q^n - \mathrm{wm}(q,n) \}$ is a basis of the vector space $\mathscr{I}(q,n)$ over $\mathbb{F}_q$.

The fact that this algorithm finds a basis of the ideal of identities of symmetric polynomials follows directly from the theory developed above and from its description. It remains to show only that the

coefficients of the basis polynomials always belong to the base field. This shall be shown in the next Section. Here some examples will be given.

Example. $\mathbb{F}_2[X_1,X_2]$: $\dim_{\mathbb{F}_2}\mathscr{I}(2,2) = 2^2 - \mathrm{wn}(2,2) = 4-3 = 1$. The monomial basis of the vector space $\mathbb{F}_2\{E_1,E_2\}$ is the set $\{1,E_1,E_2,E_1E_2\}$. The matrix $M(2,2)$ is the following:

	1	e_1	e_2	e_1e_2
00	1	0	0	0
01	1	1	0	0
11	1	0	1	0

Linear variables $\{Y_1,Y_2,Y_3,Y_4\}$ are associated to the columns. The linear system of equations over $\mathbb{F}_2$:

$$\begin{aligned} Y_1 &= 0, \\ Y_1+Y_2 &= 0, \\ Y_1+Y_3 &= 0, \end{aligned}$$

has the parametric solution $(Y_1,Y_2,Y_3,Y_4) = (0,0,0,t)$, where $t \in \mathbb{F}_2$ is a parameter. It follows that:

$$\mathscr{I}(2,2) = \{te_1e_2 \,|\, t \in \mathbb{F}_2\} = \{0, e_1e_2\}.$$

The only one nontrivial algebraic relation between elementary symmetric functions is in this case the following:

$$e_1e_2 = 0.$$

We verified in the introduction that $e_1e_2 = 0$ as a function.

Example. $\mathbb{F}_2[X_1,X_2,X_3]$: $\dim_{\mathbb{F}_2}\mathscr{I}(2,3) = 2^3 - \mathrm{wn}(2,3) = 8-4 = 4$. The monomial basis of the vector space $\mathbb{F}_2\{E_1,E_2,E_3\}$ is the set $\{1,E_1,E_2,E_3,E_1E_2,E_1E_3,E_2E_3,E_1E_2E_3\}$. The matrix $M(2,3)$ is the following:

	1	e_1	e_2	e_3	e_1e_2	e_1e_3	e_2e_3	$e_1e_2e_3$
000	1	0	0	0	0	0	0	0
001	1	1	0	0	0	0	0	0
011	1	0	1	0	0	0	0	0
111	1	1	1	1	1	1	1	1

Gauss' Algorithm produces the following matrix:

1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	1	1	1	1

Linear variables $\{Y_1,Y_2,Y_3,Y_4,Y_5,Y_6,Y_7,Y_8\}$ are associated to the columns. The linear system of equations over $\mathbb{F}_2$ has the parametric solution:

$$(Y_1,Y_2,Y_3,Y_4,Y_5,Y_6,Y_7,Y_8) = (0,0,0,t_1+t_2+t_3+t_4,t_1,t_2,t_3,t_4),$$

where $t_1,t_2,t_3,t_4 \in \mathbb{F}_2$ are parameters. It follows that:

$$\mathscr{I}(2,3) = \{(t_1+t_2+t_3+t_4)e_3 + t_1e_1e_2 + t_2e_1e_3 + t_3e_2e_3 + t_4e_1e_2e_3 \,|\, t_1,t_2,t_3,t_4 \in \mathbb{F}_2\}.$$

By replacing the parameter vector (t_1,t_2,t_3,t_4) with the unit vectors $(1,0,0,0)$, ..., $(0,0,0,1)$, following basis of linear independent algebraic relations is found:

$$\begin{aligned} e_3 + e_2e_3 &= 0 \\ e_3 + e_1e_3 &= 0 \\ e_3 + e_1e_2 &= 0 \\ e_3 + e_1e_2e_3 &= 0 \end{aligned}$$

Remark 4.1 In this algorithm, we can skip the line corresponding to the tuple $(0,0,\ldots,0)$ and the column corresponding to the monomial 1. This does not change the result.

Remark 4.2 If $n = 1$, the monomials are $\{1,X,X^2,\ldots,X^{q-1}\}$ and the tuples are the q elements of $\mathbb{F}_q$. It follows that the matrix $M(q,1)$ is nothing but the Vandermonde matrix over the field $\mathbb{F}_q$. The ideal $\mathscr{I}(q,1)$ is always 0.

5 Coefficients in $\mathbb{F}_p$

Theorem 5.1 *Let $\mathbb{F}_q$ be some finite field, $p = \text{char}\,\mathbb{F}_q$ and $n \geq 2$. Then the algorithm presented above finds a basis of $\mathscr{I}(q,n)$ consisting of polynomials with coefficients in the prime field $\mathbb{F}_p$. In particular, such a basis always exists.*

Before proving the Theorem 5.1, we must fix some notations concerning Gauss' Algorithm.

Definition: For $i < j$ we define:

$A(i,j)$ means that the equation i multiplied with an apropriate element is added to equation j. The element is chosen such that the first non-zero coefficient in the equation j becomes 0.

$L(i,j)$ means that equations (lines) i and j are inter-changed.

$C(i,j)$ means that the columns i and j are inter-changed.

Definition: Here is the step number i of the deterministic Gauss' Algorithm. If $a_{i,i} = 0$ and the whole line i consists only of zeros, find the first $j > i$ such that the line j contains non-zero elements, and apply $L(i,j)$. If $a_{i,i}$ continues to be zero, find the first $k > i$ such that $a_{i,k} \neq 0$ and apply $C(i,k)$. Now, for each $r > i$; if the element $a_{r,i} \neq 0$ below $a_{i,i}$, apply $A(i,r)$.

Definition: Let K be some field and S, S' two systems of linear equations over K; both of them consisting of e equations with u unknowns. We say that Gauss' Algorithm works in parallel for systems S and S' if the application of Gauss' Algorithm on the systems leads to the same sequence of operations $O_1,O_2,\ldots,O_s$ in the above notation. (For example, $O_1 = A(1,2)$, . . . , $O_s = L(3,4)$, $O_{s+1} = C(3,5)$, . . . and so on.)

Lemma 5.2 *Let K be a field and S, S' two homogenous systems of linear equations over K, satisfying the following conditions:*

1. *S and S' have both e equations and u unknowns, $k = u - e \geq 0$ and* rank $S = e$.
2. *S' has been obtained from S by some permutation of lines (equations).*
3. *Gauss' Algorithm works in parallel over S and S'.*

In this situation, Gauss' Algorithm independently applied for the systems S and S' computes the same parametrization of the common space of solutions:

$$\forall\, i = 1, \ldots, u \quad Y_i(\vec{t}) = Y'_i(\vec{t}),$$

where $\vec{t} = (t_1, t_2, \ldots, t_k)$ is the tuple of parameters.

Proof: Let $M\vec{Y} = \vec{0}$ be the system S of linear equations as in the statement. Running Gauss' Algorithm forth and back, we find that columns with indexes $i_1, \ldots, i_e$ build a non-singular $e \times e$ minor A. Let $j_1, \ldots j_k$ be the indexes of the left columns. For $s = 1, \ldots, k$, we set $Y_{j_s} = t_s$, where $t_1, \ldots, t_s$ are independent parameters.

The parametric solution is obtained from:

$$A \begin{pmatrix} Y_{i_1} \\ \vdots \\ Y_{i_e} \end{pmatrix} = -\sum_{s=1}^{k} \vec{c}_{j_s} t_s,$$

where $\vec{c}_{j_s}$ are the column of M left outside A. The parametric solution has the form:

$$\begin{pmatrix} Y_{i_1} \\ \vdots \\ Y_{i_e} \end{pmatrix} = \sum_{s=1}^{k} \vec{d}_{j_s} t_s,$$

completed with the k many $Y_{j_s} = t_s$ already done. Here the column vector $\vec{d}_{j_s}$ is the unique solution of the system of linear equations:

$$A \begin{pmatrix} Y_{i_1} \\ \vdots \\ Y_{i_e} \end{pmatrix} = -\vec{c}_{j_s}.$$

Now consider the system S' of linear equations $M'\vec{Y} = \vec{0}$. As Gauss' Algorithm works in parallel for the systems S and S', by running it forth and back, we find the same columns $i_1, \ldots, i_e$ to build a non-singular minor A'. But the matrix M' consists of the same lines as M in a permuted order, and the same permutation is used to get A' from A and $\vec{c'}_{j_s}$ from $\vec{c}_{j_s}$. The parametric solution is again:

$$\begin{pmatrix} Y_{i_1} \\ \vdots \\ Y_{i_e} \end{pmatrix} = \sum_{s=1}^{k} \vec{d}_{j_s} t_s,$$

completed with the k many $Y_{j_s} = t_s$, where the column vector $\vec{d}_{j_s}$ is the unique solution of the system of linear equations:

$$A' \begin{pmatrix} Y_{i_1} \\ \vdots \\ Y_{i_e} \end{pmatrix} = -\vec{c'}_{j_s}.$$

This system is the same system as previous one, up to the order in which the equations are displayed, and we know that permuting the equation in a system with unique solution, does not change this solution. □

Proof of the Theorem 5.1: If $\mathbb{F}_q = \mathbb{F}_p$ there is nothing to prove. Consider some automorphism $\varphi \in \mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$. The fact that φ is a power of Frobenius' Automorphism is not relevant here. Consider the system Σ used in the algorithm given above and the system $\Sigma' = \varphi(\Sigma)$.

Claim 1: Σ' can be obtained from Σ by some permutation of equations. Indeed, the elements of the matrix $\varphi(M(q,n))$ are $\varphi(a(\vec{\iota},m)) = \varphi([\Phi(m)](\vec{\iota})) = [\Phi(m)](\vec{\varphi(\iota)})$. The automorphism φ is a permutation of $\mathbb{F}_q$. Consequently, the line formerly indexed $\vec{\iota}$ shall be found in Σ' at the index obtained by the weakly monotone reordering of the tuple $\varphi(\vec{\iota})$. As the coefficients of the system are values of symmetric functions for this tuple, the order inside the tuple does not matter.

Claim 2: Gauss' Algorithm works in parallel over Σ and Σ'. The coefficients in Σ' are images of corresponding coefficients in Σ by φ. This situation remains true after every computation step done by Gauss Algorithm. In particular, at every step one has in both systems Σ and Σ' the same situation concerning coefficients (matrix entries) which are zero or not. So after every step, the same decision concerning the next step shall be taken: an $A(i,j)$ or a $C(i,j)$.

So all conditions requested by Lemma 5.2 are satisfied, and Gauss' Algorithm computes the same parametrization $(Y_i(\vec{t}))$ for both systems Σ and $\varphi(\Sigma)$. So for all i, $Y_i(\vec{t}) = \varphi(Y_i(\vec{t}))$; and this takes place for all automorphisms $\varphi \in \mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$. It follows that the linear functions $Y_i(\vec{t})$ have coefficients in $\mathbb{F}_p$, and the same is true for the basis of $\mathscr{I}(q,n)$ found by the algorithm. □

6 Further examples

This algorithm has been implemented by the author using the language C# on Visual Studio, and ran for the fields $\mathbb{F}_q$ with $q \in \{2,3,4,5,7,8,9,11,16,25,27,49,81\}$ and values of $n \leq 5$. The implementation uses the fact that the elementary symmetric polynomials in $x_1,\ldots,x_n$ can be computed all at a time in quadratic time and linear space. We show here only some examples.

$\mathbb{F}_3[x_1,x_2]$. $\dim_{\mathbb{F}_3}\mathscr{I}(3,2) = 3^2 - \mathrm{wn}(3,2) = 9-6 = 3$. Following basis has been found:

$$\begin{aligned} 2e_1e_2 + e_1e_2^2 &= 0 \\ e_2 + e_2^2 + e_1^2e_2 &= 0 \\ e_2 + e_2^2 + e_1^2e_2^2 &= 0 \end{aligned}$$

$\mathbb{F}_3[x_1,x_2,x_3]$. $\dim_{\mathbb{F}_3}\mathscr{I}(3,3) = 3^3 - \mathrm{wn}(3,3) = 27-10 = 17$. Following basis has been found:

$$\begin{aligned}
2e_2e_3^2+e_1e_3 &= 0\\
2e_2e_3+e_1e_3^2 &= 0\\
e_2e_3^2+e_2^2e_3^2 &= 0\\
e_2e_3^2+e_1e_2e_3 &= 0\\
e_2e_3+e_1e_2e_3^2 &= 0\\
e_2e_3+2e_1e_2+e_1e_2^2 &= 0\\
2e_2e_3^2+e_1e_2^2e_3 &= 0\\
2e_2e_3+e_1e_2^2e_3^2 &= 0\\
e_2e_3+e_2^2e_3 &= 0
\end{aligned}$$

$$\begin{aligned}
e_2e_3+e_1^2e_3 &= 0\\
e_2e_3^2+e_1^2e_3^2 &= 0\\
e_2+2e_2e_3^2+e_2^2+e_1^2e_2 &= 0\\
2e_2e_3+e_1^2e_2e_3 &= 0\\
2e_2e_3^2+e_1^2e_2e_3^2 &= 0\\
e_2+e_2e_3^2+e_2^2+e_1^2e_2^2 &= 0\\
e_2e_3+e_1^2e_2^2e_3 &= 0\\
e_2e_3^2+e_1^2e_2^2e_3^2 &= 0
\end{aligned}$$

$\mathbb{F}_4[x_1,x_2]$: The field $\mathbb{F}_4$ has been realized using the polynomial X^2+X+1 over $\mathbb{F}_2$. $\dim_{\mathbb{F}_4}\mathscr{I}(4,2)=4^2-\mathrm{wn}(4,2)=16-10=6$. The found basis has coefficients in $\mathbb{F}_2$:

$$\begin{aligned}
e_1e_2+e_1^2e_2^2 &= 0\\
e_1e_2^2+e_1^2e_2^3 &= 0\\
e_1e_2^3+e_1^2e_2 &= 0\\
e_1e_2^2+e_1^3e_2 &= 0\\
e_1e_2^3+e_1^3e_2^2 &= 0\\
e_1e_2+e_1^3e_2^3 &= 0
\end{aligned}$$

$\mathbb{F}_5[x_1,\ldots,x_5]$: $\dim_{\mathbb{F}_5}\mathscr{I}(5,5)=5^5-\mathrm{wn}(5,5)=3125-126=2999$. The first identity found was:

$$4e_4^2e_5+e_4^4e_5+3e_3e_4e_5^2+4e_3e_4^3e_5^2+3e_3e_4^4e_5^2+4e_3^2e_4e_5^3+e_3^2e_4^4e_5^3=0.$$

$\mathbb{F}_7[x_1,\ldots,x_4]$: $\dim_{\mathbb{F}_7}\mathscr{I}(7,4)=7^4-\mathrm{wn}(7,4)=2401-210=2191$. The first identity found was:

$$\begin{aligned}
&e_4+6e_4^4+6e_3^6e_4+e_3^6e_4^4+5e_2e_4^2+2e_2e_4^5+5e_2e_3^2e_4^2+2e_2e_3^2e_4^5+5e_2e_3^4e_4^2+\\
&+2e_2e_3^4e_4^5+6e_2^2e_4^3+e_2^2e_4^6+6e_2^2e_3^2e_4^3+e_2^2e_3^2e_4^6+6e_2^2e_3^4e_4^3+e_2^2e_3^4e_4^6=0.
\end{aligned}$$

References

[1] William Feller (1968): *Introduction to Probability Theory and Its Applications, Vol. 1*. Wiley Series in Probability and Statistics.

[2] Martin Kreuzer & Lorenzo Robbiano (2016): *Computational Linear Algebra and Commutative Algebra*. Springer, doi:10.1007/978-3-319-43601-2.

[3] I. G. Macdonald (1995): *Symmetric functions and Hall polynomials*. Oxford Science Publications, Clarendon Press.

[4] A. Okounkov (1998): *On Newton interpolation of symmetric functions: A characterization of interpolation Macdonald polynomials*. Advances in Applied Mathematics 20(4), pp. 395–428, doi:10.1006/AAMA.1998.0590.

[5] Bernd Sturmfels (1993): *Algorithms in invariant theory*. Springer Verlag, Wien. doi:10.1007/978-3-7091-4368-1

[6] Bartel L. van der Waerden (1971): *Moderne Algebra I*. Springer Verlag, Berlin.

Identifying Vulnerabilities in Smart Contracts using Interval Analysis

Ştefan-Claudiu Susan
Alexandru Ioan Cuza University of Iași,
Department of Computer Science
Iași, România
claudiu_susan@yahoo.com

Andrei Arusoaie
Alexandru Ioan Cuza University of Iași,
Department of Computer Science
Iași, România
andrei.arusoaie@uaic.ro

This paper serves as a progress report on our research, specifically focusing on utilizing interval analysis, an existing static analysis method, for detecting vulnerabilities in smart contracts. We present a selection of motivating examples featuring vulnerable smart contracts and share the results from our experiments conducted with various existing detection tools. Our findings reveal that these tools were unable to detect the vulnerabilities in our examples. To enhance detection capabilities, we implement interval analysis on top of Slither [3], an existing detection tool, and demonstrate its effectiveness in identifying certain vulnerabilities that other tools fail to detect.

1 Introduction

The term "smart contract" was originally used to describe automated legal contracts, whose content cannot be negotiated or changed. Nowadays, the term is most commonly known as programs that are executed by special nodes in a decentralised network or a blockchain. Indeed, the blockchain technology captures the initial meaning of the term: contracts are encoded as an immutable piece of code, and the terms of the contract are predetermined and automatically enforced by the contract itself.

This immutability property also implies more effort on the contract developers side: they have to be very careful about what gets deployed because that code is (1) public and anyone can see it and (2) it cannot be changed/updated as an ordinary program. Ethereum[1] is a very popular blockchain platform which has been affected by the most significant attacks based on vulnerabilities in the deployed code. For instance, "The DAO attack" [2] was based on the fact that a smart contract could be interrupted in the middle of its execution and then called again. This is known as the *reentrancy* vulnerability. An attacker noticed that in a withdrawal function of the smart contract, the transfer of digital assets was performed before updating the balance (i.e., decrementing the balance with the withdrawn amount) of a contract party. The attacker first deposited cryptocurrency into the smart contract. Then, by creating a scenario where the withdrawal function called itself just before updating its own balance, the attacker managed to drain the funds of the smart contract as long as its balance exceeded the amount withdrawn.

Such mistakes are unfortunate and researchers and practitioners started to propose methods and tools for detecting them. For example, Slither [3] is an easy to use static analysis tool for smart contracts written in Solidity [3]; Mythril[4] is a security analysis tool for EVM bytecode based on symbolic execution;

[1] Vitalik Buterin, *A Next-Generation Smart Contract and Decentralized Application Platform*, 2014, URL: https://ethereum.org/en/whitepaper/

[2] David Siegel, 2016: Understanding The DAO Attack. URL:https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/

[3] Solidity, version 0.8.20: https://docs.soliditylang.org/en/v0.8.20/control-structures.html

[4] Mythril docs: https://mythril-classic.readthedocs.io/en/develop/

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 144–151, doi:10.4204/EPTCS.389.12

Solhint[5] is a linter for Solidity code. The list of tools is long and it was explored in various papers (e.g., [5, 7, 6, 1, 4]). These tools are indeed very useful, but in the same time they are not perfect and they can fail to detect problematic situations in smart contracts code.

In this paper we provide several examples of smart contracts which contain vulnerabilities and we find that vulnerability detection tools are not as precise as we expect, and they fail to detect vulnerabilities in our examples. We attempt to enhance one of them (Slither) with an existing static analysis method called interval analysis. This method allows us to better approximate the values interval for each program variable. Based on the experiments that we performed, interval analysis proves to be very useful in detecting problematic situations in smart contracts. For example, integer division in Solidity ignores the reminder. In a situation where an amount of cryptocurrency must be divided and transferred to a number of recipients, a division where the remainder is ignored could lead to funds that remain locked in the smart contract. Another example is related to uninitialised variables: such variables are initialised with default values and it may be the case that the default value is not suitable for the purpose of that variable. By keeping track of all the possible values for each program variable, interval analysis allows us to signal such situations in smart contracts.

Summary of contributions.

1. We provide several examples of vulnerable smart contracts, in which the vulnerabilities prove to be challenging to detect using state-of-the-art detection tools.
2. We implement an existing analysis technique called interval analysis on top of Slither.
3. We evaluate our implementation.

Paper organisation. In Section 2 we present several examples of smart contract vulnerabilities. In Section 3 we show how state-of-the-art tools behave on these examples. The interval analysis technique is presented in Section 4 together with our implementation. We conclude in Section 5.

2 Vulnerabilities in Smart Contracts

This section contains several examples of smart contracts vulnerabilities written in Solidity. These were selected from a larger taxonomy [6]. The whole classification includes 55 vulnerabilities split among 10 categories. Both literature and existing community taxonomies were taken into account when selecting these defects. We selected these vulnerabilities because state of the art tools are not able to detect most of them and could be detectable using interval analysis.

2.1 Tautologies or Contradictions in `assert` or `require` Statements

The Solidity statements `assert` and `require` are typically used to validate boolean conditions. According to the Solidity documentation[6], `assert` is meant for checking internal errors, while `require` should be used to test conditions that cannot be determined until runtime. Both statements throw exceptions and revert the corresponding transactions. In their intended use, the conditions in `assert` should never be false as it signals contract level errors while the conditions in `require` can be false as they signal input errors. No matter what level of error a statement specifies, it is an issue if the conditions that they contain are tautologies or contradictions. These make the statement useless in the case of tautologies and make the transaction impossible to complete in the case of contradictions as illustrated by the following code:

[5]Solhint official website: `https://protofire.github.io/solhint/`

[6]Solidity docs: `https://docs.soliditylang.org/en/v0.8.20/control-structures.html`

```
1:  function notGonnaExecute(uint parameter) external pure returns(uint)
2:  {
3:      require(parameter<0); // uint cannot be < 0
4:      return parameter;
5:  }
```

```
1:  function uselessAssertUint(uint parameter) external pure returns(uint)
2:  {
3:      require(parameter>=0); // uint is always >= 0
4:      return parameter;
5:  }
```

2.2 Division by Zero

This is a classic arithmetic issue that is common among most programming languages. The Solidity compiler does not allow direct division by zero. However, the compiler cannot detect situations when the denominator could evaluate to zero. The following code snippet contains an example. The length of the recipients array is not checked before computing the amount that should be sent to each recipient (line 4):

```
1:  function split(address[] calldata recipients) external payable
2:  {
3:      require(msg.value > 0,"Please provide currency to be split among recipients");
4:      uint amount = msg.value / recipients.length; // problem here if length is 0
5:      for(uint index = 0; index < recipients.length; index++)
6:      {
7:          (bool success,) = payable(recipients[index]).callvalue:amount("");
8:          require(success,"Could not send ether to recipient");
9:      }
10: }
```

2.3 Integer Division Remainder

This is another arithmetic issue that is common among many programming languages. Solidity performs integer division which means that the result of the division operation is truncated. This could lead to situations where ignoring the remainder of the division could lead to logic errors. The snippet below contains an example: if the provided amount does not exactly divide by the number of recipients then that amount of cryptocurrency could remain locked in the contract.

```
1:  function split(address[] calldata recipients) external payable
2:  {
3:       require(recipients.length > 0,"Empty recipients list");
4:       uint amountPerRecipient = msg.value / recipients.length; // remainder ???
5:       require(amountPerRecipient > 0,"Amount must be positive");
6:       for(uint index = 0; index < recipients.length; index++)
7:       {
8:           payable(recipients[index]).transfer(amountPerRecipient);
9:       }
10:  }
```

2.4 Uninitialised Variable

Uninitialized variables could lead to logical errors or exceptions. If a variable is not initialised, there is a great chance that the default value assigned to the variable (according to its type) is not suitable for

the purpose of that variable. The following code contains an access modifier which relies on the `owner` state variable. The variable is `private`, and thus, it cannot be accessed or assigned outside the contract. Also, there is no explicit initialisation of `owner` within a constructor. This makes the variable stuck to the default value, and thus, all the functions marked with the `onlyOwner` modifier cannot be executed.

```
1:    address private owner;
2:
3:    modifier onlyOwner() {
4:        require(msg.sender == owner, "Only the owner of the contract has access");
5:        _;
6:    }
```

2.5 User Input Validation

Parameter validation or "sanitisation" is a process that must be implemented at the beginning of every method. This ensures that the method will always execute as expected. End users should not be trusted to always provide valid parameters. If validation is missing and the end user is unaware, or worse, malicious, it could cause critical errors that produce unexpected results or halt contract execution all together. The following example contains a getter method for an internal array. The user can provide an index that is not validated, thus having the possibility of going out of bounds.

```
1:    uint256[] private _array= [10, 20, 30, 40, 50];
2:
3:    function getArrayElement(uint256 index) external view returns (uint256)
4:    {
5:        return _array[index];
6:    }
```

2.6 Unmatched Type

In Solidity, enums are stored as unsigned integers. Thus, they can be compared and assigned with variables of type `uint`. Situations like these can become tricky since the value domain of an enum is likely to be much smaller than the value domain of unsigned integers. If a variable with a greater value than the range of the enum is assigned to an enum variable, than the transaction will be reverted. While it is true that reverting the transaction is considered safe, such situations signal a faulty logic in the contract code and it is preferable to be avoided.

```
1:  contract UnmatchedType {
2:      enum Options { Candidate1, Candidate2, Candidate3 }
3:      mapping(address => Options) private _votes;
4:      mapping(Options => uint) private _votesCount;
5:      function vote(uint option) external {
6:          _votes[msg.sender] = Options(option);
7:          _votesCount[Options(option)]++;
8:      }
9:      function getStatisticsForOption(uint option) external view returns(uint) {
10:         return _votesCount[Options(option)];
11:     }
12: }
```

3 Detecting Vulnerabilities Using Dedicated Analysis Instruments

This section briefly presents the results of some experiments that we performed. Basically, we used a few tools for analysing smart contracts in order to check how they behave on our examples presented in

Examples	Slither	Solhint	Remix	Mythril
Tautologies/Contradictions	✓	✗	✗	✗
Division by zero	✗	✗	✗	✗
Integer division	✗	✗	✗	✗
Uninitialised variable	✓	✗	✗	✗
User input validation	✗	✗	✗	✗
Unmatched type	✗	✗	✗	✗

Table 1: A summary of the evaluation of the tools when executed on our examples (Section 2).

Section 2. The tools that we selected are presented below. It is worth noting that we picked tools which implement different techniques (e.g., static analysis, symbolic execution, linter). For a tool to be eligible for our study, it has to be open source, active and compatible with the latest version of Solidity.

Slither is a static analysis tool written in Python. It provides vulnerability detection and code optimization advice. It features many detectors that target different issues. Its analysis runtime is very low compared to the other tools. It analyses Solidity code by transforming the EVM bytecode into an intermediary representation called SlithIR. Being an open source project, it allows anyone to contribute and improve it, being the foundation for our implementation (discussed later in Section 4). Slither was able to detect uninitialized variables as well as trivial tautologies and contradictions in our examples.

Solhint is a linter for Solidity code. An open source project, it is able to detect possible vulnerabilities, optimization opportunities and abidance to style conventions. The tool also features a customisable set of detection rules that can be employed, along with predefined configurations. The user can define its own configurations and decide which issue wants to target. Unfortunately, Solhint was not able to detect any of the issues in our examples.

Remix[7] also features a static analysis plugin. We were unable to find any information about the analysis process performed by this tool. Moreover, it was unable to detect any problems in our examples.

Mythril is a tool that leverages symbolic execution to simulate multiple runs of a contract's methods. It has a fairly long runtime compared to the others. We even encountered executions that took more than a few hours. Mythril was unable to detect any of the issues presented above.

In Table 1, we present a summary of the results that we obtained. The results indicate that nearly all tools fail to detect the vulnerabilities in our examples. This does not mean that these tools are not useful or very bad at signaling issues in smart contract code. The way we interpret these results is that these tools need to be enhanced with more powerful techniques that could increase their detection capabilities.

4 Interval Analysis for Vulnerability Detection

4.1 Interval Analysis

Interval Analysis [8] is a static analysis technique that approximates the values interval for every variable in a program for a certain instruction. The technique is not limited to predicting the values interval of a variable, it can also be used to predict certain properties that can be derived from the value of the variable. For instance, instead of working with integer intervals, an analysis can target the parity of variables and work only with 2-valued intervals (even, odd).

[7]Remix Docs: `https://remix-ide.readthedocs.io/en/latest/`

Statements	`option`	`_votes[msg.sender]`
1	[0, max]	[0, max]
2	[0, max]	[0, 2]
End	[0, max]	[0, 2]

Table 2: Interval analysis for the `vote` function.

We present interval analysis via the `Unmatched Type` example from Section 2.6. Moreover, we show how interval analysis can help us detect a problem in this example, more precisely in the `vote` function:

```
5:      function vote(uint option) external {
6:          _votes[msg.sender] = Options(option); //Statement 1
7:          _votesCount[Options(option)]++;       //Statement 2
8:      }
```

The function registers the vote of an user and increases the total vote count for its option. The problem is at line 2: the input is of type `uint` and it could easily be outside the values range [0,1,2] of the enum.

Interval analysis provides an approximation of the values interval for every program variable at each program location. In Table 2 we show how these intervals are computed for our example. Each line of the table presents the intervals for the program variables (displayed in columns) *before* the execution of each statement in the first column. For example, before the execution of `Statement 1`, we do not have any information about the `option` variable, so its range of values will correspond to the values domain for `uint`. For `_votes[msg.sender]`, the value interval changes before `Statement 2` in case of normal execution (otherwise, the transaction is reverted) to [0,2], that is, the only possible range for `Option`. Interval analysis performs this calculation using the Worklist Algorithm, an algorithm which traverses the program control flow graph, and updates the intervals for these variables until a fixpoint is reached. This algorithm is shown in Section 4.2.

Recall that the problem we are trying to detect using interval analysis is a mismatch of domains between the variable assigned and the variable whose value is assigned. Since interval analysis computes the interval for `option` and `_votes[msg.sender]`, a close inspection of the difference between the intervals is sufficient to reveal the problem. A `require` statement that checks upfront the values for the `option` parameter would solve the problem. Also, our detection technique would not signal an issue.

4.2 An Implementation of Interval Analysis on Top of Slither

We built our implementation using Python modules provided by Slither. These are the same modules that are used internally by Slither for its own detectors. During execution, Slither fills some of its internal data structures with useful information, such as contract CFG (control flow graph), an intermediary SSA (single statement assignment) representation of the code, and information about each variable (e.g., type, scope and name). We use the information in these data structures to implement interval analysis.

The Worklist Algorithm shown in Figure 1 works by processing every edge in the contract CFG. These edges are added into a list (the "worklist"). It is an iterative algorithm that processes existing elements until the list is empty. When new information is added to the current state, new edges are also added to the worklist. The algorithm stops when no more new information can be discovered.

We implemented a modular Worklist algorithm. Essential information such as extreme labels[8], order

[8]The program nodes where the analysis begins.

$Values \leftarrow InitialValues(G)$
$Worklist \leftarrow RootSet(G)$
while $HasMoreNodes(Worklist)$ **do**
  $n_i \leftarrow NextNode(Worklist)$
  while $HasMoreEdges(n_i)$ **do**
    $e \leftarrow nextEdge(n_i)$
    $t \leftarrow type(e)$
    $n_j \leftarrow farNode(e, n_i)$
    $v'_j \leftarrow F(t, v_i, v_j)$
    if $MonotonicChange(v'_j, v_j)$ **then**
      $v_j \leftarrow v'_j$
      $AddToWorklist(Worklist, n_j)$
    end if
  end while
end while
return V

Figure 1: The Worklist Algorithm.

function[9] and flow function[10] are all provided as parameters to the Worklist algorithm. This allow us to perform multiple types of analysis using the same base implementation. Our implementation leverages the CFG provided by Slither to split the code of a function into multiple parts. Each node is then split even further into SlithIR SSA [2]lines that are analyzed individually. Along with basic types such as `uint` and `bool`, our implementation is able to model complex types such as arrays, mappings and structs.

We defined our own data type to encapsulate information about a variable such as type, scope, name and, most importantly, values interval. The program state is represented as a dictionary having variable names as keys and an object of our own defined type as values. Complex types are defined as recursive dictionaries, for example, the interval for a struct is modeled as a dictionary containing intervals for each of its fields or even other dictionaries if the structs are nested.

Current status. Our implementation is now able to successfully analyze programs containing assignments and arithmetic expressions for both elementary and complex types. It takes into consideration state variables, function parameters, and local variables. We are now capable of detecting issues such as:

- Arithmetic issues including `Division By Zero` and `Integer Division Remainder`;
- Issues related to variables initialization;
- Issues related to parameter validation.

5 Conclusions

In this paper we identified some vulnerabilities that are not handled by state of the art tools for smart contract analysis. These vulnerabilities vary in their severity, but no matter the impact, defects and potential errors should be identified as soon as possible. We attempt to improve Slither with a more powerful technique called interval analysis. We explain why this technique is a good fit for detecting these issues and how it could detect them. We built a custom interval analysis on top of Slither, leveraging

[9] A function that receives two elements of the same domain and determines the greater one.
[10] A function determining the edges in the flow graph or the reverse of those edges depending on the type of analysis.

the information that Slither already provides about a contract, its attributes, methods, method parameters, program flow and many more. Currently, our implementation detects vulnerabilities that other tools miss.

5.1 Future Work

We are now handling only on a subset of expressions in the Solidity programming language, which covers expressions including integers, booleans, arrays, structures, and mappings. However, more elaborate work needs to be done to tackle addresses and operations over addresses, more complex loops or conditional statements, etc. Right now, our code can be executed on every smart contract written in Solidity, but it will perform interval analysis only for the subset that we cover.

Intraprocedural analysis would significantly improve the precision of our analysis. An example of a vulnerability that we are not yet able to detect is *Short Address*. This could be detected by monitoring the length attribute of the payload. Another example is *Tautologies and Contradictions in Assert or Require Statements*: it could be detected by approximating the result of the boolean expression and checking if the interval contains only one value: *true* for tautologies and *false* for contradictions.

Being able to handle more complex conditional statements and loops would also be of great help in obtaining a more accurate monitoring of the program state by interpreting the semantics of boolean expressions. Once we identify multiple possible states based on conditional branches, we can leverage unifying techniques such as *Trace Partitioning*. Additionally, monitoring implicit state variables that are contract-level or function-level, like `balance` or `msg.sender`, would be beneficial in identifying balance-related issues and user interaction problems.

References

[1] N. Atzei, M. Bartoletti & T. Cimoli (2017): *A Survey of Attacks on Ethereum Smart Contracts (SoK)*. In M. Maffei & M. Ryan, editors: *Principles of Security and Trust*, Springer, Berlin, Heidelberg, pp. 164–186, doi:10.1007/978-3-662-54455-6_8.

[2] R. Cytron, A. Lowry & F.K. Zadeck (1986): *Code motion of control structures in high-level languages*. In: *the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 70–85, doi:10.1145/512644.512651.

[3] J. Feist, G. Grieco & A. Groce (2019): *Slither: a static analysis framework for smart contracts*. In: *2019 IEEE/ACM 2nd Intl Workshop on Emerging Trends in Software Engineering for Blockchain*, IEEE, pp. 8–15, doi:10.1109/WETSEB.2019.00008.

[4] I. Grishchenko, M. Maffei & C. Schneidewind (2018): *A Semantic Framework for the Security Analysis of Ethereum Smart Contracts*. In L. Bauer & R. Küsters, editors: *Principles of Security and Trust*, Springer, pp. 243–269, doi:10.1007/978-3-319-89722-6_10.

[5] A. Mense & M. Flatscher (2018): *Security Vulnerabilities in Ethereum Smart Contracts*. In: *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications and Services*, iiWAS2018, Association for Computing Machinery, New York, NY, USA, p. 375–380, doi:10.1145/3282373.3282419.

[6] H. Rameder, M. di Angelo & G. Salzer (2022): *Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum*. *Frontiers in Blockchain* 5, doi:10.3389/fbloc.2022.814977.

[7] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu & Z. Li (2021): *A Survey of Smart Contract Formal Specification and Verification*. *ACM Comput. Surv.* 54(7), doi:10.1145/3464421.

[8] Y. Wang, Y. Gong, J. Chen, Q. Xiao & Z. Yang (2008): *An Application of Interval Analysis in Software Static Analysis*. In: *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, 2, pp. 367–372, doi:10.1109/EUC.2008.60.

Asynchronous Muddy Children Puzzle
(work in progress)

Dafina Trufaș Ioan Teodorescu Denisa Diaconescu Traian Șerbănuță
University of Bucharest
Runtime Verification, Inc.

Vlad Zamfir
Independent Researcher

In this work-in-progress paper we explore using the recently introduced VLSM formalism to define and reason about the dynamics of agent-based systems. To this aim we use VLSMs to formally present several possible approaches to modeling the interactions in the Muddy Children Puzzle as protocols that reach consensus asynchronously.

1 Introduction

Formally modeling and reasoning about distributed systems with faults is a challenging task [2]. We recently proposed the theory of *Validating Labeled State transition and Message production systems (VLSMs)* [9] as a general approach to modeling and verifying distributed protocols executing in the presence of faults.

The theory of VLSMs has its roots in the work on verification of the CBC Casper protocol [10, 5] and follows the correct-by-construction methodology for design and development. Even though the theory of VLSMs was primarily designed for applications in faulty distributed systems, and in blockchains in particular, the framework is general and flexible enough to capture various types of problems in distributed systems. As an illustration, in this paper we show how we can use VLSMs to model and solve (an asynchronous variant of) the epistemic Muddy Children Puzzle [1].

2 Classical Muddy Children Puzzle

Let us begin by recalling the statement of the Muddy Children Puzzle and a classical epistemic logic approach to solving it [1].

There are n children playing together. It happens during their play that k of the children get mud on their foreheads. Their father comes and says: "At least one of you has mud on your forehead." (if $k > 1$, thus expresses a fact known to each of them before he spoke). The father then asks the following question, repeatedly: "Does any of you know whether you have mud on your own forehead?".

The initial assumptions are expressed in terms of common knowledge. Hence, we shall assume that it is common knowledge that the father is truthful, that all the children can and do hear the father, that all the

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 152–166, doi:10.4204/EPTCS.389.13

children can and do see which of the other children besides themselves have muddy foreheads and that all the children are truthful and intelligent enough to make all the necessary deductions during the game.

The solution Let's consider first the situation before the father speaks. We model the problem by a Kripke structure $M = (S, \models, \mathscr{K}_1, \ldots, \mathscr{K}_n)$ over Φ, where:

- $\Phi = \{p_i, \ldots, p_n, p\}$ (p_i = "child i has a muddy forehead" and p = "at least one child has a muddy forehead"). Note that p could be obtained as the disjunction of all p_i's; however, for simplicity one can consider it a primitive (albeit non-atomic) predicate.
- S is a set of 2^n states (corresponding to all the possible configurations of clean and muddy children, represented as binary tuples)
- $(M, (x_1, \ldots, x_n)) \models p_i$ iff $x_i = 1$
  $(M, (x_1, \ldots, x_n)) \models p$ iff $x_i = 1$ for some i
- $(s,t) \in \mathscr{K}_i$ if s and t are identical in all components except eventually the i^{th} one.

It's crucial to remark that, in the absence of the father's initial announcement, the fact that "there is at least one muddy child" is not common knowledge and the state of knowledge never changes, no matter how many rounds we take into account. Indeed, after the first question, all the children will certainly answer "No", since they all consider possible the situation in which they themselves do not have mud on their forehead. No information is gained from this round and the situation remains the same after each of the following ones, because each child considers possible a state in which they are clean.

Now let's analyze how the epistemic context changes after the father speaks: as mentioned above, the common knowledge is now larger (even though in the case with $k \geq 2$ muddy children, p was already common knowledge), because of the *public* nature of the announcement. Let's consider how the children reason after all of them answered "No" in the first round: it is obvious that all of them eliminate the states containing one muddy child, since the others could not have all answered "No" otherwise. Continuing inductively, we obtain that after k rounds in which all the children answer "No", we can eliminate from the problem graph all the nodes corresponding to states with at most k muddy children. An immediate consequence of this is that after $k-1$ rounds, it becomes common knowledge that there are at least k muddy children. Hence, the muddy children, who each only see $k-1$ muddy children will conclude that they are muddy and answer "Yes".

There are multiple formalizations of this puzzle in the literature [8, 4, 6, 7, 3]; indeed it seems that each new formalism reasoning about knowledge includes a modeling of this puzzle as a basic example of the expressiveness of the formalism.

3 VLSMs – basic notions

We give a high-level presentation of the theory of VLSMs. More details can be found in [9].

Definition 1 (VLSM)**.** *A Validating Labeled State transition and Message production system (*VLSM*, for short) is a structure of the form* $\mathscr{V} = (L, S, S_0, M, M_0, \tau, \beta)$*, where* L *is a set of labels,* $(S_0 \subseteq)$ S *is a non-empty set of (initial) states,* $(M_0 \subseteq)$ M *is a set of (initial) messages,* $\tau : L \times S \times M? \to S \times M?$ *is a*

transition function which takes as arguments a label, a state, and possibly a message, and outputs a state and possibly a message, while β is a validity constraint on the inputs of the transition function.[1]

The transition function in a VLSM is total; however, it indirectly becomes a partial function since the validity constraint can filter out some of its inputs. The set of labels in a VLSM can be used to model non-determinism: it is possible to have multiple parallel transitions from a state using the same input message, each with its own label.

Validity. A transition is called *constrained* if the validity constraint holds on its input. We denote a constrained transition $\tau(l,s,m) = (s',m')$ by $s \xrightarrow[m \to m']{l} s'$. A *constrained trace* is a sequence of constrained transitions that starts in an initial state. A *valid trace* is inductively defined as a constrained trace in which the input of each transition can be emitted from a valid trace. A state is *constrained/valid* if there is a constrained/valid trace leading to it. Similarly, a message is *constrained/valid* if it is produced on a constrained/valid trace (we also consider the no-message to be valid). A transition is called *valid* if it is a constrained transition that uses only valid states and messages; thus a valid trace is a sequence of valid transitions starting in an initial state.

Equivocation. In the literature concerning fault-tolerance in distributed systems, equivocation models the fact that certain agents can claim to be in different states to different parties. VLSMs allow modeling such behavior, by specifying that in a valid trace the valid input messages can be produced on different (though still valid) traces.

Composition. A single VLSM can represent the local point of view of a component in a distributed system. We can obtain the global point of view by composing multiple VLSMs and lifting the local validity constraint of each component. Designers of systems can impose additional restrictions, which are stronger than the ones that can be specified locally on individual components because they can be stated in terms of the global composite state. We capture this phenomenon by the notion of *composition constraint*.

Definition 2 (Composition)**.** *Let $\{\mathscr{V}_i\}_{i=1}^n$ be an indexed set of VLSMs over the same set of messages M. The constrained composition under a composition constraint φ is the VLSM $\mathscr{V} = \left(\sum_{i=1}^n \mathscr{V}_i\right)\Big|_\varphi = (L,S,S_0,M,M_0,\tau,\beta \wedge \varphi)$ where L is the disjoint union of labels, the (initial) states are the product of (initial) states of the components, the transition function τ and the constraint predicate β are defined component-wise, while the composition constraint $\varphi \subseteq L \times S \times M?$ is an additional predicate that filters the inputs for the transition function.*

When a composition constraint is trivial, i.e., it is the set $L \times S \times M?$, we refer to the composition as the *free composition* and drop the subscript φ in the notation.

No Equivocation constraint. As mentioned above, VLSMs implicitly allow equivocation. Nevertheless, a truthful behavior of the agents in a composition can still be enforced by means of a *no equivocation* constraint, which does not allow receiving in a (composite) state a message from a component if that component could not have emitted the message in a trace leading to the current state.

[1] For any set M of messages, let $M? = M \cup \{\boxtimes\}$ be the extension of M with $\boxtimes$, where $\boxtimes$ stands for *no-message*.

4 Asynchronous Muddy Children Puzzle as a VLSM — take 1

We would like to model the Muddy Children Puzzle as a collective effort of a group of agents (the children) to reach a common goal (knowing whether they are muddy or not) by exchanging messages which reveal as little as possible of their personal (initial) knowledge about the problem (a.k.a., which children they see as muddy). To this aim, there are several characteristics which will be common to both our presented models:

- Each child needs to know (at all times) which of the other children are muddy (the child's initial knowledge);
- Every message needs to have a sender;
- Every message has to report on the epistemic status of its sender at the time the message was sent;
- Decisions are final: once a child has decided upon their own status (clean/muddy), they will not receive additional messages, as these would not bring new knowledge.

In our models we choose to index the children with natural numbers and to represent a child's initial knowledge as a set of indices of the muddy children they see (note that the index of the child cannot appear in this set). We will also use these indices to identify the sender of a message. For the epistemic status, we will use three possible values:

- u stands for *"child doesn't known their status"*;
- m stands for *"child knows they have mud on their forehead"*;
- c stands for *"child knows they don't have mud on their forehead"*.

For our first modeling attempt, we let the children maintain and communicate a number reflecting "the round number" (from the original solution) at which they perceive themselves to be.

Let us start formalizing this setting using VLSMs. We represent each child as a VLSM of the form $\mathscr{C}_i = (L_i, S_i, S_{0,i}, M, M_{0,i}, \tau_i, \beta_i)$. The states of $\mathscr{C}_i$ are either initial states of the form $\langle Obs \rangle$ where Obs represents the children seen as muddy by child i, or running states of the form $\langle Obs, r, status \rangle$, where additionally r is the round perceived by child i, and $status$ represents the epistemic status of child i.

- $L_i = \{init, emit, receive\}$ reflects the three types of transitions (*init*ialization, corresponding to the father's announcement and *emit*ting/*receiv*ing messages)
- $M_i = \{\langle j, r, status \rangle \mid j \in \{1, \ldots, n\},\ r \in \mathbb{N},\ status \in \{u, m, c\}\}$ — messages also communicate the round number
- $M_{0,i} = \emptyset$
- $S_i = \{\langle Obs \rangle \mid Obs \subseteq \{1, \ldots, n\}\} \cup \{\langle Obs, s, r \rangle \mid Obs \subseteq \{1, \ldots, n\},\ s \in \{u, m, c\},\ r \in \mathbb{N}\}$
- $S_{0,i} = \{\langle Obs \rangle \mid Obs \subseteq \{1, \ldots, n\}\}$,

The invariant we would like to maintain is that a child at round r knows (from the messages exchanged with its peers) that there are more than r muddy children.

Assuming that such an invariant holds for all accessible states, then when a child (say i) sends a message containing their round number (say k) and the fact that they still cannot determine their status, a child receiving such a message (say j) can derive from it that i knows that there are more than k muddy children except themselves (since otherwise i would know their status). Moreover, since the receiving child knows whether the sender is muddy or not, if j sees i as muddy, j can infer that there are actually more than $k+1$ muddy children. If the current round of j is smaller than the number inferred (either k, if i is clean, or $k+1$, if i is muddy), then j can update their current round to that number.

If that happened, say r is the new current round number, if r is less than the number of muddy children that j sees, then the information provided by r is not yet useful enough to draw a conclusion about the status, so the status will stay unknown. However, if r ever becomes equal to the number of muddy children that j sees, then the child knows that there are more than r muddy children, and since they can only see r muddy children, they will conclude they have to be muddy.

An interesting fact holds for clean children. Note first that they see all the children who are muddy (say N), so for them the number of muddy children they see is larger by 1 than the number of muddy children seen by any muddy child. Hence, they cannot infer their status using the reasoning above, because for that they would have to receive a message from a muddy child, say i with an unknown status at round $N-1$, but that is precisely the number of muddy children that child i sees, so at that round child i would already be able to infer their status. Let's assume a child i is clean. If there are enough muddy children, i can receive a message from a muddy child with round $N-2$, and update their round to $N-1$ (using the reasoning above) while maintaining their status as unknown (since the child still sees more muddy children). Note that $N-1$ is its maximal round according to the invariant proposed above. But, at the same time, one of the other muddy children, say j can receive the same message and update their round to $N-1$, which coincides to the number of muddy children they see, so they would change their status to knowing that they are muddy. Now if j sends a message with their new round and status and i receives it, then i would know they must be clean, but the question is: what would happen with the child's round? If i keeps the same round (as to not violate the invariant), then they would have two statuses at the same round. To avoid that, we decide to break the invariant in this case and let a clean child advance to round N when they infer they are clean, thus staying in sync with the traditional solution to the puzzle.

Thus we can rephrase the first invariant as: "a message $\langle j, r, status \rangle$ with status different than clean guarantees that j knows that there are more than r muddy children", and add a second property, saying that a message $\langle j, r, status \rangle$ with status different than unknown guarantees that j sees precisely r muddy children.

In the sequel, we propose a transition function (and constraint predicate) to help us realize the proposal above.

Init From the initial state, each child takes one (silent) transition, analyzing their current knowledge and initializing the dynamic part of the state accordingly.

- The round number is initialized with 0
- If the set of muddy children the child sees is empty, then the knowledge flag is set to muddy (since at least one must be muddy); otherwise to unknown

$\tau_i(init, \langle Obs \rangle, \aleph) = (\langle Obs, 0, u \rangle, \aleph)$, if $Obs \neq \emptyset$
$\tau_i(init, \langle Obs \rangle, \aleph) = (\langle Obs, 0, m \rangle, \aleph)$, if $Obs = \emptyset$
$\beta_i(init, \langle Obs \rangle, m) = (m = \aleph)$

$$
\begin{array}{lrl}
\tau_i(\mathit{receive}, \langle \mathit{Obs}, r, \mathit{status}\rangle, \langle j,\, r',\, \mathit{status}'\rangle) = & (\langle \mathit{Obs}, r, \mathit{status}\rangle, \otimes), & \text{if } \mathit{status} \in \{m, c\} \\
& (\langle \mathit{Obs}, r', c\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = c \\
& & \text{and } j \notin \mathit{Obs} \text{ and } r' = |\mathit{Obs}| \\
& (\langle \mathit{Obs}, r' - 1, m\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = c \\
& & \text{and } j \notin \mathit{Obs} \text{ and } r' = |\mathit{Obs}| + 1 \\
& (\langle \mathit{Obs}, r', m\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = m \\
& & \text{and } j \in \mathit{Obs} \text{ and } r' = |\mathit{Obs}| \\
& (\langle \mathit{Obs}, r' + 1, c\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = m \\
& & \text{and } j \in \mathit{Obs} \text{ and } r' = |\mathit{Obs}| - 1 \\
& (\langle \mathit{Obs}, r, \mathit{status}\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = u \\
& & \text{and } j \in \mathit{Obs} \text{ and } r' < r \\
& (\langle \mathit{Obs}, r' + 1, \mathit{status}\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = u \\
& & \text{and } j \in \mathit{Obs} \text{ and } r \leq r' < |\mathit{Obs}| - 1 \\
& (\langle \mathit{Obs}, r' + 1, m\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = u \\
& & \text{and } j \in \mathit{Obs} \text{ and } r' = |\mathit{Obs}| - 1 \\
& (\langle \mathit{Obs}, r, \mathit{status}\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = u \\
& & \text{and } j \notin \mathit{Obs} \text{ and } r' \leq r \\
& (\langle \mathit{Obs}, r', \mathit{status}\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = u \\
& & \text{and } j \notin \mathit{Obs} \text{ and } r < r' < |\mathit{Obs}| \\
& (\langle \mathit{Obs}, r', m\rangle, \otimes), & \text{if } \mathit{status} = u \text{ and } \mathit{status}' = u \\
& & \text{and } j \notin \mathit{Obs} \text{ and } r' = |\mathit{Obs}| \\
& \langle \mathit{Obs}, r, \mathit{status}\rangle, \otimes), & \text{for the cases not treated above}
\end{array}
$$

Figure 1: The transition function for the *receive* label

Emit From any non-initial state, a child can emit a message consisting of their identifier, current round number and epistemic status, without changing state.

$\tau_i(\mathit{emit}, \langle \mathit{Obs}, r, \mathit{status}\rangle, \otimes) = (\langle \mathit{Obs}, r, \mathit{status}\rangle, \langle i,\, r,\, \mathit{status}\rangle)$
$\beta_i(\mathit{emit}, \langle \mathit{Obs}, r, \mathit{status}\rangle, m) = (m = \otimes)$

Receive To update their state, whenever receiving a message $\langle j, r', \mathit{status}'\rangle$ in a state $\langle \mathit{Obs}, r, \mathit{status}\rangle$, the child does the following:

- If their current *status* is not u, they ignore the message (decisions are final).

  $\tau_i(\mathit{receive}, \langle \mathit{Obs}, r, \mathit{status}\rangle, \langle j,\, r',\, \mathit{status}'\rangle) = (\langle \mathit{Obs}, r, \mathit{status}\rangle, \otimes)$, if $\mathit{status} \in \{m, c\}$

- Otherwise:

  – If message status (status') is c, and j is not known to be muddy ($j \notin \mathit{Obs}$), then from the property above r' must represent the actual number of muddy children. Hence:

* If $r' = |Obs|$, then child i can update their round to the received message's round and then conclude that they are clean.

  $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r', c\rangle, \textcircled{x})$

* If $r' = |Obs| + 1$, then child i can update their round to the round before received message's round and then conclude that they are muddy.

  $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r' - 1, m\rangle, \textcircled{x})$

– If message status ($status'$) is m then it must be that j is known as muddy ($j \in Obs$); then, from the property above, $r' + 1$ must represent the actual number of muddy children. Hence:

  * If $r' = |Obs|$, then child i can update their round to the received message's round and then conclude that they are muddy.

    $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r', m\rangle, \textcircled{x})$

  * If $r' = |Obs| - 1$, then child i can update their round to round after the received message's round and then conclude that they are clean.

    $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r' + 1, c\rangle, \textcircled{x})$

– If message status ($status'$) is u, and j is known as muddy ($j \in Obs$), then i can infer that j knows that there are more than r' muddy children, and therefore infers that there are more than $r' + 1$ muddy children. Hence:

  * If $r' < r$, then child i can ignore the message (it brings nothing new).

    $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r, status\rangle, \textcircled{x})$

  * If $r \leq r' < |Obs| - 1$, then child i can update their round to $r' + 1$, but their status will remain unknown (they already know there are at least $|Obs|$ muddy children).

    $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r' + 1, status\rangle, \textcircled{x})$

  * If $r' = |Obs| - 1$, then child j sees at least $|Obs|$ muddy children. Since the sender is known as muddy, there are at least $|Obs| + 1$ muddy children. On the other hand, child i knows there are at most $|Obs| + 1$ muddy children. Combining the two inequalities, we get that there are precisely $|Obs| + 1$ muddy children, so child i can advance to round $r' + 1$ and knows they are muddy.
    $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r' + 1, m\rangle, \textcircled{x})$

– If message status ($status'$) is unknown, and j is not known as muddy ($j \notin Obs$), then i can infer that j knows that there are more than r' muddy children, and therefore infers (only) that there are more than r' muddy children. Hence:

  * If $r' \leq r$, then child i can ignore the message.

    $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r, status\rangle, \textcircled{x})$

* If $r < r' < |Obs|$, then child i can update its round to r', but its status will remain unknown (it already knows there are at least $|Obs|$ muddy children).

  $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r', status\rangle, \text{⊠})$

* If $r' = |Obs|$, we reason analogous to the similar above case: child j knows that there are at least $|Obs|+1$ muddy children and. Adding the fact that child i knows there are at most $|Obs|+1$ muddy children, we get that there are precisely $|Obs|+1$ muddy children, so child i knows they are muddy.

  $\tau_i(receive, \langle Obs, r, status\rangle, \langle j, r', status'\rangle) = (\langle Obs, r', m\rangle, \text{⊠})$

Composition. Note that the notion of VLSM composition in general allows arbitrary initial states for the components. However, in this setting, we need to ensure that the sets of children known by each child to be muddy are *consistent*. Formally, given a child-state s, let $Obs(s)$ be the observation-set associated to s. Then, for any composite state $\sigma = \langle \sigma_1, \ldots, \sigma_n\rangle$, let **consistent**$(\sigma)$ be the predicate defined by

- $M \neq \emptyset$ (there should be at least one muddy child)
- $Obs(\sigma_i) = M \setminus \{i\}$, for any i (each child sees all other muddy children)

where $M = \bigcup_{i=1}^{n} Obs(\sigma_i)$.

Finally, the game flow can be formalized as a constrained VLSM composition.

$$\mathscr{M}uddy\mathscr{P}uzzle = \Big(\mathscr{C}_1 + \ldots + \mathscr{C}_n\Big)\Big|_{\varphi}$$

where φ specifies the following composition constraint:

init At the first transition from an initial state we check that the observation sets corresponding to each component are consistent

$\varphi((i, init), (\langle Obs_1\rangle, \ldots, \langle Obs_n\rangle), m) = \textbf{consistent}(\langle Obs_1\rangle, \ldots, \langle Obs_n\rangle)$

receive We must enforce a no-equivocation constraint to ensure the truthfulness of the participants

$\varphi(\langle i, receive\rangle, \langle \sigma_1, \ldots, \sigma_n\rangle, \langle j, r', status'\rangle) = (status' = status_j \wedge r' = r_j) \vee (status' = u \wedge r' < r_j)$, where $\sigma_j = \langle Obs, r_j, status_j\rangle$.

4.1 Correctness of the protocol

In the following, we give a justification of the fact that the above described protocol is correct in the sense that it converges to a solution. The valid states of the protocol (S_V) correspond to the composite states which are VLSM-valid in the constrained composition described above. We define the set of non-initial valid states $S_V^* = S_V \setminus S_0$ and the set of final states $S_F = \{\langle \sigma_i \ldots \sigma_n\rangle \in S_V \mid \forall i \in \{1, \ldots, n\}, status(\sigma_i) \neq u\}$.

It can be easily checked that the consistency predicate holds for any $\sigma \in S_V^*$:

Remark 1. *For each* $\sigma \in S_V^*$, **consistent**(σ) *holds.*

Another VLSM-related property, which we state without proof is the fact that the constraint for *receive* transitions is indeed a no-equivocation constraint:

Remark 2 (No Equivocation)**.** *For any valid trace tr leading from an initial state σ_0 to a state $\sigma \in S_v^*$ and for any input message m which is valid for σ there exists a valid trace starting in σ_0, of length less than that of tr which emits m.*

Let us now show that the invariant that we stated about the dynamics of the protocol is indeed preserved during a (valid) protocol run.

Lemma 1 (Invariant Preservation)**.** *For any $\sigma \in S_V$, if $\sigma \notin S_0$, then:*

For any component i of σ of the form $\sigma_i = \langle Obs_i, r_i, status_i \rangle$:

- *If $status_i = u$, then $r_i < |Obs_i|$ (and $|Obs_i| \leq N$)*
- *If $status_i = m$, then $r_i = N - 1 = |Obs_i|$*
- *If $status_i = c$, then $r_i = N = |Obs_i|$*

Proof. Note that it is common knowledge that $|Obs_i| \leq N \leq |Obs_i| + 1$ for any i.

We prove the invariant by induction on the length of a valid trace leading to σ. The property trivially holds for $\sigma \in S_0$. For the induction case, we consider the final (valid) transition leading to σ, say $\sigma' \xrightarrow[m \to m']{(i,l)} \sigma$, and we assume that the invariant holds for σ'. We proceed by case analysis on the label of the transition.

$l = init$**:** This transition obviously preserves the invariant, because of the father's statement.

$l = emit$**:** In case of a *emit* transition, the conclusion is also immediate, since the state remains unchanged.

$l = receive$**:** From Remark 2 there is a composite valid state from which the input message can be emitted which has the same observation sets as σ' (since they both are reachable from the same initial state) and for which we can apply the induction hypothesis and thus assume that the invariant holds.

- $\tau_i(receive, \langle Obs_i, r, m \rangle, \langle j, r', status' \rangle)$: the message is ignored and the state remains unchanged, so the conclusion is immediate.
- $\tau_i(receive, \langle Obs_i, r, c \rangle, \langle j, r', status' \rangle)$: we proceed analogous to the previous case.
- $\tau_i(receive, \langle Obs_i, r, u \rangle, \langle j, r', c \rangle)$ and $j \notin Obs_i$ and $r' = |Obs_i|$:
  By applying the induction hypothesis to the state from which the message is obtained, we have $r' = N = |Obs_j|$.
  The resulting state $\langle Obs_i, r', c \rangle$ preserves the invariant, because $r' = N = |Obs_i|$.
- $\tau_i(receive, \langle Obs_i, r, u \rangle, \langle j, r', c \rangle)$ and $j \notin Obs_i$ and $r' = |Obs_i| + 1$:
  By applying the induction hypothesis to the state from which the message is obtained, we have $r' = N = |Obs_j|$.
  The resulting state $\langle Obs_i, r' - 1, m \rangle$ preserves the invariant, because $r' - 1 = N - 1 = |Obs_i|$.
- $\tau_i(receive, \langle Obs_i, r, u \rangle, \langle j, r', m \rangle)$ and $j \in Obs_i$ and $r' = |Obs_i|$:
  By applying the induction hypothesis to the state from which the message is obtained, we have $r' = N - 1 = |Obs_j|$.
  The resulting state $\langle Obs_i, r', m \rangle$ preserves the invariant, because $r' = N - 1 = |Obs_i|$.

- $\tau_i(receive, \langle Obs_i, r, u\rangle, \langle j, r', m\rangle)$ and $j \in Obs_i$ and $r' = |Obs_i| - 1$:
  By applying the induction hypothesis to the state from which the message is obtained, we have $r' = N - 1 = |Obs_j|$.
  The resulting state $\langle Obs_i, r' + 1, c\rangle$ preserves the invariant, because $r' + 1 = N = |Obs_i|$.
- $\tau_i(receive, \langle Obs_i, r, u\rangle, \langle j, r', u\rangle)$ and $j \in Obs_i$ and $r' < r$:
  The state after applying the transition remains unchanged, so the conclusion is immediate.
- $\tau_i(receive, \langle Obs_i, r, u\rangle, \langle j, r', u\rangle)$ and $j \in Obs_i$ and $r \leq r' < |Obs_i| - 1$:
  The resulting state $\langle Obs_i, r' + 1, u\rangle$ obviously preserves the invariant, since $r' + 1 < |Obs_i|$.
- $\tau_i(receive, \langle Obs_i, r, u\rangle, \langle j, r', u\rangle)$ and $j \in Obs_i$ and $r' = |Obs_i| - 1$:
  We have $r' + 1 = |Obs_i| \geq N - 1$, so $r' \geq N - 2$.
  On the other hand, applying the induction hypothesis, we have that $r' < |Obs_j|$ and since child j is muddy, $|Obs_j| = N - 1$, and combining these we get $r' < |Obs_j| = N - 1$, which implies $r' \leq N - 2$.
  We can conclude that $r' = N - 2$, so the resulting state $\langle Obs_i, r' + 1, m\rangle$ preserves the invariant, because $r' + 1 = |Obs_i| = N - 1$.
- $\tau_i(receive, \langle Obs_i, r, u\rangle, \langle j, r', u\rangle)$ and $j \notin Obs_i$ and $r' <= r$:
  The state after applying the transition remains unchanged, so the conclusion is immediate.
- $\tau_i(receive, \langle Obs_i, r, u\rangle, \langle j, r', u\rangle)$ and $j \notin Obs$ and $r < r' < |Obs_i|$:
  The resulting state $\langle Obs_i, r', status\rangle$ preserves the invariant, since $r' < |Obs_i|$.
- $\tau_i(receive, \langle Obs_i, r, u\rangle, \langle j, r', u\rangle)$ and $j \notin Obs_i$ and $r' = |Obs_i|$:
  By applying the induction hypothesis to the state from which the message is obtained, we have $r' < |Obs_j|$ and since child j is clean, $|Obs_j| = N$ so we get $r' < N$ and combining this with the last condition in the transition, we immediately obtain that $|Obs_i| < N$.
  But since it is common knowledge that $|Obs_i| \geq N - 1$ it must be that $|Obs_i| = N - 1$. We can conclude that $r' = |Obs_i| = N - 1$, so the resulting state $\langle Obs_i, r', m\rangle$ preserves the invariant.
- For all the cases not treated above, the child's β predicate does not hold, so there cannot be any transition.

□

Theorem 1. *From any initial consistent state, there is a path leading to a final state in which each child's status is consistent with the instance of the problem.*

Proof sketch. The result can be obtained from the following properties:

Progress From any valid non-final state σ, there is a valid transition leading to a state σ' on some component i, such that $round(\sigma'_i) > round(\sigma_i)$, where we let $round(\langle Obs\rangle)$ be -1.

If there are any component initial states left, we advance one of them. If there are any children which already know their status, we advance to the final state any of the others (there must be at least one child with unknown status since the state is not final). If no child knows their status yet, then there must be at least two muddy children. Then take the one among them with minimum round number (if their round numbers are equal, take any of them) and receive the message corresponding to the current state of the other. This will surely increase their current round number.

Invariant Preservation The invariant from Lemma 1 holds.

In particular, if a final state is reached, each child's status will be consistent with the current instance of the problem.

Termination No child can increase their round number past the number of muddy children they see.

Easy to prove by induction and analysis on the cases of the transition function.

□

4.2 Possible optimization

If we analyze the dynamics of the solution proposed above, we notice that the only relevant exchange of messages happens in the final stages of the protocol. Indeed, assuming a child sees n muddy children, they know that any other child sees at least $n-1$ muddy children; therefore no message it would send with round less than $n-1$ can help any other child determine their status. Hence, the child can "jump" during the initialization phase directly to the round $n-1$. Formally, we add a new label **jump** and replace the **Init** transition with the following:

From any state with status unknown, a child can take one (silent) transition "jumping" to a future state it knows it can reach based on existing knowledge, where the round number becomes the number preceding the number of muddy children the child sees and the child's knowledge flag remains unknown.

$\tau_i(jump, \langle Obs, r, status\rangle, m) = (\langle Obs, |Obs|-1, u\rangle, \otimes)$
$\beta_i(jump, \langle Obs, r, status\rangle, m) = (status = u) \wedge (m = \otimes) \wedge (r < |Obs|-1)$

After such a jump one can find a very short path to a solution:

1. a child sends a message after the jump with the unknown status
2. another (muddy) child receives that message and discovers they are muddy and sends a message with this discovery
3. all other children (including the first) receive this second message and discover their status

4.3 Discussion

The above proposed solution seems to satisfy our initial guidelines for a good asynchronous solution. Nevertheless, it is far from being perfect, as illustrated by the following example.

Example 1 (Information leak)**.** *Assume there are* 5 *children,* 4 *of whom are muddy and one is not. Then a muddy child (say, child* 1*) can discover that they are muddy by only exchanging messages with the clean child (say, child* 5*) with the following scenario:*

1. 1 *at round* 0 *sends message* m_0^1 *that they do not know*
2. 5 *at round* 0 *receives message* m_0^1*, advances to round* 1*, and sends message* m_1^5 *that they do not know*
3. 1 *at round* 0 *receives the message* m_1^5*, advances to round* 1*, and sends message* m_1^1 *that they do not know*

4. 5 *at round* 1 *receives message* m_1^1*, advances to round* 2*, and sends message* m_2^5 *that they do not know*
5. 1 *at round* 1 *receives the message* m_2^5*, advances to round* 2*, and sends message* m_2^1 *that they do not know*
6. 5 *at round* 2 *receives message* m_2^1*, advances to round* 3*, and sends message* m_3^5 *that they do not know*
7. 1 *at round* 2 *receives the message* m_3^5*, advances to round* 3 *which equals the number of muddy children they see and changes their status to knowing they are muddy (and sends message* m_3^1 *with status muddy)*

Moreover, once 1 *knows they are muddy and sends a message about it, upon receiving that message all the other children will know their status.*

There are at least two issues about the above example: (1) that a muddy child needs no information from other muddy children to discover that they are muddy; and (2) that once a child knows their status, all other children immediately know their status, even if they have not taken part in the conversation so far.

It thus seems that, although sharing the perceived round as part of the message helps with making the discovery process asynchronous, it also alters it more than expected in terms of what becomes inferable during an exchange of information. The following section proposes a solution we believe to be closer to the original formulation and intended dynamics, while staying asynchronous.

5 Solving the puzzle by extracting information from messages

To alleviate the issues identified with the solution in Section 4, we propose a new solution, which follows more closely an epistemic logic point of view, in the sense that the messages exchanged between children only carry information that the child has previously seen. This guarantees that deduction can only be done according to information that is (was) publicly available.

To do that, we define the child i as the following VLSM, resembling the previous encoding, but using *message histories* (lists of messages previously received) instead of round numbers:

- $L_i = \{init, emit, receive\}$
- $M = \{\langle j, s, h\rangle \mid j \in \{1,\ldots,n\},\ s \in \{u,m,c\},\ h \in M^*\}$
- $M_{0,i} = \emptyset$
- $S_i = \{\langle Obs\rangle \mid Obs \subseteq \{1,\ldots,n\}\} \cup \{\langle Obs, s, h\rangle \mid Obs \subseteq \{1,\ldots,n\},\ s \in \{u,m,c\},\ h \in M^*\}$
- $S_{0,i} = \{\langle Obs\rangle \mid Obs \subseteq \{1,\ldots,n\}\}$,

Let us now describe the dynamics of a child's interaction with the others.

Init The initialization phase will remain basically the same as for the previous solution, except that, instead of round 0, now the children will have the empty history:

$$\tau_i(init, \langle Obs\rangle, \text{⊠}) = \begin{cases} (\langle Obs, u, []\rangle, \text{⊠}), & \text{if } Obs \neq \emptyset \\ (\langle Obs, m, []\rangle, \text{⊠}), & \text{if } Obs = \emptyset \end{cases}$$
$$\beta_i(init, \langle Obs\rangle, msg) = (msg = \text{⊠})$$

Emit Similarly, the emit phase is basically the same, except that, instead of sending their round number, the children will now send their current history:

$$\tau_i(emit, \langle Obs, s, h\rangle, \text{⊠}) = (\langle Obs, s, h\rangle,\ \langle i, s, h\rangle)$$
$$\beta_i(emit, \langle Obs, s, h\rangle, msg) = (msg = \text{⊠})$$

Receive Upon receiving a message (in a not yet known status) the child will aggregate the message received with the information they already had (append it to its history) and compute its new status based on its *Obs*ervation set and its new history.

$$\tau_i(receive, \langle Obs, s, h\rangle, \langle j, s_j, h_j\rangle) =$$

$$\begin{cases} (\langle Obs, s, h\rangle, \text{⊠}), & \text{if } s \in \{c, m\} \\ (\langle Obs, compute(Obs, h +\!\!+ [\langle j, s_j, h_j\rangle]), h +\!\!+ [\langle j, s_j, h_j\rangle]\rangle, \text{⊠}), & \text{if } s = u \end{cases}$$

// we compute the new status

$$compute(Obs, h) = \begin{cases} m, & \text{if } \min_{k \in Obs} |groupSimilar_i(unknown_k(flatten(h)))| \geq |Obs| \\ c, & \text{if the first rule didn't apply and } |muddy(flatten(h))| = |Obs| \\ u, & \text{otherwise} \end{cases}$$

// we unfold all the messages contained (in depth) in the history

$$flatten(h +\!\!+ [\langle j, s_j, h_j\rangle]) = flatten(h) \cup flatten(h_j) \cup \{\langle j, s_j, h_j\rangle\} \cup extra(\langle j, s_j, h_j\rangle)$$

// we extract extra information from a message about messages which could have been emitted by its sender (e.g. in all prefixes of the history in which the sender didn't know their status)

$$extra(\langle j, s_j, h_j\rangle) = \{\langle j, u, h'_j\rangle \mid h'_j \lhd h_j\}$$[2]

// we select messages belonging to a given child and having an unknown status

$$unknown_k(h) = \{m \mid m \in h \wedge m = \langle k, u, h'\rangle \text{ for some h'}\}$$

// we group messages that carry the same information relative to child i

$$groupSimilar_i(h) = h \setminus \{\langle j,\ u,\ h' +\!\!+ [\langle i, \_, \_\rangle]\rangle \text{ for some h'}\}$$

// we construct the set of all children who know they are muddy in a history

$$muddy(h) = \{j \mid \langle j, m, h_j\rangle \in h\}$$

Finally, the game can be formalized as a constrained VLSM composition.

$$\mathscr{M}uddy\mathscr{P}uzzle = \Big(\mathscr{C}_1 + \mathscr{C}_2 + \mathscr{C}_3\Big)\Big|_{\varphi}$$

where

[2] We denote by $h_1 \lhd h_2$ the fact that h_1 is a strict prefix of h_2.

// consistency
$\varphi((i,init),(\langle Obs_1\rangle,\langle Obs_2\rangle,\langle Obs_3\rangle),m) = \textbf{consistent}(\langle Obs_1\rangle,\langle Obs_2\rangle,\langle Obs_3\rangle)$,

// no equivocation
$\varphi((i,receive),\langle\sigma_1,\sigma_2,\sigma_3\rangle,\langle j,s_j,h_j\rangle) =$

$$(s_j = status(\sigma_j) \wedge h_j = history(\sigma_j)) \vee (s_j = u \wedge h_j \triangleleft history(\sigma_j)),$$

where for a state $\sigma = \langle Obs, s, h\rangle$, we define $status(\sigma) = s$ and $history(\sigma) = h$.

It is relatively easy to see that the kind of reasoning described in the (counter-)Example 1 is no longer possible, because after the first exchange of messages, child 1 cannot really infer anything, because all they see is a message from child 5 containing a message from themselves.

We argue that the information contained in a message represents the epistemic knowledge of its sender about the status of the other children at the time the message was sent, thus making this solution much closer to the standard solution (using synchronous rounds and public announcements).

First, let N be the total number of muddy children, and let us observe that if a child knows N, then they also know their status (by comparing N with the number of muddy visible children). On the other hand, telling the others that they know N (without actually telling them the value) is no different than telling them that their status can be inferred. Let $K_1,\ldots,K_n$ be the epistemic operators assigned to each child, with the intuitive meaning of $K_i\psi$ that i can infer that ψ holds, based on the commonly available information, and on the private information that i has. Let also $q_1,\ldots,q_n$ be primitive propositions, q_i meaning that i knows the value of N.

Then, we can recursively encode a message m as a formula $E(m)$, as follows:

$$E(\langle j,s_j,h_j\rangle) = \begin{cases} (K_j \bigwedge_{m\in h_j} E(m)) \wedge K_j\left(\bigwedge_{m\in h_j} E(m) \rightarrow q_j\right) & \text{if } s_j \neq u \\ (K_j \bigwedge_{m\in h_j} E(m)) \wedge \neg K_j\left(\bigwedge_{m\in h_j} E(m) \rightarrow q_j\right) & \text{if } s_j = u \end{cases}$$

One important observation is that the formulas corresponding to all additional messages introduced by *flatten(h)* are directly deducible from the formulas of h.

Discussion. Our preliminary exploration of the interactions possible within the model seems to hint that a solution is reachable from any initial state for the case of three children. Nevertheless, our current definition of *groupSimilar* was engineered for the case of three children and we know it doesn't scale to more children as-is. We believe that for the general case it should be replaced by factoring the messages through an equivalence relation grouping messages which carry the similar amount of information as perceived by the child receiving these messages.

6 Conclusion and Future Work

In this paper we have shown that compositions of VLSMs seem like a useful tool to model the dynamics of distributed agents, allowing independent description of their behavior, but also enabling the specification of global constraints (e.g., truthfulness, fairness, etc.).

Regarding the models proposed for the Muddy Children Puzzle, for the first model we were able to show that it achieves the goal of converging towards a solution, but also showed that it might leak more information than intended through communication. We believe the second model to be promising and more faithful to our modeling goals and plan to further explore its possible generalizations to an arbitrary number of children.

References

[1] R. Fagin, J.Y. Halpern, Y. Moses & M. Vardi (2004): *Reasoning About Knowledge*. A Bradford Book, MIT Press, doi:10.7551/mitpress/5803.001.0001.

[2] Pedro Fonseca, Kaiyuan Zhang, Xi Wang & Arvind Krishnamurthy (2017): *An Empirical Study on the Correctness of Formally Verified Distributed Systems*. In: *European Conference on Computer Systems*, pp. 328–343, doi:10.1145/3064176.3064183.

[3] Jelle Gerbrandy & Willem Groeneveld (1997): *Reasoning about information change*. *Journal of logic, language and information* 6, pp. 147–169, doi:10.1023/A:1008222603071.

[4] Nina Gierasimczuk & Jakub Szymanik (2011): *A note on a generalization of the muddy children puzzle*. In: *Proceedings of the 13th Conference on Theoretical Aspects of Rationality and Knowledge*, pp. 257–264, doi:10.1145/2000378.2000409.

[5] Elaine Li, Traian Florin Șerbănuță, Denisa Diaconescu, Vlad Zamfir & Grigore Roşu (2020): *Formalizing Correct-By-Construction Casper in Coq*. In: *International Conference on Blockchain and Cryptocurrency*, pp. 1–3, doi:10.1109/ICBC48266.2020.9169468.

[6] Minghui Ma, Alessandra Palmigiano & Mehrnoosh Sadrzadeh (2014): *Algebraic Semantics and Model Completeness for Intuitionistic Public Announcement Logic*. *Annals of Pure and Applied Logic* 165(4), pp. 963–995, doi:10.1016/j.apal.2013.11.004.

[7] Daniel Miedema & Malvin Gattinger (2023): *Exploiting Asymmetry in Logic Puzzles: Using ZDDs for Symbolic Model Checking Dynamic Epistemic Logic*. *arXiv preprint arXiv:2307.05067*, doi:10.48550/arXiv.2307.05067.

[8] Hans Van Ditmarsch, Wiebe van Der Hoek & Barteld Kooi (2007): *Dynamic epistemic logic*. 337, Springer Science & Business Media, doi:10.1007/978-1-4020-5839-4.

[9] V. Zamfir, M. Calancea, D. Diaconescu, W. Kołowski, B. Moore, K. Palmskog, T.F. Șerbănuță, M. Stay, D. Trufaș & J. Tušil (2022): *Validating Labelled State Transition and Message Production Systems: A Theory for Modelling Faulty Distributed Systems*. *arXiv:2202.12662v3*, doi:10.48550/ARXIV.2202.12662.

[10] Vlad Zamfir, Nate Rush, Aditya Asgaonkar & Georgios Piliouras (2019): *Introducing the "Minimal CBC Casper" Family of Consensus Protocols*. https://github.com/cbc-casper/cbc-casper-paper.