Contents

1

1. Introduction

In 1956 Edward F. Moore published a paper with the topic of recreating a finite state machine via experiments[12] where each experiment is a kind of membership query. This was one of the earliest instances of extracting information from a automaton, knowledge of specific states and transitions. Then Tsyh-Wen Pao and John W. Carr expanded this idea with a way to fully describe a Language via experiments. Their algorithm obtains a subset of the Language, with words that exercise every live transition, and has the ability to ask if any string is part of the Language [13]. While this approach allows for full learning of an Automaton, it is not well suited for black box applications and the subset significantly simplifies the task. To combat this Dana Angluin created of the Minimal Adequate Teacher(MAT) framework [1]. In the new MAT framework the idea of membership queries stays but a second kind of query, checking if two languages are equal and returning a counterexample if not, is added. In the same Paper Dana Angluin proposes the L^* algorithm that can learn an automaton in a polynomial amount of queries. Since then the $L*$ algorithm has been used in multiple applications for instance formal Verification of black box Systems [15], ... The L^* algorithm has seen many improvements over time [**?**] and different learning algorithms such as TTT [8], or algorithms using Homing Sequences [17] and the $L\#$ algorithm [19]. In this thesis we will focus on verifying the $L\#$ algorithm formally.

While Frits Vandraager already verified that the $L\#$ algorithm works, this verification was done on paper [19]. Proofs on paper are easy to read but can be error prone, for example a proof for the Two-Children problem from Martin Gardener and the proof on three-dimensional simple orthorhombic Ising lattices by Zhidong Zhang et al. were later found to contain errors[9, 16]. Formal verification is a way to avoid errors in proofs altogether, one of the earliest formalization, the Principa Mathematica, was done entirely on paper and spans three whole books. It only uses a few axioms and inference Rules to prove a lot of fundamental mathematics. [21] A simple assumption would be that in the framework of the Principa Mathematica every mathematical problem is solvable, but Kurt Gödel showed that there are true statements that are not provable in any formal system [6]While the formal principles of the Principa Mathematica are already advanced and simple to check, formalizations on paper are long and can still contain errors. Machine checked formalizations can prevent errors, but require difficult programming. To develop a proof one idea is to let the machine reason the whole proof, leading to automatic theorem provers. One example of automatic theorem provers is Vampire, a First order logic solver that gets expanded to this day [4], or the SMT solver Z3 that is developed by Microsoft[5]. While automatic theorem provers are helpful for simple proofs they struggle with complex proofs [2]. A more expressive way of proving machine checked are interactive Theorem Provers (ITP). In an ITP the machine and the user work together to achieve a proof, often the ITP also allows access to multiple automatic theorem provers to help achieve a formal proof. Two of the most popular ITPs are Isabelle[14] and HOL light [7]. There is a list of 100 arbitrary important mathematical proofs and Isabelle and

2

HOL light lead the comparison of most proven lemmata out of this list [1]. The Isabelle theorem prover has been successfully to prove the seL4 micro kernel [10], Generation of LL(1)parsers [18], the post quantum cryptography protocol Kyber [11] and many more.
chapter 2 explains the $L\#$ automata learning algorithm for mealy machines. It focuses on the reasoning why different apsects of the algorithm work. chapter 3 shows the Formalization of the algorithm in Isabelle and explains the core theorems of this formalization.

[1] https://www.cs.ru.nl/~freek/100/

2. L#

Before discussing the formalization of the $L\#$ algorithm it is important to understand how it works. The following chapter will discuss the algorithm following the explanation of Frits Vandraager et al. in the paper "A New Approach for Active Automata Learning Based on Apartness" [19].

2.1. Functions

First we define the notation used for functions. We write $f : X \rightharpoonup Y$ to denote that f is a partial function from X to Y. To denote that f is defined for a input x we write $f(x) \downarrow$ and to denote that f is undefined for input x we write: $f(x) \uparrow$. A function is called total if $f(x) \downarrow$ for all x, we write $f : X \to Y$ to denote that f is a total function from X to Y. The composition of two partial functions $f : X \rightharpoonup Z$ and $g : Y \rightharpoonup Z$ is denoted as $g \circ f : X \rightharpoonup Y$, then the for the composition $g \circ f(x) \downarrow \iff (f(x) \downarrow \wedge\, g(f(x)) \downarrow)$ holds.

2.2. Mealy Machines and apartness

A Mealy Machine can be understood as a finite state automaton with transitions that can accept different inputs and return outputs.

Definition 1. Any Mealy Machine M is a 6-tuple $M = (Q, q_0, I, O, \delta, \lambda)$where

- Q is the finite set of states
- q_0 is the initial state
- I is the finite set of input symbols
- O is the finite set of output symbols
- $\delta : Q \times I \rightharpoonup Q$ the state transition function
- $\lambda : Q \times I \rightharpoonup O$ the output function

It is important to note that both functions need to be defined on the same inputs $\delta(x) \downarrow \iff \lambda(x) \downarrow$. For combined use of output and transition we use $\langle \lambda, \delta \rangle : Q \times I \rightharpoonup O \times Q$ as the output transition function, e.g. if $\lambda(q, i) = o$ and $\delta(q, i) = q'$ then $\langle \lambda, \delta \rangle(q, i) = (o, q')$. To denote that $\langle \lambda, \delta \rangle(q, i) = (q', o)$ we write $q \xrightarrow{i/o} q'$. Furthermore, to use the transition output function with words of length $n \in \mathbb{N}$ we compose $\langle \lambda, \delta \rangle$ ntimes with itself:

Definition 2. For a Mealy Machine $M = (Q, q_0, I, O, \delta, \lambda)$ we define the composition of multiple functions $\langle \lambda_n, \delta_n \rangle : Q \times I^n \rightharpoonup O^n \times Q$, inductivley. The base case is defined as the identity function of states with no output: $\langle \lambda_0, \delta_0 \rangle = id_Q$ and each step $\langle \lambda_{n+1}, \delta_{n+1} \rangle$ using

4

the previous step, and adding one application of the transition output system $\langle \lambda_{n+1}, \delta_{n+1} \rangle = Q \times I^{n+1} \xrightarrow{\langle \lambda_n, \delta_n \rangle \times id_I} O^n \times Q \times I \xrightarrow{id_{O^n} \times \langle \lambda, \delta \rangle} O^{n+1} \times Q$.

A Mealy Machine is complete if δ (and therefore λ) is total. To differentiate between Mealy Machines we the name of the mealy machine as superscript e.g. $M = (Q^M, q_0^M, I, O, \delta^M, \lambda^M)$ or $N = (Q^N, q_0^N, I, O, \delta^N, \lambda^N)$.
Semantically a state q of a Mealy Machine is a map from input words to output words.

Definition 3. We write $[\![q]\!] : I^* \times O^*$ where $[\![q]\!](\sigma) = \lambda(q, \sigma)$. Two states q and q' are equivalent $q \approx q'$ iff $[\![q]\!] = [\![q']\!]$, both states can be from different Mealy Machines. Two Mealy Machines are equivalent if their initial states are equivalent $M \approx N \iff q_0^M \approx q_0^N$.

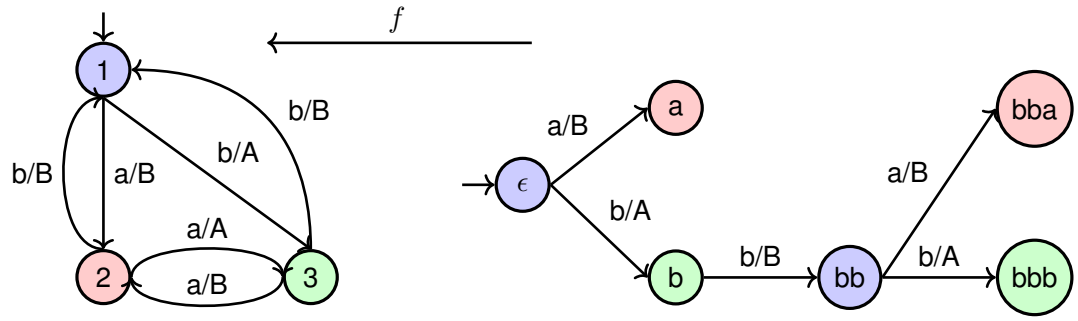
2.3. The observation Tree

For the learning context there are two Mealy Machines, a complete Mealy Machines that the teacher controls M and a partial Mealy Machine T the learner seeks to expand, with transitions that stem from information learned from the teacher. As the partial Mealy Machine seeks to emulate M an undefined transition represents a lack of knowledge. Furthermore the partial Mealy Machine is supposed to act the same way on all defined inputs as the hidden one. We define this behavior as functional simulation:

Definition 4. For two Mealy Machines M and N the map $f : M \to N$ is called a functional simulation if: f is a map $f : Q^M \to Q^N$ and $f(q_0^M) = q_0^N$ and $q \xrightarrow{i/o} q' \implies f(q) \xrightarrow{i/o} f(q')$

When the algorithm is running each transition it adds also adds a new state, so every state only has one incoming transition. Thus the hidden mealy machine is called a tree, specifically an observation tree. A Mealy Machine T is a tree if for each $q \in Q^T$ there is a unique accessor $\delta \in I^*$ so that $\sigma^T(q_0^T, \sigma) = q$, we write $accsess(q) = \sigma$. A Tree T is called an Observation tree for a Mealy Machine M if there is a functional simulation $f : T \to M$.

Figure 1 shows a complete Mealy Machine for $I^M = \{a, b\}$ as well as an observation tree T for M. The nodes of T are labeled with their respective accessor, e.g. $\sigma^T(q_0^T, bb) = [bb]$. Because T is an observation tree there exists an f that is a functional simulation from T to M $f : T \to M$, and each node in the observation tree has the color of the node that it corresponds with in M for example the functional simulation in Figure 1 the node labled with a corresponds to the node labeled with 2 in the Mealy Machine: $f([a]) = 2$.
Throughout the learning algorithm, the Observation Tree T expanded, but the functional simulation is unknown because the learner has no information about M. Thus a metric is needed



(a) A Mealy Machine M

(b) An observation tree T for M with accessor as node names

Figure 1 an observation tree simulates a Mealy

to produce a Mealy Machine that could be matching the hidden Mealy. With knowledge learned the algorithm can infer that two states can not correspond to the same state of the hidden , e.g. it can infer that $f([b]) \neq f([])$ because $\lambda^T([b], b) = B$ and $\lambda^T([], b) = A$. We call this form of inequality apartness.

Definition 5. For a Mealy Machine M, two states $p, q \in Q^M$ are apart (written $p\#q$) if there is some $\sigma \in I^*$ such that $[\![p]\!](\sigma) \downarrow \wedge [\![q]\!](\sigma) \downarrow$ and $[\![p]\!](\sigma) \neq [\![q]\!](\sigma)$. The word σ is a witness for $p\#q$, we write $\sigma \vdash p\#q$.

Because the apartness relation notes a difference in the semantics of two states, two apart states can not be mapped to the same state in a functional simulation $f : T \to M$.

$$p\#q, p, q \in Q^T \implies f(q) \not\approx f(p)$$

Thus whenever the learner knows that two states are apart in the observation tree, it knows that these are corresponding to different states in the hidden Mealy Machine.

2.4. The learning Algorythm $L\#$

With the invention of automata learning Dana Angluin also proposed the minimal adequate teacher framework. In this framework, the teacher is able to answer two types of queries accurately: a membership query, that allows to check if a string is in a hidden set and a equivalence query, checking if an automaton is equal to the hidden one. The second query allows the learner to ask if their description of a set produces the same set as the hidden set, if the answer is false the teacher also responds with a counterexample why the sets are not the same.[1]
For the purpose of learning a hidden Mealy Machine we define the two types of query's as follows:

Definition 6. in the learning game between a teacher and a learner where the teacher has

6

knowledge of a hidden Mealy Machine $M$and answers the following querys correctly:

$\texttt{OutputQuery}(\sigma)$: for $\sigma \in I^*$ the teacher replies with the output sequence $\lambda^M(q_0^M, \sigma) \in O^*$

$\texttt{EquivQuery}(H)$: for a complete Mealy Machine H the teacher replies with yes if $H \approx M$ or no and a counterexample $\sigma \in I^*$ with $\lambda^M(q_0^M, \sigma) \neq \lambda^H(q_0^H, \sigma)$

The output query lets the algorithm expand its observation tree along a chosen route σ, and the equivalence query can help if the algorithm can no longer ask productive output query's, but a the observation tree needs to be transformed to a Mealy Machine and the counterexample needs to be processed.

2.4.1. Frontier, Basis and Norm

The $L\#$ algorithm describes the learner in the learning process. The algorithm operates on an observation tree $T = (Q^T, q_0^T, I, O, \delta^T, \lambda^T)$ for the hidden Mealy Machine M. Furthermore, the learner knows I and O of the hidden Mealy Machine and uses them for the Observation Tree. The algorithm splits T into three sets:

1. The Basis $S \subseteq Q^T$ states which have already been identified. The learner knows that each state in S represents a distinct state in M. The Basis always contains the starting state $q_0^T \in S$ so initially: $S = \{q_0^T\}$. Since all states in S are different states in the hidden Mealy Machine the learner only adds states to the Basis that are pairwise apart: $\forall p, q \in S, p \neq q : p\#q$.
2. The Frontier $F \subseteq Q^T$ which are candidates to be added to S. Rather than changing the contents of F manually the learner defines that $F := \{q' \in Q \mid \exists q \in S, q' \notin S, i \in I : q' = \delta(q, i)\}$.
3. The remaining states $Q^T \backslash (S \cup F)$.

The algorithm aims to extend the Basis with states from the Frontier until there exists a state in S for each equivalence class in M. The algorithm achieves this through the application of four Rules. We will describe the algorithm as applying the four Rules non-determanistically, until no Rule can be applied anymore. The algorithm adds the response of every $\texttt{OutputQuery}(\sigma)$ to the observation tree. To argue about runtime and termination we define a Norm that is bound and that each Rule application increases. We define the Norm $N(T)$:

$$N(t) = \frac{|S| \cdot (|S| + 1)}{2} + |\{(q, i) \in S \times I \mid \delta^T(q, i) \downarrow\}| + |\{(q, q') \in S \times F \mid q\#q'\}|$$

Every Rule we define later will increase one of those summands, the first Rule increases the first summand, the second Rule increases the second summand and the third and fourth Rule increase the last summand. The first summand is exponentially proportional to the number of states in the Basis, because adding states to the Basis removes them from the Frontier, causing a drop in the third summand. The second summand tracks the number of defined

transitions for each state in s, this is increased as states get added to the Frontier. The last summand tracks the apartness relationship between S and F. For an observation tree T for a Mealy Machine M with n equivalence classes and $|I| = k$ the Norm is bound.

$$N(T) \leq \frac{n \cdot (n+1)}{2} + kn + (n-1)(kn+1)$$

intuitively: if each two states in S are pairwise apart and there exists a functional simulation $f : T \to M$ than there can only be as many states in S as there are equivalence classes in M. As I is static and F can only have $|S \times I| = |S| \cdot |I|$ states.

We will discuss each Rule and how they increase the Norm separately. To choose states the algorithm uses different metrics for states:

Definition 7. In an observation tree T a state $q \in F$ is called isolated if it is apart from all states in S: $\forall p \in S, q\#p$,q identified if it is apart from all but one state in S: $\exists p \in S, \neg q\#p, \forall s \in S, s \neq p : q\#s$. The Basis S is complete if each state in S has a transition for every input in I: $\forall q \in S, i \in I : \delta(q, i) \downarrow$

2.4.2. Adding states to the Basis (Rule 1)

The algorithm uses Rule 1 to adds states from F to S, it is important to note that adding q to S also removes it from F. The Algorithm chooses an isolated state $q \in F, \forall p \in S : q\#p$ nondeterministically, and then adds them to the Basis $S' = S \cup \{q\}$ because it fulfills the requirement for states in the Basis. Because the algorithm chooses q from the Frontier F, S can remain a subtree of T. The Norm increases as the first term of the Norm has changed to $\frac{(|S|+1)(|S|+1+1)}{2} = \frac{|S|(|S|+1)}{2} + |S| + 1$. The second term can only get bigger when S grows, and the third term can shrink when a state from F is removed. Because q was apart from all states in S and we have no information about the apartness of q and the states in F. In the worst case the state q is not apart from any state in F thus $|\{(p, p') \in (S \cup \{q\}) \times (F \setminus \{q\}) | p\#p'\}| \geq |\{(p, p') \in S \times F | p\#p'\}| - |S|$the norm increases in the worst case by one. To characterize the whole Rule we write

$$isolated\, q \text{ for some } q \in F \to S \leftarrow S \cup \{q\}$$

Figure 2 showsan application of Rule 1, it shows an observation tree T of the mealy machine shown in page 6. Since the sate labled b is apart from the starting state $b\#\epsilon$ and the state labeled ϵ is the only state in the Basis, the Rule extends the Basis by adding the state b: $S' = S \cup b$, then states following b are also added to the Frontier. Nothing else changes in the application of Rule 1.

8

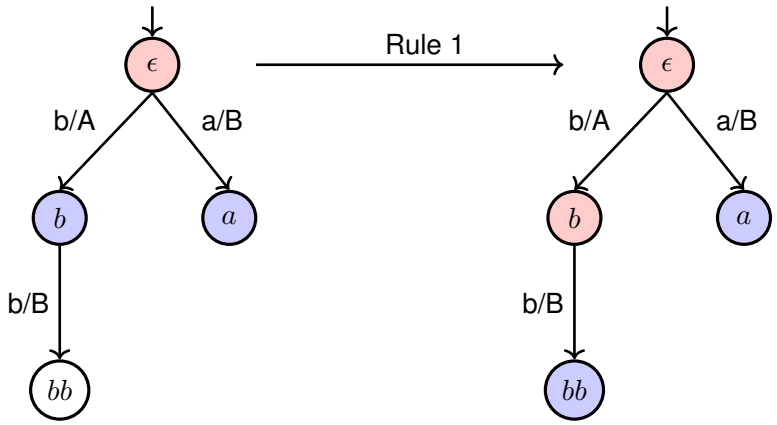


Figure 2 application of Rule 1 with Basis states marked **red** and Frontier states marked **blue**

2.4.3. Expanding the Frontier (Rule 2)

To expand the Frontier the algorithm uses Rule 2. Rule 2 chooses a Basis state $q \in S$ that where the transition for the input $i \in I$ is undefined i.e. $\delta^T(q,i) \uparrow$. Then it asks an output query with the accessor $\sigma = access(q)$ and the input: $\texttt{OutputQuery}(\sigma \cdot i)$ (where $\cdot$ denotes the concatenation of two words) to the teacher. Then the Rule adds the information, gained by the output query, to the tree, so that $\delta^T(q,i) \downarrow$. No other information is gained, as the values along the path of σ are already defined $\texttt{OutputQuery}(\sigma) = [\![q_0^T]\!](\sigma)$. This Rule only adds a state to the Frontier. Because it does not change the Basis, the first summand of the Norm remains unchanged, the second summand increases by one, because a new follow state of the Basis was introduced, and the last summand can only grow, as the apartness relationship can only grow, and no states are removed from S or F. the whole Rule can be written as:

$$\delta^T(q,i) \uparrow, \text{ for some } q \in S, i \in I \rightarrow \texttt{OutputQuery}(access(q)i)$$

Figure 2 shows an application of Rule 2. It shows that for state b the transition with input a is missing. So Rule 2 constructs the accessor $\sigma = access(b) = b$ and asks the teacher the query $\texttt{OutputQuery}(\sigma a) = \texttt{OutputQuery}(ba) = AB$. Then it extends the tree with the new information. Because the new state it is a follow state of b which is a Basis state, the Rule adds the new state to the Frontier.

2.4.4. Isolating Frontier States (Rule 3)

The algorithm now has Rules for adding states to the Basis and to the Frontier, the last two Rules aim to increase the apartness relationship. Explicitly, Rule 3 aims to identify the states in the Frontier. To extend the apartness relationship the Rule chooses a state q that is not apart from two states $r, r' \in S, r \neq r' : \neg q\#r, \neg q\#r'$. Because both states are part of the Basis they have to be apart so the algorithm chooses a witness $\sigma \vdash r\#r'$. Following that the algorithm asks the teacher the query $\texttt{OutputQuery}(access(q)\sigma)$. after this query it adds the new knowledge to the tree. Now either $\sigma \vdash q\#r$ or $\sigma \vdash q\#r'$ must hold, because $[\![r]\!](\sigma) \neq [\![r']\!](\sigma)$. Thus the third summand of the Norm increases, and the first and second summand remain unchanged, because no states are added or removed from S or F. We can

9

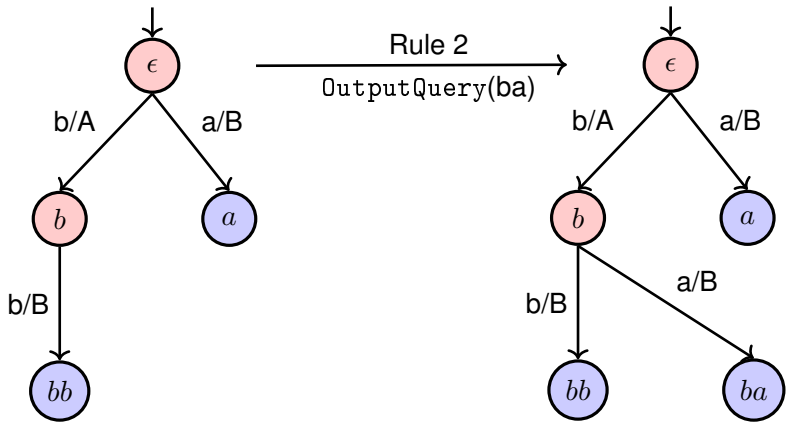


Figure 3 application of Rule 1 with states in S marked **red** and states in F marked **blue**

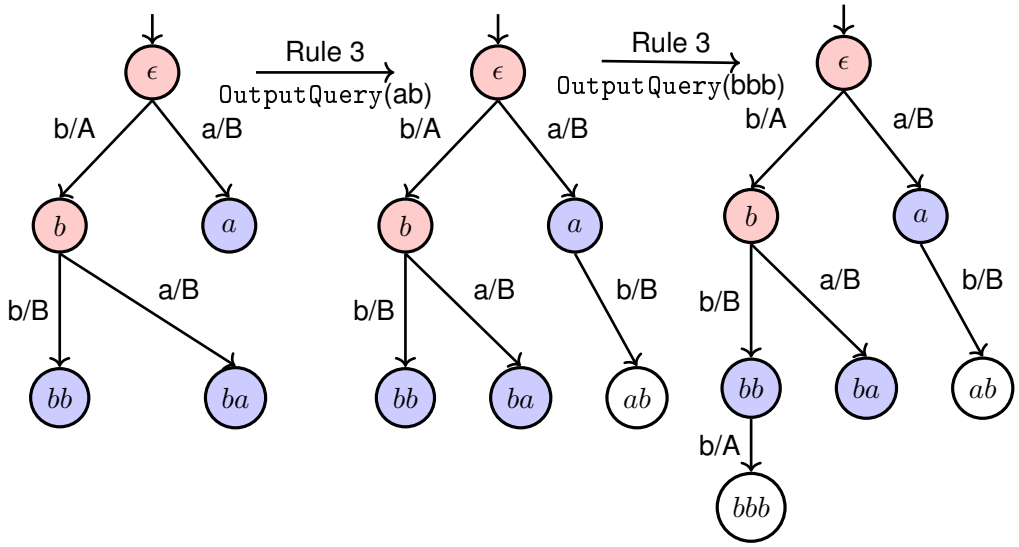


Figure 4 application of Rule 3 with states in S marked **red** and states in F marked **blue**

write the whole Rule as

$\neg(q\#r), \neg(q\#r'), \delta \vdash r\#r'$ for some $q \in F, r, r' \in S, \delta \in I^* r \neq r' \rightarrow$ `OutputQuery`$(access(q)\delta)$

Figure 4 shows the two states of the Basis $b, \epsilon \in S$ are apart with witness $b \vdash \epsilon\#b$. Each state in the Frontier is not apart from both Basis states. The algorithm now can apply Rule 3 once for every state in F. Figure 4 shows the application Rule 3 to the node a first, to achive that Rule 3 performs the query `OutputQuery`(ab) performed and the query returns BB. After this query $b \vdash a\#\epsilon$ holds. Next state bb gets chosen resulting in the query `OutputQuery`(bbb), that returns ABA. This query has expended the apartness relationship, now $bb\#b$ holds. After this the algorithm could apply Rule 3 again be chosen for the state ba but in this example it does not.

10

2.4.5. Verifying the Basis (Rule 4)

When only applying Rule 1-3 the algorithm reaches a state where no `OutputQuery` can reliably increase the norm. Thus the algorithm needs to perform an `EquivQuery` to the teacher. Since the `EquivQuery` requires a complete Mealy , the algorithm needs to construct one from the knowledge stored in the observation tree. We call the constructed Mealy Machine a Hypothesis H.
In particular The algorithm wants Hypothesis H with a functional simulation $f : T \to H$, as there is a functional simulation to the hidden Mealy Machine. We call such a hypothesis consistent. To construct our Hypothesis the algorithm takes the states from the Basis as states for the hypothesis $Q^H = S$, because the states in S correspond to distinct states in the hidden Mealy Machine. The transitions between Basis states remain the same. For transitions leaving the Basis $q \in S, p \notin S, q \xrightarrow{i/o} p$, the algorithm needs to loop them back into the Basis. Since each follow state of the Basis is a state of the Frontier, the algorithm defines a map $h : F \to S$ to define the new target of the transition making it $q \xrightarrow{i/o} h(p)$. To create a hypothesis that could be consistent $\neg f \# h(f)$ needs to hold for every $f \in F$.

Definition 8. Let T be an obseravtion tree with Basis S and Frontier F

1. A Mealy Machine H contains the Basis if $Q^H = S$ and $\delta^H(q_0^H, access(q)) = q$ for all $q \in S$
2. A hypothesis is a complete Mealy Machine H containing the Basis such that $q \xrightarrow{i/o'} p'$ in $H, (q \in S)$ and $q \xrightarrow{i/o} p$ in T imply $o = o'$ and $\neg(p \# p')$ (in T)
3. A hypothesis H is consistent if there is a functional simulation $f : T \to H$
4. For a Mealy Machine H containing the Basis, we say that an input sequence $\sigma \in I^*$ leads to a conflict if $\delta^T(q_0^T, \sigma) \# \delta^H(q_0^H, \sigma)$ (in T)

An Hypothesis can only exists, if no isolated state $p \in F$ exists, and for every Basis state $q \in S$ and input $i \in I$ the transition is defined $\delta(q, i) \downarrow$, as there does not exist a choice that satisfies Definition 9.2 otherwise.

Figure 5 shows the observation tree, and two choice functions $h : F \to S, h' : F \to S$ which leads to the twoHypothesis. Because one state is unidentified two different hypothesis are possible.
Now, the algorithm has a complete Mealy machine, so it can start an `EquivQuery`. Because long counterexamples are difficult to process Equivalence querys can be expensive. Thus the algorithm checks first if the hypothesis is consistent with the observation tree. To check for consistency the algorithm uses a procedure `CheckConsistency`. `CheckConsistency` is checking that each state existing in the tree is not apart from the state reached via the same input in the hypothesis. The procedure does this via a breadth first search on the observation tree. If any two states $q \in Q^T, p \in S$ are apart on this search, the $access(q)$ can be taken as

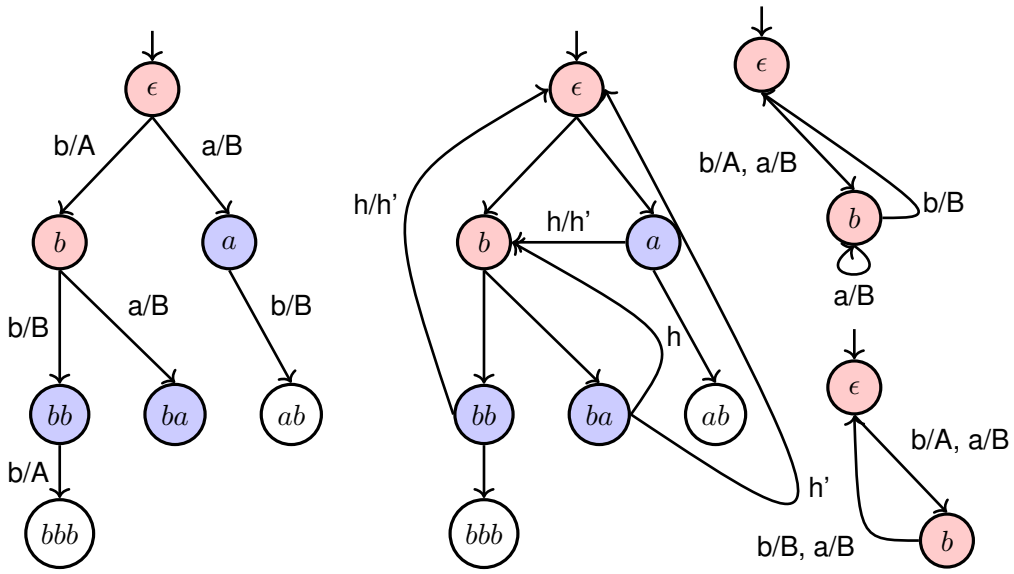


Figure 5 a observation tree (left) two possible choice functions (middle) and two hypothesis(right) one for h (above) and one for h' (below)

a counterexample four the hypothesis. The specific procedure can be seen in 1

Figure 6 shows an observation tree that could be produced for the Mealy Machine shown in Figure 1, when calling `CheckConsistency`(H) it will return (No, aba) as $aba\#b$ and $\delta^H(q_0, aba) = b$.

If consistency checking returns yes the algorithm needs to perform the `EquivQuery`(H). The answer will either be yes, leading to termination and H being the learned Mealy Machine, or (no, cex) where cex is a counterexample such that `OutputQuery`$(cex) \neq [\![q_0^H]\!](cex)$. Since the length of the counterexample is unknown, the learner needs to process it, to guarantee extension of the apartness relationship.

To process counterexamples the algorithm uses a binary search approach cutting the counterexample in half every iteration. The algorithm uses `ProcCounterEx`with the goal that `ProcCounterEx`(σ) changes the tree in such a way, that H can never be a Hypothesis for T again. For example, the algorithm creates the hypothesis shown in Figure 5, `EquivQuery`(H) could return any string $(bb)^n ba$ to classify the same issue. If the algorithm adds a long counterexample it could extend the Observation Tree with new information but not expand the apartness relationship. Thus the `ProcCounterEx`needs to find the list of inputs that leads

Algorithm 1 check if hypothesis H is consistent with observation tree T

procedure CHECKCONSISTENCY(H)
  Q $\leftarrow$ **new** queue $\subseteq Q^T \times S$
  $enqueue(Q, (q_0^T, q_0^H))$
  while $(q, r) \leftarrow dequeue(Q)$ **do**
    if $q\#r$ **then return** no: $access(q)$
    for all $q \xrightarrow{i/o} p$ in T **do**
      $enqueue(Q, (p, (\delta^H(r, i))))$
  return yes

12

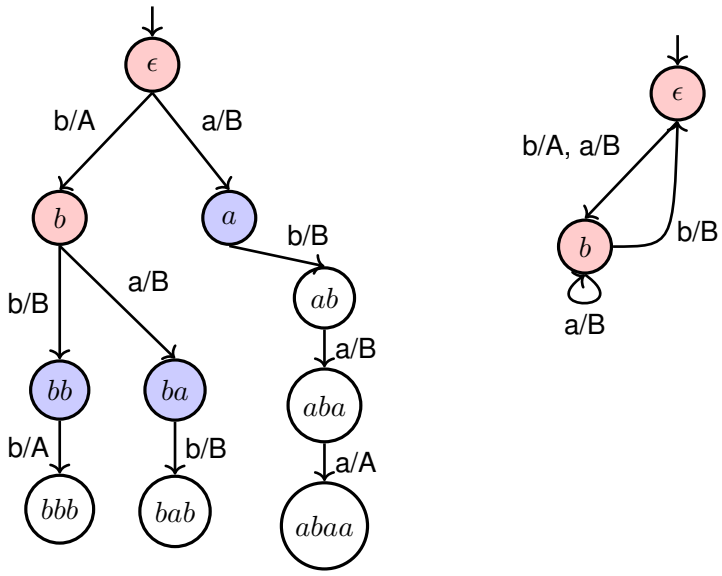


Figure 6 A observation tree leading to a conflict when checking consistency for the observation tree on the right

to this conflict. Since $\delta^H(q_0^H,(bb)^n)\#\delta^T(q_0^T,(bb)^n)$,`ProcCounterEx` moves this apartness closer to the Frontier. The exact counterexample processing algorithm is complex and not needed to understand the formalization later on so it will not be explained further here.

After processing a counterexample, the algorithm knows that a Frontier state is apart from one more Basis state than before. This increases the third summand of the norm. Because the Basis has not changed, the first summand remains the same, and because of the assumption no transition out of the Basis is undefined, the second summand can not change either.

2.4.6. Simplification of Rule 4

Rule 4 has proven to be the hardest version to verify formally, thus we propose a simpler version that accomplishes the same. For this simplified versionwe introduce a new type of query:

Definition 9. Expanding Definition 2.6 the teacher can also answer the following query:

$$\texttt{DifferenceQuery}(p,q):\ \text{the teacher replies with yes if:} \neg\delta^M(q_0,p)\#^M\delta^M(q_0,q)$$
$$\text{and no and a counterexample } \sigma\in I^* \text{ with } \sigma\vdash\delta^M(q_0,p)\#^M\delta^M(q_0,q) \text{ otherwise.}$$

Note that we are referring to specific nodes in the hidden Mealy Machine through an input sequence. Using $\texttt{DifferenceQuery}(access(q),access(p))$ the algorithm can be sure that for a functional simulation $f:T\to M$ the states reached via functional simulation are semantically equal to the state reached through the accessor: $f(q)\approx\delta^M(q_0,access(q))$. This way

13

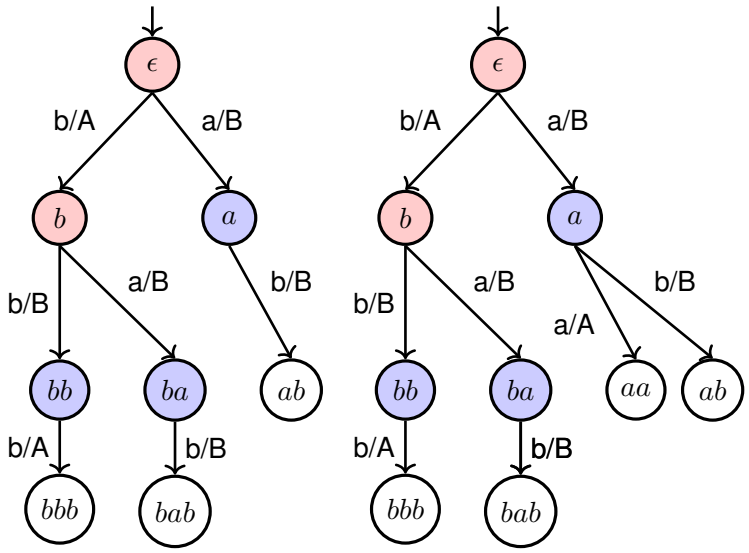


Figure 7 Application of the simplified Rule 4

the algorithm does not need to produce a hypothesis and can check two states against each other directly.

Since the algorithm did not create an hypothesis, no consistency checking is possible, leading to further simplification of the Rule . Thus the algorithm needs to ask `DifferenceQuery`'s every time it wants to apply Rule 4. Each application of the simplified Rule 4 needs up to $(|S| * |F|)$ queries, because every state of the Frontier needs to be checked against every state from the Basis. If every state in the Frontier is identified, it only needs $|F|$ queries, because all tuples $(s, f) \in S \times F$ can be ignored. For this nondeterministic approach the algorithm can assume a counterexample and does not need specific searching. If no queries can answer with No, the algorithm now knows that it can only construct H that are equal to the hidden mealy machine $H \approx M$. If any $\texttt{DifferenceQuery}(q, p)$ answers with (no, σ) the algorithm performs two output querys $\texttt{OutputQuery}(p\sigma)$ and $\texttt{OutputQuery}(q\sigma)$,after which $\sigma \vdash \delta^T(q_0^T, q)\#^T\delta^T(q_0^T, p)$ holds. This means the algorithm does not need to process the counterexamples found.

For example, Figure 7 shows the simplified Rule 4 applied to the observation tree that is generated by applying Rule 3 to the observation tree found in Figure 4. Now the algorithm compares every state $q \in F$ to the states $p \in S$ where $q\#p$. Then it asks the queries $\texttt{DifferenceQuery}(access(bb), access(\epsilon)), \texttt{DifferenceQuery}(access(ba), acess(b))$ and $\texttt{DifferenceQuery}(access(a), access(b))$. If all of those queries returned yes the algorithm could construct a Hypothesis H where $H \approx M$. When the teacher controls the hidden mealy machine shown in Figure 1, $\texttt{DifferenceQuery}(access(ba), acess(b))$ and $\texttt{DifferenceQuery}(access(a), access(b))$ would return $no : cex$ where the shortest $cex = a$ and any other counterexample would decompose into au where $u \in I^*$. Longer counterexamples are possible for more complex hidden s, but for this approach the length of counterexample does not lead to more queries being needed to solve them.

Figure 7 shows $\texttt{DifferenceQuery}(access(a), access(b))$ being asked first and returns $no : a$ as a reply, then it shows the tree generated when the algorithm asks `OutputQuery`(aa) and

`OutputQuery`(ba), the second query does not change anything, but the first one adds a state leading to $a \vdash a\#b$. Since state a is now isolated and the algorithm can not continue performing Rule 4.

15

3. Formalization

Based on the inner workings of the L# algorithm the following chapter covers the formalization. Theorem provers are used to verify formal proofs, a interactive theorem prover lets us write the formal proof in a programming like fashion. We choose Isabelle/HOL for this thesis.

3.1. Definitions

The first Basis is the formalization of the concepts used in the algorithm.

3.1.1. Mealy Machines

For the formalization we differentiate between complete Mealy Machines and observation trees, because it reduces the amount of information needed to classify the complete Mealy. Here we describe the complete Mealy Machine first. We use a combined output transition function similar to $\langle \lambda, \delta \rangle : Q \times I \rightharpoonup O \times Q$ in the mealy machine.

```
type_synonym ('s,'in,'out) trans = "(('s × 'in) ⇒ ('s × 'out))"
```

Next we define an equivalent for Mealy Machines $M = (Q, q_0, I, O, \delta, \lambda)$ using the trans type as the transition output function. Furthermore the sets of states Q, inputs I and outputs $O$are types rather than explicitly adding them to the Mealy Machine constructor. This enforces that any transition is set on the bounds of the defined types, otherwise some invariant would be necessary to show that no transition from the set of states can leave the predefined set. Instantiating a mealy machine only requires q_0 and a transition output function.

```
type_synonym ('s,'in,'out) mealy = "'s × ('s,'in,'out) trans"
```

Performing a run through the Mealy Machine with a word of length n requires a function similar to $\langle \lambda_n, \delta_n \rangle$. The function *run_mealy* takes a transition function, a state where the run starts and the input sequence of a run, it then recursively adds an output to the list and returns the output list as well as the state where the run ends.

```
fun run_mealy :: "('s,'in,'out) trans ⇒ 's ⇒ 'in list ⇒ ('s × 'out list)" where
"run_mealy f q [] = (q, [])" |
"run_mealy f q (w # ws) = (let (st,op) = f (q,w) in
  (let (q',x) = run_mealy f st ws in (q',op # x)))"
```

For easier reading in more complex proofs we define *outs_run_Mealy* and *state_run_Mealy* as the functions extracting the output or the reached state from *run_Mealy*. Since *outs_run_-Mealy f q* is equivalent to $[\![q]\!]$ we can also define the equality of states similar to the definition in Chapter 2, namely that two states are equal if they produce the same output for every input.

16

Since states can be from different Mealy Machines however we need to specify witch Mealy Machine a state belongs to. The function *eq_mealy* takes two states and a mealy machine that they belong to and returns $True$ iff both are equal.

definition *eq_mealy :: "('s,'in,'out) mealy ⇒ 's ⇒ ('s2,'in,'out) mealy ⇒ 's2 ⇒ bool"* **where**
"eq_mealy a q b p ≡ (case (a,b) of
((q_0,f),(p_0,g)) ⇒ (∀ is. outs_run_mealy f q is = outs_run_mealy g p is))"

The equality of two Mealy Machines is the equality of the two starting states. The *equal* abbreviation takes two mealy machines and compares the starting states, returning true iff those are equal.

abbreviation *equal :: "('s,'in,'out) mealy ⇒ ('s2,'in,'out) mealy ⇒ bool"* (**infixr***"≈" 80)* **where**
"a ≈ b ≡ (case (a,b) of
((q_0,f),(p_0,g)) ⇒ eq_mealy a q_0 b p_0)"

3.1.2. Observation Trees

After we have defined the hidden Mealy Machine and its functions, we now define the Observation tree. For partial Mealy Machines in a tree shape, we use a datatype that contains a partial function to the following nodes and output. Each node keeps track only of its own followers. This simplifies adding nodes to the transition function. Furthermore the algorithm only needs to keep track of the root node of the tree, traversing it for any operation needed. We will refer to specific nodes by their accessors because the datatype *otree* only contains information about the next transitions. The partial function in Isabelle is realized via the option datatype, where any transition can either contain *None* referring to an undefined transition or *Some (node,output)* where *node* is the next node reached and *output* is the output value.

datatype *('in,'out) otree = Node "'in ⇒ (('in,'out) otree × 'out) option"*

Similar to *run_mealy* and $\langle \lambda_n, \delta_n \rangle$ we define the function *run* to traverse the observation tree. *run* will traverse the tree along the input list edges, and return *None* if any edge along the path would be undefined:

fun *run :: "('in,'out) otree ⇒ 'in list ⇒ (('in,'out) otree × 'out list) option"* **where**
"run ot [] = Some (ot, [])" |
"run (Node t) (i # is) = (case t i of
Some (n,op) ⇒ (case (run n is) of
Some (ot,ops) ⇒ Some (ot,op # ops) |
None ⇒ None) |
None ⇒ None)"

To simplify reading of more complex proofs we define *out_run* as the output function, however not via *run* but similar to the *run* definition with only the output being returned. This brings the

benefit of making inductions easier for proofs where we use *out_run*.
To argue about the effectivness of the algorithm a definition of apartness is also necessary. The definition is similar to Definition 5, and because *out_run q* is equivalent to $[\![q]\!]$ we replace one with the other in our formal definition. For apartness we refer to each state by the input sequence that is the unique accessor for that state, so the *apart* function takes an observation tree, as Basis for comparing the two nodes and two input lists that refer to a node. It returns *False* if from both nodes there is a sequence of inputs that leads to different outputs and *True* otherwise.

```
fun apart :: "('in,'out) otree ⇒ 'in list ⇒ 'in list ⇒ bool" where
"apart q_0 t1 t2 = (∃ i x y. outs_run q_0 (t1 @ i) = Some x ∧
    outs_run q_0 (t2 @ i) = Some y ∧
    drop (length t1) x ≠ drop (length (t2)) y)"
```

In some cases we need to refer a witness of two apart states, for this we describe the function *apart_witness*. The definition is very similar to the one of *apart* just with i being moved from the existence quantor to the inputs of the function.
Following the original definition we also define a functional simulation, but the proof will use it to a lesser extend, it is not part of the invariant, but the invariant we will define dictates that a functional simulation must exist. For the observation tree a functional simulation is defined as a function from an input list to a Mealy Machine state. Furthermore, the definition *func_sim* classifiesa function f as a functional simulation for Mealy Machine m and observation tree T if it returns *True*. Because states in the observation tree are labeled via input strings, the starting state of any observation tree is $[]$. Otherwise the definition is similar to Definition 4.

```
definition func_sim :: "('s,'in,'out) mealy ⇒ ('in,'out) otree ⇒ ('in list ⇒ 's)
⇒ bool" where
"func_sim m T f = (case m of
      run_mealy t (f acc) is = (f (acc @ is),(drop (length acc) ops)))))"
```

3.1.3. Further Prerequisites

After both the Mealy Machine and the Observation tree are properly defined in the formalization, some other helpful functions are needed to perform a run of the algorithm. First we will refer to any output query via a wrapper function *output_query* expecting a Mealy Machine M and a list of inputs. This function only calls *outs_run_Mealy* internally, we do this for readability.
When the algorithm asks an *output_query* it needs to add the new information to the existing tree, the function *process_output_query* is responsible for this job. the function *process_output_query* takes a root node T, one input list as well as an output list of the same length, it traverses the tree along the input edges, and defines any edge that is undefined with the corresponding output. New nodes created by *process_output_query* start of as nodes with no transitions. Furthermore we note that *process_output_query* is not defined for two lists that don't have the same length. At the end a new root node is returned.

```
fun process_output_query :: "('in,'out) otree ⇒ 'in  list ⇒
'out list ⇒ ('in,'out) otree" where
"process_output_query q [] [] = q" |
"process_output_query (Node t) (i # is) (op # ops) = (case t i of
    None ⇒ (Node (λ j. if j = i
      then Some (process_output_query (Node (λ k. None)) is ops,op)
      else t j)) |
    Some (tree,out) ⇒ (Node (λ j. if j = i
      then Some ((process_output_query tree is ops),out)
      else t j)))"
```

To keep track of the current state of the algorithm we define a datatpye *state*. The state contains the Basis, the Frontier and the root node of the observation tree. the Basis and Frontier are defined as *'in list set*'s where each *'in list* refers to one node. Later we see *(S,F,T)* to classify a state most of the time.
We have defined the Frontier before as all the follow states of S in the explanation of the algorithm. To tack specific changes, or if no changes need to be made we want to change the Frontier explicitly. Tracking explicit changes of the Frontier in Rule 1 is complicated, because a state gets added to the Basis, and no information about the follow states is given. Thus we define the *frontier* function that returns a Frontier set based on the input of the current state.

```
fun frontier :: "('in,'out) state ⇒ 'in list set" where
"frontier (S,F,T) = {f.((∃ s ∈ S. ∃ i. f = s @ [i]) ∧ f ∉ S ∧ outs_run T f ≠ None)}"
```

To argue about termination and number of Rules applied we also need to define the norm. We split the Norm into three different sections, called *norm1*, *norm2* and *norm3* each function takes a state as an input and corresponds to a summand in the original norm. To build a whole Norm we define a function *norm* adding the three Norm summands together.
To create a complete Mealy Machine out of the observation tree, a Hypothesis is needed. For the purposes of verifying the algorithm we classify if a Mealy Machine $([], t)$ is a possible hypothesis for the state of our algorithm. A function *hypothesis* takes the state and a transition function t as input. A transition function t can be a hypothesis if for all nodes in the Basis the output for any transition is the same, all transitions between Basis states remain the same and any transition leaving the Basis to a state f leads to a state $q \in S$ and $\neg(f \# q)$.

```
fun hypothesis :: "('in ,'out) state ⇒ ('in list,'in,'out) trans ⇒ bool" where
"hypothesis (S,F,T) t =
    (∀ s ∈ S. ∀ i. ∃ tran op n out. (run T s = Some (Node tran,op)) ∧
      (tran i = Some (n,out)) ∧
      (if (s @ [i]) ∈ S
        then t (s,i) = (s @ [i],out)
        else (∃ y ∈ S. ¬ apart T y (s @ [i]) ∧ t (s,i) = (y,out))))"
```


19

3.1.4. The Locale

For the whole proof we use a locale to fix different variables. We do this to define behavior that is hard to describe in function form, and to make some parts of the proof more readable. First we fix a Mealy Machine M, we implicitly fix the types *'s'*, *'in* and *'out* for the proof as well. The input type *'in* and state type *'s* are required to be finite, as the algorithm never terminates for some infinite Mealy Machines. Next we fix the set of inputs *I*, we assume that *I* represents the universe of the input type, the set of states *Q* is also fixed in this way. These two sets are fixed for readability as writing *I* is easier to understand than *(UNIV::'in set)*.
The proposed difference query is also defined in the locale, because the behaiviour is complex and concrete implementation is not needed to prove the existing algorithm. The described difference query answers with *None* for any two input lists s and fs there does not exist an input x where the two states reached by s and fs produce different outputs, i.e. $\sigma^M(q_0, s) \approx \sigma^M(q_0, s)$. However if the difference query answers with *Some x* then x has to be a counterexample so that $x \vdash \sigma^M(q_0, s) \# \sigma^M(q_0, s)$. Both of those classifications are defined only via *outs_run_Mealy*, an only comparing the output not produced by s and fs.

```
locale Mealy =
  fixes M :: "('s :: finite,'in :: finite,'out) mealy" and
    I :: "'in set" and
    Q :: "'s set" and
    difference_query :: "('s,'in,'out) mealy ⇒ 'in list ⇒ 'in list ⇒
     'in list option"
  assumes
  univI: "I = UNIV" and
  univQ: "Q = UNIV" and
  difference_query_def: "(difference_query (q_0,f) s fs = Some x) ⟶
    (drop (length s) (outs_run_mealy f q_0 (s @ x)) ≠
      drop (length fs) (outs_run_mealy f q_0 (fs @ x)))" and
  difference_query_def_none: "(difference_query (q_0,f) s fs = None) ⟷
    (∄ x.(drop (length s) (outs_run_mealy f q_0 (s @ x)) ≠
      drop (length fs) (outs_run_mealy f q_0 (fs @ x))))"
begin
```

3.1.5. The Invariant

Since the algorithm is complex and we make many assumptions about the state of the algorithm, we define a specific invariant that is upheld throughout the execution of the algorithm. The Invariant holds if:

- No node can be in the Frontier and in the Basis at the same time
- For all accessors in the Basis the run sequence must be defined
- The Frontier and the Basis are finite sets
- Each two items in the Basis are pairwise apart (we write *sapart (S,F,T)*)
- All output sequences the tree can produce must be the same as a output query with the

same input would produce

- The Frontier is always equal to the defined *frontier* function
- The empty list is in the Basis and any state in the Basis that is not the empty list has to be a state following another state of the Basis

Because the invariant is a definition two extra lemata are useful for proofs involving *invar*, one describes how to verify if a state is consistent with the invariant, and one describes what we know about any state that fulfills the invariant. In isabelle we define the Invariant as a definition *invar*

```
definition invar :: "('in,'out) state ⇒ bool" where
"invar st = (case st of
    (S,F,T) ⇒
    (∀ e. ¬ (e ∈ S ∧ e ∈ F)) ∧
    (∀ e ∈ S. outs_run T e ≠ None) ∧
    finite S ∧ finite F ∧
    sapart (S,F,T) ∧
    (∀ i. outs_run T i ≠ None ⟶ outs_run T i = Some (output_query M i)) ∧
    (F = frontier (S,F,T)) ∧
    [] ∈ S ∧ (∀ s ∈ S. s = [] ∨ (∃ s2 ∈ S. ∃ i. s2 @ [i] = s)))"
```

3.1.6. The Algorithm

For verification purposes we describe the algorithm as a transition system, i.e. we only describe when a step from the algorithm is possible. For that we describe a inductive *algo_step*, that is split in four Rules that correspond to the Rules described before. We use an inductive for the algorithm descriptions as it matches well with the nondeterministic approach described in chapter 2, and because we don't need to define functions that return certain inputs we need for each Rule . For example we define *Rule 1*, as: there is a transition from one state (S, F, T) to another state where a node f is added to the Basis and the Frontier being updated, if f is isolated.

```
rule1: "⟦f ∈ F; ∀ s ∈ S. apart T s f⟧ ⟹
    algo_step (S,F,T) (S ∪ {f},frontier (S ∪ {f},F,T),T)" |
```

For the transition of the second Rule we write that a transition from (S, F, T) to $(S, F \cup \{s@[i]\}, process\_output\_query\ T\ (s@[i])\ out)$ is possible if s is a Basis state and $s@[i]$ is not defined on the observation tree. Then the algorithm performs an output query with the input $s@[i]$ that returns out. Here we could also write $process\_output\_query\ T\ (s@[i])\ (output\_query\ M\ (s@[i]$ but it would overcrowd the right side of the Rule .

```
rule2: "⟦s ∈ S; (outs_run T (s @ [i]) = None);
      output_query M (s @ [i]) = out⟧ ⟹
    algo_step (S,F,T) (S,F ∪ {s @ [i]},process_output_query T (s @ [i]) out)" |
```


21

The definition of Rule 3 in Isabelle entails two different Basis states $s1 \neq s2, s1, s2 \in S$ and a Frontier state $f \in F$. For Rule 3 to be applicable f needs to be apart from both $s1$ and $s2$. The algorithm then takes any witness $w \vdash s1 \# s2$ and obtains the result out from the output query with input $f@w$. now a transition from (S, F, T) to $(S, F, process\_output\_query\ T\ (f@w)\ out)$ is possible, and f is apart from at least one of the two states from the Basis.

```
rule3: "⟦s1 ∈ S; s2 ∈ S; s1 ≠ s2; f ∈ F;
      ¬ apart T f s1;
      ¬ apart T f s2;
      apart_witness T s1 s2 w;
      output_query M (f @ w) = out⟧ ⟹
    algo_step (S,F,T) (S,F,process_output_query T (f @ w) out)" |
```

The formalization of Rule 4 requires two calls of *process_output_query*, to make sure that the counterexample given by a difference query is defined for both states, so the transition from (S, F, T) to
$(S, F, process\_output\_query\ (process\_output\_query\ T\ (s@inp)\ outs)\ (fs@inp)\ outf)$ is possible if:

- all transitions coming from a state in the Basis are defined
- no state in the Frontier is isolated
- fs is a state in the Frontier
- s is a state in the Basis
- s and fs are not apart
- the difference query for s and fs returns a counterexample inp
- $outs$ and $outf$ are the results of output queries for input $s@inp$ and $fs@inp$ respectively

Following this description the whole *Rule 4* looks like this:

```
rule4: "⟦∀ s1 ∈ S. ∀ i. outs_run T (s1 @ [i]) ≠ None;
      ∀ f1 ∈ F. ¬ (isolated T S f1);
      fs ∈ F;
      s ∈ S;
      ¬ apart T s fs;
      difference_query M s fs = Some inp;
      output_query M (s @ inp) = outs;
      output_query M (fs @ inp) = outf⟧ ⟹
    algo_step (S,F,T) (S,F,process_output_query
     (process_output_query T (s @ inp) outs) (fs @ inp) outf)"
```


22

3.2. Proof

After we have defined all important functions, we will focus on the lemmata and theorems that are proven in the formalization. Since the formalization required a lot of small lemmata the following chapters will focus on the most important ones and the intuition behind why and how the proofs where conducted. Before inspecting any lemmas in particular we formulate the goal. We want to prove that the transition system *algo_step* leads to a hypothesis that is equal to the Mealy Machine M, we also want to prove that this is possible in a polynomial amount of Rule applications. With these goals in mind the next sections will explain the important lemmata needed to formalize the proof.

3.2.1. Retaining the invariant

The invariant is crucial to describe the behaiviour of the observation tree, for proofs using a state of the algorithm, one assumption will usually be that the state fulfills the requirements for the invariant. To use this invariant throughout the algorithm we need to proof that a step will always retain the invariant. This is why we prove lemma *algo_step_keeps_invar*. This lemma assumes that a state (S, F, T) is consistent with the invariant and there is a step from (S, F, T) to (S', F', T'). using those assumptions we need to show that the invariant holds for (S', F', T').

```
lemma algo_step_keeps_invar:
  assumes "algo_step (S,F,T) (S',F',T')" and
    "invar (S,F,T)"
  shows "invar (S',F',T')"
```

The proof of this lemma focusses on showing every requirement for every possible Rule application seperatley, since if a step is possible one of the four Rules has to be applied. For most requirements this is trivial, butthe conditions relying on specific *outs_run* behaiviour especially that all output sequences the tree produces are the same as the output query, are more changeling.
One of the most important lemmas used to argue about different *run_output* results is *output_query_retains_some_specific_output* wich argues that all sequences that are defined on a tree give the same output before and after an output query is processed.The lemma assumes that we use two lists of the same length ip and op and that for a tree *Node* r the sequence acc is defined. it then shows that the result of *outs_run* is the same for the node and the node that has processed an output query with ip as input and op as output for sequence acc.

```
lemma output_query_retains_some_specific:
  assumes "length ip = length op" and
    "outs_run (Node r) acc = Some (out1)"
  shows "outs_run (Node r) acc = outs_run (process_output_query (Node r) ip op) acc"
```

The proof of this lemma uses an induction on the sequence acc. The base case equality is

23

easy to prove it, extending the sequence by one letter $acc' = a \cdot acc$ is more difficult. To prove the induction step the proof differentiatesn ip and op being empty or containing at least one symbol. The empty case is once again trivial, in the other case the proof checks if the first letter of ip and a are equal. If both letters are equal, the proof transforms the goal term to a term only dependent on the output of the Node function with input a: $r(a)$ andthe input ip and output op for *process_output_query* without the first letter, then use the induction hypothesis to show equality, otherwise the *process_output_query* does not change anything about the output.
We use this lemma in the proof of *algo_step_keeps_invar*. For two states (S, F, T) and (S, F', T') where T' is only different from T through applying *process_output_query* functions. We can now show that the invariant requirements involve *outs_run_mealy* still hold for T' on the inputs already defined for T. For example in the Rules 2-4 the Basis does not change, so the proofs shows the requirement, that every Basis state is defined, with *output_query_retains_some_specific_output*. For the requirement that all output sequences are equal to the output query, the proof uses *output_query_retains_some_specific_output* to show that all values that were defined on the original tree T are still equal to the *output_query*. To prove that goal, we also need information about the transitions *process_output_query* adds, so we need another lemma.
We define a auxiliary lemma*output_query_keeps_invar_aux* that shows that any input sequence not defined for a tree T will fall in line with the requirement after processing an output query.

```
lemma output_query_keeps_invar_aux:
  assumes "outs_run T i = None" and
    "outs_run_mealy t q_0 j = out" and
    "T' = process_output_query T j out" and
    "∀ k y. outs_run T k = Some y ⟶ outs_run_mealy t q_0 k = y"
  shows "outs_run T' i ≠ None ⟶ outs_run T' i = Some (outs_run_mealy t q_0 i)"
```

The proof of *process_output_query* in isabelle is quite difficult even if the lemma would be easy to do on paper. The main idea of the proof is an induction on the input that was undefined before. Performing the induction step is rather difficult, because to apply the induction hypothesis the proof needs to make some case distinctions. First the proof shows that any sequence that still produces *None* does not need to be considerd, since $False \implies x$ holds for any x. Then for the case that the sequences $is = a \cdot i$ does not produce none, the proof needs to consider the first step in the original tree, if $T = Node\ f$ the proof considers two cases if $f\ a = Some\ c$ the proof reformulates the goal case in a way that allows it to apply the inudction hypothesis. If $f\ a = None$ even further case distinctions are needed, specifically only the case that i is not empty is interesting for the proof, where it uses the induction hypothesis once again, with a empty node.
The requirement of eqaulity to *output_query* for *algo_step_keeps_invar* now gets solved thorugh applying the new lemma. the proof now shows every requirement for every Rule and thus is concluded.

24

3.2.2. Termination and runtime via norm

To argue about termination we will show that every Rule in the transition system increases the norm, and that the Norm is bound. To show that the Norm increases, we use the theorem *algo_step_increases_norm*. This lemma assumes that there is a step possible from (S, F, T) to (S', F', T') and that the invariant holds for (S, F, T). The conclusion of this lemma is that the Norm of the first state is smaller than the Norm of the second state.

```
theorem algo_step_increases_norm:
  assumes "algo_step (S,F,T) (S',F',T')" and
    "invar (S,F,T)"
  shows "norm (S,F,T) < norm (S',F',T')"
```

To show this, we use a case distinction to argue about each Rule separately. For Rules two to four we show that one part of the Norm is strictly bigger on the new state than on the old and for the others we show that it is at least as big. Rule 1 is once again a special case where the first norm increases and the last one decreases.
To argue about cardinalities the proof uses the *card_mono* lemma from the standard library, that says if $X \subseteq X'$ and X' is finite then $|X| \leq |X'|$ for any two stets S and S'. The proof shows that any set on the right side is finite through bulidng an overapproximation that is finite and showing a subset relationship. To proof subset relationships for the norms, both sets are compared very detailed showing similarities or building a subset chain. To argue about the last three Rules two lemmata are important, *output_query_retains_some_specific_output* is used to show that *process_output_query* does not change existing transitions, and *process_op_query_not_none* is used for new transitions. Here the Proof uses the first lemma to sow subset relationships and the second one to show that a new item is in the sets of norm2 or norm3.
The lemma *process_op_query_not_none* shows that if an output query has been proceeded for a input sequence ip than that sequence can not be undefined for the tree.

```
lemma process_op_query_not_none_output:
  assumes "length ip = length op"
  shows "outs_run (process_output_query (Node t) ip op) ip ≠ None"
```

This lemma, once again requires and inducton, we have only one viable target with the input sequence. Since ip is used for both *process_output_query* and *run* the proof can use the hypothesis after considering the possibility of the first transition being undefined.
Using this lemma for the Norm is straigth forward, e.g. in Rule 2 we can show that the follow transition previously undefined now has a definition increasing *norm2*, for Rule 3 the existence of a defined transition is enougth to proof that the apartness relationship has expanded, and in Rule 4 we use the invariant, i.e. the property that all defined outputs are equal to the ouptut query (thanks to *algo_step_keeps_invar* also for the new tree) in tandem with both transitions being defined to show increase.

The property that the Norm increases shows progress, but to show Termination we need to

25

provide a upper bond on the Norm. The function *norm_max* is the maximum possible Norm for any Basis S:

abbreviation *max_norm* **where**
*"max_norm S ≡ (card S * (card S + 1)) div 2 + card S * card I + card S * (card S * card I)"*

The proof that *max_Norm S* is bigger or equal than *Norm (S,F,T)* for any F, T if the invariant holds is important, and in isabelle works through giving supersets for the sets used in *norm2* and *norm3*. Showing that the Norm is bound can now be done by showing that the number of states in the Basis is bound.

lemma *max_size_S_aux:*
  assumes *"func_sim M T f"* **and**
    "sapart (S,F,T)" **and**
    "finite S"
  shows *"card S ≤ card Q"*

The lemma uses the assumption that a functional simulation exists, this is possible, because there exists a functional simulation for any state that is consistent with the invariant. Now it uses a contraposition argument, assuming that there are more states in S than in Q. Thentotwo states $s1, s2 \in S$ must exist that map to the same state with f $f\ s1 = f\ s2$, but since every two states are apart $s1 \# s2$ they can not map to the same state in a functional simulation, leading to a contradiction.
This upper bound for S is less accurate than the bound proposed by Vandraager et.al [19], since Vandraagers Norm uses the number of Equivalence classes of the hidden Mealy Machine. We chose the number of states since it is easier to formalize, and gives an appropriate bound, still relying on the size of any hidden Mealy Machine.
Using these two lemmata, we now know the maximum Norm is $max\_norm\ Q$ and because every step of *algo_step* increases the norm, the algorithm has to terminate after $max\_norm\ Q$ steps.

3.2.3. Correctness

To show corectness of the algorithm we show that the transition system terminates in a state, where any hypothesis is equal to the hidden Mealy Machine. To show correctness of a whole run we define a starting point for the algorithm $(\{[]\}, \{\}, (Node\ Map.empty))$ where the Basis only contains the starting node, the Frontier is the empty set and the tree only contains the starting node. Furthermore, show a run from start to finish we need a way of concatenating steps. So we define*algo_steps* as the concatenation from *algo_step* with itself. Now we can define our goal, we want to obtain a state (S, F, T) that is reachable via *algo_steps* from the starting configuration, where the algorithm can not perform further steps. We will call a state where no further step can be taken a final state. We then want to show that there both esxist an hypothesis and that any hypothesis is equal to the hidden Mealy Mealy machine.

corollary *no_step_mealy_equal2:*

26

```
  assumes "algo_steps ({[]},{},Node Map.empty) (S,F,T)" and
    "¬ (∃ S' F' T'. algo_step (S,F,T) (S',F',T'))"
  shows "(∃ t. hypothesis (S,F,T) t) ∧(∀ t. hypothesis (S,F,T) t ⟶ M ≈ ([],t))"
```

The proof of final states require them to fulfill the invariant, since we have proven that *algo_step* retains the invariant, the proof now need to show that the starting state also fulfills the invariant. Then we can argue, that any final state that fulfills the invariant has a hypothesis, and any hypothesis for a final state that fulfills the invariant is equal to the hidden Mealy Machine.
Beacuse the empty state fulfills the invariant is trivial we focus on the other two lemmata. First we will proof the lemma that a hypothesis exists, to define the lemma we use the knowledge from Definition 8 that a hypothesis exists if every transition from a Basis state is defined and no Frontier state is isolated, we use those two as assumptions, this holds for final state, since a application of Rule 1 or Rule 2 would be possible otherwise.

```
lemma exists_hypothesis:
  assumes "invar (S,F,T)" and
    "¬ (∃ f ∈ F. ∀ s ∈ S. apart T s f)" and
    "¬ (∃ s ∈ S. EX i. (outs_run T (s @ [i]) = None))"
  shows "∃ t. hypothesis (S,F,T) t"
```

Now this proof that a hypothesis function exists, uses an example. To build a hypothesis, the proof uses help function f , f will take a input list i and return i if it is a Basis state, and a Basis state that is not apart from i if $i \in F$, and is not defined in any specific way if the state is outside of the Basis and the Frontier. This works, because every state in the Frontier is not apart from at least one state in the Basis. Now the proof describes a function that will be a hypothesis, this function takes a input list s as state and a single input letter i, it will return a dummy value if either s or the transition from s with i is undefined. If both are defined, it returns the output from the transition from s with i as well as the result of the function f $(s@[i])$ as the next state. Here is how the function looks in Isabelle, the output $none$ can be any possible output, it is not relevant for the hypothesis function.

```
t = (λ (s,i). (case run T s of
    None ⇒ ([],none) |
    Some (Node tran,op) ⇒ (case tran i of
      None ⇒ ([],none) |
      Some (n,out) ⇒ (f (s @ [i]),out))))
```

Since a hypothesis only needs to work on the Basis states and all outgoing transitions and we assumed that all of those are defined, the proof now can show that t is a hypothesis for the state (S, F, T).
Now that we have proven that a hypothesis exists, the next step is to prove that this hypothesis is equal to a Mealy Machine M. For this we take the assumptions that (S, F, T) is a final state that fulfills the invariant and t is a hypothesis function for the state:

27

```
theorem no_step_mealy_equal:
  assumes "invar (S,F,T)" and
    "¬ (∃ S' F' T'. algo_step (S,F,T) (S',F',T'))" and
    "hypothesis (S,F,T) t"
  shows "M ≈ ([],t)"
```

The proof of this theorem is crucial, but since no induction is possible, we need an auxiliary lemma. The auxiliary lemma proves the equality of all *outs_run_Mealy* applications to one state of the Mealy Machine $q \in Q$ and one of the hypothesis p.It uses different assumptions so that the induction can work, one that the list p is a Basis state because the hypothesis only works on Basis states, and one that shows the equality of both sates in the Mealy Machine. Since two equal states are not apart, the lemma assumes that the two states are not apart.

```
lemma hypothesis_no_step_output_same:
  assumes "invar (S,F,T)" and
    "¬ (∃ S' F' T'. algo_step (S,F,T) (S',F',T'))" and
    "hypothesis (S,F,T) h" and
    "M = (q_0,f)" and
    "¬ (apart_mealy M (state_run_mealy f q_0 p) q)" and
    "p ∈ S"
  shows "outs_run_mealy f q inp = outs_run_mealy h p inp"
```

The proof uses an induciton on the input string inp. The base case is trivial, but for the step of the induction it needs to make a case distinction, either the next state reached is part of the Basis or not, if the next state is part of the Basis, an easy application of the induction hypothesis can take place. Otherwise if the transition leaves the Basis, an application of the induction hypothesis is more difficult, because the non apartness is harder to show. For the rest of this proof we call the first letter of the input a, the state reached from the hypothesis p' where $h\ (p, a) = (p', op)$ and the next state for the hidden Mealy Machine q' where $f(q, a) = (q', op)$. To apply the induction hypothesis to the case where $p@[a]$ is not in the Basis, the proof needs to show that the state it can find in the Mealy Machine $p'm = state\_run\_Mealy\ f\ q_0\ p'$ is not apart from q'. It uses a clever contradiction to show that: if those two were apart, then $p@[a]$ and p' would also be apart in the Mealy Machine, which is impossible, because the states are not apart in the complete observation tree and every difference query returns $None$.
Now using the lemmata in tandem we can show that the algorithm is correct.

28

4. Conclusion

In the context of this work we formalized the $L\#$ algorithm in Isabelle. To simplify the proof we changed the last Rule of the algorithm through adding a different query to the teacher. This change simplified formal verification of rule four. We also added explicit parsing of output queries, as no specific way to change the observation tree was mentioned in the original paper. Then we proved that the algorithm is correct and that it Terminates in a polynomial amount of time. One of the drawbacks of this work is that we did not show equality of the two different forms of Rule 4, and another one is that no executable version, where every part of the algorithm is written as a function rather than as a inductive, of this algorithm was proven.

4.1. Future Work

A proof of an executable version of the $L\#$ algorithm would improve the formalization further. In the context of the work we tried to make a formalization work, but did not achieve a proof or an fully executable function. The exact implementation of functions that obtain are hard to figure out and using an proofs using induction scheme on a function where the increase of the norm is the only metric that changes is even more difficult. Another improvement would formalize the equality of both Rule 4 versions, or inculde similar proofs for a transition system with the original Rule 4. In the context of this work this was not achived because the details of `ProcCounterEx` are hard to understand and hard to formalize. Processing counterexamples would need a second big proof with some kind of invariant of the function. Within the limited timeframe of this thesis those two extra goals where not reached.
Further steps would be to extend the proof for new extenstions like $L\#$ for DFAs [20] or the $L\#$ algorithm extended for mealy machines with Timers [3].

29

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[2] Lasse Blaauwbroek, David M. Cerna, Thibault Gauthier, Jan Jakubův, Cezary Kaliszyk, Martin Suda, and Josef Urban. *Learning Guided Automated Reasoning: A Brief Survey*, page 54–83. Springer Nature Switzerland, 2024.

[3] Véronique Bruyère, Bharat Garhewal, Guillermo A. Pérez, Gaëtan Staquet, and Frits W. Vaandrager. Active learning of mealy machines with timers. *CoRR*, abs/2403.02019, 2024.

[4] Filip Bártek, Ahmed Bhayat, Robin Coutelier, Márton Hajdu, Matthias Hetzenberger, Petra Hozzová, Laura Kovács, Jakob Rath, Michael Rawson, Giles Reger, Martin Suda, Johannes Schoisswohl, and Andrei Voronkov. The vampire diary, 2025.

[5] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[6] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198, Dec 1931.

[7] John Harrison. Hol light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[8] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing.

[9] Tanya Khovanova. Martin gardner's mistake, 2011.

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

30

[11] Katharina Kreuzer. Crystals-kyber. *Archive of Formal Proofs*, September 2022. `https://isa-afp.org/entries/CRYSTALS-Kyber.html`, Formal proof development.

[12] Edward F. Moore, W. R. ASHBY, J. T. CULBERTSON, M. D. DAVIS, S. C. KLEENE, K. DE LEEUW, D. M. MAC KAY, J. MC CARTHY, M. L. MINSKY, E. F. MOORE, C. E. SHANNON, N. SHAPIRO, A. M. UTTLEY, and J. VON NEUMANN. *GEDANKEN-EXPERIMENTS ON SEQUENTIAL MACHINES*, pages 129–154. Princeton University Press, 1956.

[13] Tsyh-Wen Pao and John W. Carr. A solution of the syntactical induction-inference problem for regular languages. *Computer Languages*, 3(1):53–64, 1978.

[14] Lawrence C. Paulson. Isabelle: The next 700 theorem provers, 2000.

[15] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. *Black Box Checking*, pages 225–240. Springer US, Boston, MA, 1999.

[16] Jacques H. H. Perk. Erroneous solution of three-dimensional (3d) simple orthorhombic ising lattices, 2013.

[17] R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[18] Sarah Tilscher and Simon Wimmer. Ll(1) parser generator. *Archive of Formal Proofs*, May 2024. `https://isa-afp.org/entries/LL1_Parser.html`, Formal proof development.

[19] Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. *CoRR*, abs/2107.05419, 2021.

[20] Frits W. Vaandrager and Martijn Sanders. $L^{\#}$ for dfas. In Nils Jansen, Sebastian Junges, Benjamin Lucien Kaminski, Christoph Matheja, Thomas Noll, Tim Quatmann, Mariëlle Stoelinga, and Matthias Volk, editors, *Principles of Verification: Cycling the Probabilistic Landscape - Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*, volume 15262 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2024.

[21] A.N. Whitehead and B. Russell. *Principia Mathematica*. Number Bd. 1 in Principia Mathematica. Cambridge University Press, 1910.