Bachelor's Thesis Computing Science

Radboud University Nijmegen

Towards a Formalization of $L^{\#}$ in Coq

Author:
Sander Suverkropp
s1019051

First supervisor/assessor:
Dr. Freek Wiedijk

Second assessor:
Dr. Jurriaan Rot

August 30, 2022

Abstract

In active automata learning, we learn the behaviour of an automata with the help of a teacher that answers queries. $L^{\#}$ is an algorithm for active automata learning of Mealy machines. It is based on a constructive form of inequality called apartness. In this thesis, I formalized the preliminaries for the $L^{\#}$ algorithm in Coq. This includes partial Mealy machines, observation trees, apartness and hypothesis construction.

Contents

Chapter 1

Introduction

Active automata learning was introduced in 1987 by Dana Angluin [2]. In active automata learning, a learner learns a regular language L with the help of a teacher. The learner can ask two types of queries to the teacher: "Is this word part of L?" and "Is L the language accepted by automata H?"

Active automata learning is used in formal methods to analyse and verify systems without complete specifications [8, 16]. In these cases, active automata learning is used to construct a behavioural model of the system. Then, the model can be checked for certain properties.

In the same paper, Angluin also introduced the L^* algorithm [2], which can learn regular languages in polynomial time. Various improvements have been made on this algorithm [17, 12, 13, 5, 9]. Isenberner showed that these algorithms all fit into a single framework [10].

Active automata learning can also be applied to many other types of automata. For instance, L^* has been adapted to different types of automata, like non-deterministic automata [3], mealy machines [19, 14], I/O automata [1] and visibly pushdown automata [10].

Many of these algorithms are similar to L^*, but with various improvements. The $L^\#$ algorithm [24], which learns Mealy machines, differs in that tries to establish apartness of observations, instead of equivalence of observations. Apartness is a constructive form of inequality [11]. In other words, it tries to determine which inputs must lead us to different states, instead of determining which inputs might lead to the same state. As its data structure it only uses a partial Mealy machine called an observation tree. This makes it simpler to implement than some other algorithms, while also achieving the worst-case time complexity.

In this thesis I work towards a formalization of $L^\#$ in the formal proof management system Coq [21]. Coq allows us to state and prove theorems and to export verified programs. It checks all proofs are valid, so we can verify that all proofs are completely correct.

I have not formalized the algorithm itself, but I have formalized Mealy

machines and the observation tree and apartness relation that the algorithm relies on. In addition, I have formalized the construction of a hypothesis. This is a Mealy machine that is given to the teacher for an equivalence query.

In Chapter 2, I will give background information about Mealy machines, observation trees and the apartness relation. Chapter 3 gives an overview of the $L^{\#}$ algorithm, and explains in detail how hypothesis construction works. Chapter 4 discusses the formalization in Coq.

Appendix A contains a code listing of the formalization. The full code can also be found at:
`https://gitlab.science.ru.nl/ssuverkropp/bachelor-thesis/-/tree/ca183462d8714edaf45d92dd115a4651896472b9`

Chapter 2

Mealy machines and apartness

A Mealy machine is a type of finite-state machine that produces outputs when given inputs. As such, it represents a function from input words to output words. The $L^\#$ algorithm learns this function for some hidden Mealy machine. It uses a partial Mealy machine to store the information that has been gathered about the hidden Mealy machine thus far. In this chapter, we define partial Mealy machines and their semantics, as well as functional simulations which relate them and an apartness relation on states in Mealy machines. This chapter is based on Section 2 of [24]. Some of the proofs are taken from the appendix in the preprint version [23]. The names the lemmas refer to the names of the corresponding lemma in the Coq formalization.

2.1 Partial functions and Kleene star

First, we need some definitions and notations around partial functions. A partial function $f\colon X \rightharpoonup Y$ is defined as a set $f \subseteq X \times Y$, where for every $x \in X$, there is at most one $y \in Y$ such that $(x, y) \in f$. This forms a function from some subset of X to Y.

If $(x, y) \in f$, we say that f is defined on x, and write $f(x)\downarrow$ and $f(x) = y$. If for all $y \in Y$, $(x, y) \notin f$, f is undefined on x, and we write $f(x)\uparrow$.

Partial functions from X to Y are partially ordered by $\sqsubseteq$. For $f, g : X \rightharpoonup Y$, we say that $f \sqsubseteq g$ if $f \subseteq g$ as sets. Equivalently, for all $x \in X$,

$$f(x)\downarrow \implies g(x)\downarrow \wedge f(x) = g(x).$$

The composition of partial functions $f : X \rightharpoonup Y$ and $g : Y \rightharpoonup Z$, is $g \circ f : X \rightharpoonup Z$ defined as follows. $g \circ f(x)\downarrow$ if $f(x)\downarrow$ and $g(f(x))\downarrow$. Then we have $g \circ f(x) = g(f(x))$.

Given an alphabet Σ, we define Σ^n to be the set of all words of length n over that alphabet:

$$\Sigma^0 = \{\epsilon\}$$
$$\Sigma^{i+1} = \{a : \sigma \mid a \in \Sigma \wedge \sigma \in \Sigma^i\}$$

Then Σ^* is the set of all words of any length over Σ.

$$\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma^i$$

We write $a : \sigma$ for the concatenation of a letter $a \in \Sigma$ and a word $\sigma \in \Sigma^*$, and $\sigma_1\ \sigma_2$ for the concatenation of two words $\sigma_1, \sigma_2 \in \Sigma^*$.

2.2 Mealy machines

A Mealy machine [15] is a finite-state machine, similar to a deterministic finite automaton (DFA). Where a DFA accepts certain input words based on accepting and non-accepting states, a Mealy machine produces an output. For each input letter, one output letter is generated, which is determined by both the state and input. For the input and output, we fix finite alphabets I and O.

Definition 2.2.1. A partial Mealy machine is a quadruple $(Q, q_0, \delta, \lambda)$, where

- Q is a finite set representing states
- $q_0 \in Q$ is the initial state
- $\delta : Q \times I \rightharpoonup Q$ is a partial function giving the next state
- $\lambda : Q \times I \rightharpoonup O$ is a partial function giving the output

We require that δ and λ are defined on the same pairs, so $\delta(q, i)\downarrow \iff \lambda(q, i)\downarrow$.

We use a superscript to differentiate between the components of different Mealy machines, so Q^M refers to the set of states of the Mealy machine M.

Notation 2.2.2. We write $q \xrightarrow{i/o} q'$ for $\delta(q, i) = q'$ and $\lambda(q, i) = o$.

A Mealy machine M is considered complete when δ^M and λ^M are total functions.

We can extend the transition functions δ and λ to apply them to input words instead of input letters. For δ on words, we compose the δ function with itself once for each input letter. For λ on words, we concatenate the output letters from the transitions for each input letter. The semantics of states in a Mealy machine is defined by how it transforms input words into output words.

Definition 2.2.3. We define the function $\delta^* : Q \times I^* \rightharpoonup Q$ inductively by

$$\delta^*(q, \epsilon) = q.$$

$$\delta^*(q, a : \sigma) = \delta^*(\delta(q, a), \sigma).$$

We define the function $\lambda^* Q \times I^* \rightharpoonup O^*$ inductively by

$$\lambda^*(q, \epsilon) = \epsilon$$

$$\lambda^*(q, a : \sigma) = \lambda(q, a) : \lambda^*(\delta(q, a), \sigma).$$

Note that both of these definitions use the composition of partial functions, so λ^* and δ^* are undefined if one of the transitions is undefined. From now on, we will omit the star, and also use δ and λ on words.

The semantics $[\![q]\!]$ of a state q is defined as a partial function from words in I to words in O by

$$[\![q]\!](\sigma) = \lambda(q, \sigma).$$

Figure 2.1 shows an example of a Mealy machine with states $Q = \{q_0, q_1, q_2, q_3, q_4\}$. This Mealy machine is not complete because there are no outgoing transitions from q_4, and because there is no outgoing transition from q_3 with input a.

For the initial state q_0, the semantics over input word aba is as follows:

$$[\![q_0]\!](aba) = \lambda(q_0, aba) = B\ \lambda(q_1, ba) = BB\ \lambda(q_1, a) = BBA$$

On the other hand, for q_1, the semantics over aba is not defined, since the third letter would be

$$\lambda(\delta(q_1, ab), a) = \lambda(q_3, a)\uparrow .$$

We can define equivalence of states and Mealy machines if their semantics are identical.

Definition 2.2.4. Two states q and q' of a Mealy machine are equivalent if $[\![q]\!] = [\![q']\!]$. Two Mealy machines M and N are equivalent if their initial states are equivalent. We write this as $q \approx q'$ for states and $M \approx N$ for Mealy machines.

In the example from Figure 2.1, we can see that $q_0 \approx q_2$.

2.3 Trees and functional simulations

In the $L^\#$ algorithm, a Mealy machine is created that resembles a part of a hidden Mealy machine. We can formalize this notion using functional simulation. This is a function that maps the states of one Mealy machine to the states of another, while preserving transitions between states.

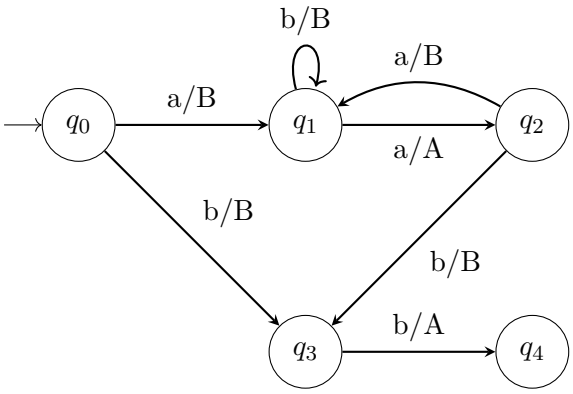

Figure 2.1: An example of a Mealy machine over the sets $I = \{a, b\}$ and $O = \{A, B\}$

Definition 2.3.1. Let M and N be two Mealy machines. A *functional simulation* $f : M \to N$ is a function $f \colon Q^M \to Q^N$ with

- $f(q_0^M) = q_0^N$, and
- if $q_1 \xrightarrow{i/o} q_2$ in M, then $f(q_1) \xrightarrow{i/o} f(q_2)$ in N.

With this definition of a functional simulation, we can now relate this to the semantics of states in the Mealy machines.

Lemma 2.3.2 (`func_sim_sem`)**.** *If $f : Q^M \to Q^N$ is a functional simulation, then for all $q \in Q^M$,*

$$[\![q]\!] \sqsubseteq [\![f(q)]\!]$$

Proof. For all words $\sigma \in I^*$, if $[\![q]\!](\sigma)\downarrow$, we need to show that $[\![f(q)]\!](\sigma)\downarrow$, and that $[\![q]\!](\sigma) = [\![f(q)]\!](\sigma)$. This can be proven by structural induction on the word. For the empty word, the semantics of a state are defined to be ϵ regardless of the state or Mealy machine. This makes the statement trivially true.
For a non-empty word, say $\sigma = i : \tau$, we know that there must be a transition $q \xrightarrow{i/o} q'$. The semantics is defined to be

$$[\![q]\!](i : \tau) = o : [\![q']\!](\tau).$$

Because f is a functional simulation, that means $f(q) \xrightarrow{i/o} f(q')$. Now we can write

$$[\![f(q)]\!](i : \tau) = o : [\![f(q')]\!](\tau).$$

With the induction hypothesis stating that $[\![q]\!](\tau) = [\![f(q')]\!](\tau)$, we can conclude that $[\![q]\!] \sqsubseteq [\![f(q)]\!]$. □

Definition 2.3.3. A Mealy machine T is a *tree* if, for all states q, there is exactly one input word $\mathsf{access}(q)$ such that $\delta(q_0^M, \mathsf{access}(q)) = q$. It is called an *observation tree* of a Mealy machine M if it is a tree and there exists a functional simulation $f : T \to M$.

In the $L^\#$ algorithm, we will build an observation tree of the hidden Mealy machine to store the information which we have accumulated.

2.4 Apartness

In the $L^\#$ algorithm, we build a complete Mealy machine out of an observation tree to give to the teacher. In doing this, we unify some states. We can never know for sure which states in the observation tree correspond to the same state in the hidden Mealy machine. Instead, we use the notion of apartness to look at the states that do not correspond to the same state in the hidden Mealy machine.

Definition 2.4.1. Two states q_1 and q_2 are considered *apart* if there is some input word σ such that $[\![q_1]\!](\sigma)\downarrow$, $[\![q_2]\!](\sigma)\downarrow$ and $[\![q_1]\!](\sigma) \neq [\![q_2]\!](\sigma)$. The word σ is then called the *witness*.

Notation 2.4.2. If two states q and q' are apart, we write $q \# q'$. If the word σ is the witness of this, we write $\sigma \vdash q \# q'$.

To illustrate this, we can apply it in Example 2.1. The state q_4 is not apart from any state, since it has no outgoing transitions. The states q_0 and q_1 are apart: $[\![q_0]\!](a) = B$, while $[\![q_1]\!](a) = A$. This makes a a witness, so we can write $a \vdash q \# q'$. With input b, both q_0 and q_1 give B as output, but another transition with input b reveals that they are in fact apart.

Note that for states in a complete Mealy machine, $q \# q'$ iff $q \not\approx q'$.

We can now relate apartness to functional simulation, by showing that a functional simulation can never map two states that are apart to two states that are equivalent.

Lemma 2.4.3 (`apart_func_sim`)**.** *Let $f : M \to N$ be a functional simulation, and $q, q' \in Q^M$. If $q \# q'$, then*

$$f(q) \not\approx f(q').$$

Proof. Let σ be a witness of $q \# q'$. Then we know that $[\![q]\!](\sigma)\downarrow$, $[\![q']\!](\sigma)\downarrow$ and $[\![q]\!](\sigma) \neq [\![q']\!](\sigma)$. By Lemma 2.3.2, we know that $[\![f(q)]\!](\sigma) = [\![q]\!](\sigma)$ and $[\![f(q')]\!](\sigma) = [\![q']\!](\sigma)$. Using this, we see that

$$[\![f(q)]\!](\sigma) = [\![q]\!](\sigma) \neq [\![q']\!](\sigma) = [\![f(q')]\!](\sigma). \qquad \square$$

When two state r, r' are apart, then we can show that a third state q must be apart from at least one of them if its semantics are defined on a witness for $r \# r'$. This property is called *weak cotransitivity*.

Lemma 2.4.4 (`weak_cotrans`)**.** *Let $r, r', q \in Q^M$, and $\sigma \in I^*$. If $\sigma \vdash r \# r'$, and $[\![q]\!](\sigma)\downarrow$, then $r \# q$ or $r' \# q$.*

Proof. Because σ is a witness of $r \# r'$, we know that $[\![r]\!](\sigma)\downarrow$ and $[\![r']\!](\sigma)\downarrow$, and we assumed that $[\![q]\!](\sigma)\downarrow$. If both $[\![r]\!](\sigma) = [\![q]\!](\sigma)$ and $[\![r']\!](\sigma) = [\![q]\!](\sigma)$, then it follows that $[\![r]\!](\sigma) = [\![r']\!](\sigma)$. This contradicts our assumption, so $r \# q$ or $r' \# q$. □

This lemma can be used to reveal apartness relations during learning. For example, in figure 2.1, we noted earlier that $bb \vdash q_0 \# q_1$, but q_4 is not apart from any state. If we add transitions for the word bb to q_4, q_4 will be apart from at least one of q_0 and q_1.

Chapter 3

The $L^{\#}$ algorithm

In automata learning, the goal is to learn the behaviour of a hidden Mealy machine. The $L^{\#}$ algorithm accomplishes this using an observation tree to store the information that has been collected about the hidden Mealy machine. In addition, it uses apartness to keep track of the states in the observation tree that cannot be equivalent.

Most of this chapter is based on Section 3 of [24]. Some of the proofs are from the appendix in the preprint version [23].

3.1 Framework

We use the MAT model, which stands for Minimally Adequate Teacher. This model was first proposed by Angluin in her seminal paper [2]. In this framework, a learner tries to learn the behaviour of a hidden automata with the help of a teacher. The teacher answers to two types of queries. In the original version for DFA's, these were the membership query and equivalence query.

The membership query indicates whether a word is accepted by the hidden DFA. The equivalence query checks if a given DFA is equivalent to the hidden DFA. If it isn't, a word is returned that is accepted by one, but not the other.

The L^* algorithm introduced by Angluin in the same paper accomplished this in a polynomial number of queries. It uses an observation table to store the information from the membership queries.

The semantics of a Mealy machine does not correspond to some language, but produces output words. To adapt the framework to this, the membership query is replaced by an output query, which gives the output word given a certain input word. Given a hidden Mealy machine M, the output query is formally defined as follows:

OutputQuery(σ)**:** *Returns the corresponding output word* $\lambda^M(q_0^M, \sigma)$.

The equivalence query checks if a given Mealy machine is equivalent to M.

If it isn't, an input word is given for which H and M give different outputs.

EquivalenceQuery(H)**:** *Returns* `yes` *if* $H \approx M$*. Otherwise it returns* `no`*, together with an input word* σ *such that* $\lambda(a_0^M, \sigma) \neq \lambda(q_0^H, \sigma)$*.*

In practice, there is no teacher. The behaviour that we want to learn is from a system that is treated as a black box. That means we can give inputs and receive outputs, but we cannot observe the inner workings of the system. In this situation, the equivalence query can't be answered with complete certainty. Instead, the equivalence query is approximated. One way to do this is using conformance testing [7].

In conformance testing, we try to verify whether a system meets its specifications. In this case the specifications are given in terms of the Mealy machine. To do this, we generate a test suite. Ideally, we want the system to pass the test suite if and only if it corresponds to the given Mealy machine. This is impossible to do in general. However, with an additional assumption, like a maximum size for the Mealy machine that the system corresponds to, it is possible.

3.2 Overview

The algorithm builds an observation tree T recording the results from all output queries. Initially T consists of only the initial state q_0 and an empty transition function. The observation tree is split into three parts.

- The *basis* $S \subseteq Q$ consists of the states that must represent different states in the hidden Mealy machine. All states in the basis are pairwise apart. That is, for every pair of distinct states in the basis $q, q' \in S, q \neq q'$ we know that $q \# q'$. Initially, the basis consists of only the initial state, $S = \{q_0\}$.

- The *frontier* $F \subseteq Q$ consists of the states that are not in the basis, but can be reached in one transition from a state in the basis. That is,
$$F = \{q \in Q \setminus S \mid \exists q' \in S, i \in I . \delta(q, i) = q\}.$$
These are the candidates to be added to the basis S.

- The remaining states $Q \setminus (S \cup F)$.

Because the states that are added to the basis are always chosen from the frontier, the basis has the shape of a tree. More formally, all states in the basis except for the initial state can be reached by a transition from a state in the basis. Conversely, if for a state $q \in Q$ there is a transition $\delta(q, i) \in S$, q must be in the basis.

Whenever we use an output query, the observation tree will be expanded to include the results from the output query.

The algorithm consists of four rules, (R1) through (R4), which applied non-deterministically. These rules can be found in Section 3.4. The algorithm halts only when it gets a response of `yes` to an equivalence query. The first rule adds states from the frontier to the basis. The second and third rule expand the observation tree with output queries. The fourth rule creates a hypothesis to use with the equivalence query. This is a complete Mealy machine that is created from the basis of the observation tree. If the answer is to the equivalence query is `no`, or if the hypothesis contradicts the information already encoded in the observation tree, then output queries are used to prevent the same hypothesis from being generated again.

Below is an example to illustrate how the algorithm works. The details are given in Section 3.4.

3.2.1 Example

Suppose we are learning the Mealy machine in Figure 3.1.

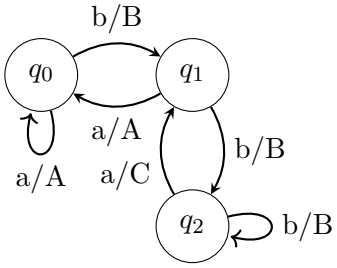

Figure 3.1: The hidden Mealy machine M

Initially, our observation tree will consist of only the initial state t_0, without any transitions. To indicate that the initial state is in the basis, it is coloured red.

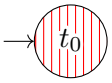

Figure 3.2: T_1: The initial observation tree

When there are no outgoing transitions from a state in the basis, we use output queries to get them. This is Rule (R2). The output queries a and b will return A and B respectively, allowing us to expand our observation tree to Figure 3.3. Blue is used to mark the states in the frontier.

All states in the basis have outgoing transitions, and the states in the frontier aren't apart from all the states in the basis, so now we can construct a hypothesis for an equivalence query. A hypothesis is a Mealy machine consisting of the states in the basis, that is partially consistent with the

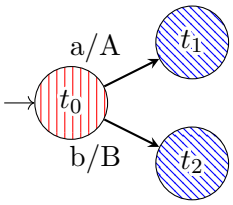

Figure 3.3: T_2: The observation tree after one step

information we have. Constructing a hypothesis is Rule (R4). This will look like Figure 3.4.

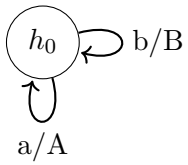

Figure 3.4: H_1: The hypothesis generated from T_2

This hypothesis is completely consistent with the observation tree, so we will use an equivalence query. Since this hypothesis is not equivalent to the hidden Mealy machine, we get a counterexample, for example bba. We always use an output query on the counterexample, which extends the observation tree to Figure 3.5.

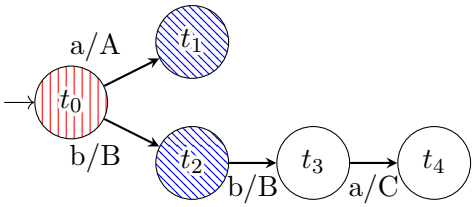

Figure 3.5: The observation tree after processing the equivalence query

Now t_0 and t_2 are apart. This means the frontier state t_2 is apart from all states in the basis, so we can add it to the basis. This is Rule (R1). Afterwards, we can also add t_3 to the basis, since it is now in the frontier, and it is apart from both t_0 and t_2. The next step is to use output queries to obtain outgoing transitions form the basis again. This results in Figure 3.6.

At this point, we could create another hypothesis. When we create a hypothesis, every frontier state is mapped to one state in the basis from which it is not apart. The transition going to that frontier state will then be replaced by a transition going to that state in the basis. For example, if t_1

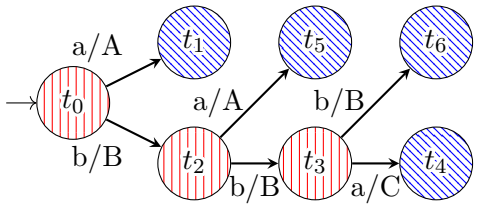

Figure 3.6: The observation tree after one step

is mapped to t_0, the transition $t_0 \xrightarrow{a/A} t_1$ will be replaced by $t_0 \xrightarrow{a/A} t_0$ as we have seen in the first hypothesis. This time, there are multiple states in the basis, and the frontier states are not yet apart from any of them. We will use Lemma 2.4.4 to extend the apartness relation until we can map every frontier state onto just one basis state. This is Rule (R3). For example, to find out whether a state is apart from t_3 or t_0 we can observe the output with input a. If the output is A, it is apart from t_3 and if the output is C it is apart from t_0. An output of B makes it apart from both. Doing this gives us the observation tree in Figure 3.7.

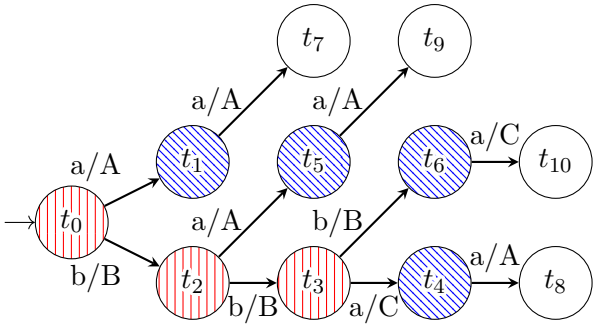

Figure 3.7: The observation tree when after output queries to distinguish between t_0 and t_3

Now $t_1 \# t_3$, $t_4 \# t_3$ and $t_5 \# t_3$, while t_6 is apart from both t_0 and t_2. Using more output queries we can also find apartness between $t_1 \# t_2$ and $t_4 \# t_0$ and $t_5 \# t_2$. Then all states in the frontier are apart from all but one state in the basis. We can map t_1 and t_5 to t_0, t_4 to t_2, and t_6 to t_3. When we use this to construct another hypothesis, the result is identical to the hidden Mealy machine. This hypothesis is accepted by the equivalence query.

3.3 Hypothesis

In the fourth rule of $L^\#$, we create a Mealy machine on which we can use the equivalence query using the observation tree. This is called a hypothesis.

We assume an observation tree T with basis S and frontier F. The apartness relation and access function always refer to the observation tree T.

A hypothesis is constructed by building a Mealy machine whose states are the basis of T. Transitions from the basis to the frontier in T need to be replaced by transitions to the basis in the hypothesis. To do this, we map the states in the frontier to states in the basis. We require that transitions in a hypothesis give the same output as their corresponding transitions in T, but a hypothesis might give different outputs for longer input words.

Definition 3.3.1. A Mealy machine H *contains the basis* if $Q^H = S$, and for all $q \in S$, $\delta^H(q_0, \mathsf{access}(q)) = q$. A *hypothesis* is a complete Mealy machine H that contains the basis, such that if $q \xrightarrow{i/o} p$ in T, then $q \xrightarrow{i/o} p'$ in H and $\neg p \# p'$ for some $p' \in S$. A hypothesis H is *consistent* if there exists a functional simulation from T to H. An input word σ is said to *lead to conflict* if $\delta^H(q_0, \sigma) \# \delta^T(q_0, \sigma)$.

The frontier F consists of all states that can be reached from the basis in one transition.

Definition 3.3.2. A state $q \in F$ is called:

- *isolated*, if for all $q' \in S$, $q \# q'$;
- *identified*, if $q \# q'$ for all but one $q' \in S$.

With these definitions, we can say something about the existence and uniqueness of a hypothesis.

Lemma 3.3.3 (`hypothesis_existence`, `hypothesis_unique`)**.** *If none of the states in the frontier are isolated, there exists a hypothesis. If the basis is complete and all states in the frontier are identified, the hypothesis is unique.*

Proof. Because a hypothesis must contain the basis, Q^H and q_0^H are fixed.

The only requirement for the output function is that for any transition in the basis $\lambda^T(q, i) = o$, we must have $\lambda^H(q, i) = o$. Such an output function always exists, and if the basis is complete, this defines a unique function λ^H.

Consider a relation $R \subseteq (S \times I) \times S$ defined by

$$R := \{((q, i), p) \mid \delta^T(q, i)\uparrow \vee \neg(\delta^T(q, i) \# p)\}.$$

For H to contain the basis, the transition function must map each pair (q, i) in the basis for which $\delta^T(q, i)$ is also in the basis, to $\delta^T(q, i)$. Since the

basis is pairwise apart, this requirement is fulfilled exactly when $\neg(\delta^H(q,i) \# \delta^T(q,i))$. For H to be a hypothesis, we must also have $\neg(\delta^H(q,i) \# \delta^T(q,i))$ for all other transitions. Thus, a transition function δ^H makes H a hypothesis if and only if it is some subset of R.

For every pair $(q,i) \in S \times I$ there are three cases:

1. $\delta(q,i) \in S$: Because the basis is pairwise apart, this means that (q,i) is only related to $\delta(q,i)$.

2. $\delta(q,i) \in F$: The pair (q,i) is related to all states $p \in S$ from which $\delta(q,i)$ is not apart. If $\delta(q,i)$ is not isolated, there is at least one such state, and if $\delta(q,i)$ is identified, there is exactly one such state.

3. $\delta(q,i)\uparrow$: In this case (q,i) is related to all states in the basis.

If all pairs (q,i) are related to at least one state, some complete function $\delta^H \subseteq R$ exists. In the first and third case, the pair (q,i) is always related to some $p \in S$. In the second case, the pair is related to some $p \in S$ if the frontier state $\delta^T(q,i)$ is not isolated. Thus, a hypothesis exists if none of the states in the frontier are isolated.

If all pairs (q,i) are related to exactly one state, R forms a transition function for δ^H. That means that the hypothesis is unique. In the first case, the pair (q,i) is only related to $\delta^T(q,i)$. In the second case, the pair is related to exactly one state if $\delta^T(q,i)$ is identified. In the third case, the pair might be related to many states, but this case will not occur if the basis is complete. So, if all states in the frontier are identified and the basis is complete, the hypothesis is unique. □

This result informs us when output queries need to be used to gather more information, and when a unique hypothesis can be constructed to provide in an equivalence query. While isolated states exist, no hypothesis can be constructed, so we need to add them to the basis. When multiple hypotheses can be constructed, output queries are used to extend the observation tree until only one is left. This avoids using costly equivalence queries when output queries can be used as well.

3.4 The $L^\#$ algorithm

The $L^\#$ algorithm consists of four rules which are applied nondeterministically. The algorithm is shown in pseudocode in Algorithm 1. The rules are as follows:

(R1) If a state in the frontier is isolated, that is to say, apart from all states in the basis, it must represent a different state in M, so it is added to the basis.

(R2) If for a certain state q in the basis there is no outgoing transition for some input i, we can use an output query with $\mathsf{access}(q)i$ to define this transition. This then also extends the frontier.

(R3) If a state q in the frontier is not apart from two different states r and r' in the basis, we can use rule (R3). If $\sigma \vdash r \# r'$, then we use an output query with $\mathsf{access}(q)\sigma$. Lemma 2.4.4 ensures that q is now apart from at least one of r and r'.

(R4) If none of the frontier states are isolated, and all outgoing transitions from the basis are defined, we can create a hypothesis. Then we check that it is consistent with the information in the observation tree. If it isn't, we use output queries to ensure that this hypothesis won't be generated in the future. If it is consistent, we use an equivalence query.

If neither of rules (R1) and (R2) are applicable, rule (R4) must be applicable, so the algorithm never blocks. This also means that rule (R3) is not strictly necessary. However, by using and prioritizing rule (R3) above rule (R4), the algorithm becomes much faster.

Algorithm 1 The $L^\#$ algorithm

```
procedure LSharp
    do q isolated for some q ∈ F →                                  ▷ Rule (R1)
        S ← S ∪ {q}
    [] δ^T(q, i)↑ for some q ∈ S, i ∈ I →                           ▷ Rule (R2)
        OutputQuery(access(q)i)
    [] ¬(q # r) ∧ ¬(q # r'), for some q ∈ F, r, r' ∈ S, r ≠ r' →   ▷ Rule (R3)
        σ ← witness of r # r'
        OutputQuery(access(q)σ)
    [] F has no isolated states, and the basis S is complete →     ▷ Rule (R4)
        H ← BuildHypothesis
        (b, σ) ← CheckConsistency(H)
        if b = yes then
            (b, ρ) ← EquivQuery(H)
            if b = yes then: return H
            else: σ ← shortest prefix of ρ such that δ^H(q0, σ) # δ^T(q0, σ)
        end if
        ProcCounter(H, σ)
    end do
end procedure
```

Algorithm 1 shows the $L^\#$ algorithm in pseudocode. It is important to note that whenever OutputQuery is used, the result will be used to

update the observation tree T.

In the fourth rule of the algorithm, three subroutines are used: BuildHypothesis, CheckConsistency and ProcCounter. BuildHypothesis simply picks one of the possible hypotheses that are proven to exist by Lemma 3.3.3. CheckConsistency checks whether the given hypothesis is *consistent*. If it isn't, an input word is returned, for which the hypothesis and observation tree give different outputs. ProcCounter uses a number of output queries to ensure that the given hypothesis will not be generated again. The details of these subroutines can be found in the original paper about $L^{\#}$ [24].

Since the algorithm only halts if the equivalence query returns `yes`, the correctness of the algorithm is proven by showing it terminates. This in turn can be done by showing that each of the rules expands S, F or $\#$ restricted to $S \times F$. These are all bounded by the size of the hidden Mealy machine. Again, the details of this proof can be found in the original paper about $L^{\#}$ [24].

Chapter 4

Implementation in Coq

I formalized everything from Chapter 2 and Section 3.3 in Coq. In short, this consists of partial Mealy machines, the observation trees and the construction of the hypothesis. I also proved Lemma 3.3.3, which states the hypothesis exists and is unique under certain circumstances. The entire formalization consists of 774 lines of Coq code. It can be found in Appendix A. For this, I used the stdpp library [22].

4.1 Coq and stdpp

Coq is a formal proof assistant based on the Calculus of Inductive Constructions. It allows one to state theorems and prove them using proof tactics. These proof tactics manipulate the hypotheses and goals of the proof.

It can also be used to write algorithms as a functional programming language. These algorithms must always be proven to terminate. We can also prove other properties about these algorithms. Program extraction can then be used to extract an algorithm with certain verified properties.

The stdpp library is a Coq library that is created to serve as an extension of the standard library [22]. It provides additional data structures like finite maps, as well as lemmas about them. It uses typeclasses for common properties like finiteness or countability and overloaded notations like monad notations.

4.2 Mealy machines

Mealy machines are formalized as a record consisting of a set of states, an initial state and a transition function, together with a number of type class instances. The transition function is represented with a `gmap`. This is a type for finite maps from stdpp. This means that the map is defined on only finitely many inputs.

```
Record mealy (input output : Type) `{Countable input} := Mealy {
  Q :> Set;
  eqDecisionQ : EqDecision Q;
  finiteQ : Finite Q;
  q0 : Q;
  transition : gmap (Q * input) (Q * output)
}.
```

In order to define a `gmap` on `Q * input`, we need it to be countable. This is why we need countability of the input and the states. Since `gmap` limits the number of transitions to a finite number, we do not need to require that the set of state and set of inputs are finite. The finite transition function then makes sure that only a finite number of inputs and states are actually used. For the input we do this, and we will only later require that there are finitely many inputs.

Only requiring countability of the set of states instead of finiteness in this definition caused problems with conflicting type classes later. The countability of the set of states would then be derived from the finiteness in some cases, while a separate definition would be used in other cases. This is why finiteness of `Q` is required here. To define this, the equality of states needs to be decidable, so an instance of `EqDecision Q` is needed as well.

From the transition function, functions `delta` and `lambda` can be derived. Their outputs are `option Q` and `option output` to represent the fact that they are partial functions. Similarly, we define the semantics as follows.

```
Fixpoint sem (q : Q M) (is : list input) : option (list output) :=
  match is with
  | [] => Some []
  | i :: is => '(q',o) ← transition !! (q,i); (o ::.) <$> sem q' is
  end.
```

In this definition, monad notation is used to compose partial functions.

4.3 Observation trees and apartness

The definition for a functional simulation is a straightforward translation to Coq. With the definition of a tree, the access function is included in the definition.

```
Definition tree (T:mealy input output) (access:T->list input) : Prop :=
  ∀ (q:T),
    repeat_delta (q0 T) (access q) = Some q
    /\ ∀ is, repeat_delta (q0 T) is = Some q -> is=access q.
```

The alternative was to have an existential quantifier in the definition. In this version, the definition would look like this

```
Definition tree (T:mealy input output) : Prop :=
  ∃ (access:T -> list input), ∀ (q:T),
    repeat_delta (q0 T) (access q) = Some q
    /\ ∀ is, repeat_delta (q0 T) is = Some q -> is=access q.
```

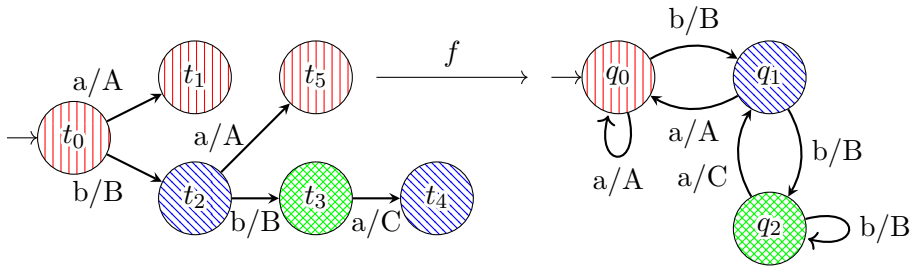

Figure 4.1: The example of an observation tree from Figure 1 in [24]

However, with this version the access function cannot be used in definitions of types in `Set`. This makes the definition of `contains_basis` more complex.

To check whether the created definitions were correct, I formalized the example from Figure 1 in [24], used to illustrate trees and functional simulation. Figure 4.1 shows this example. In this example, the colours represent how the function f maps states from the left Mealy machine to the states of the right Mealy machine.

I used new `Inductive` types for the input and output sets, and for the states of the Mealy machines. To prove that these types are countable or finite, I used bijections with `fin` types. `fin n` is a type with n instances, for which proofs of countability and finiteness are provided in stdpp. The proofs that this is an observation tree are straightforward, but slow in terms of processing time. On my machine, it took 12 seconds to process the proof that this is a tree. This is mainly due to the need to show that the provided access function gives the only input word leading to a specific state. To prove this, every input word that ends up in some state is checked for every state.

To define apartness in Coq, I first defined what a witness is, and then derived the definition for apartness from that.

```
Definition witness : list input -> M -> M -> Prop := λ is q1 q2,
  is_Some (sem q1 is) /\
  is_Some (sem q2 is) /\
  sem q1 is <> sem q2 is.

Definition state_apart : M -> M -> Prop := λ q p,
  ∃ is, witness is q p.
```

Lemmas 2.4.3 and 2.4.4 and their proofs could be translated to Coq in a very straightforward manner.

4.4 Hypothesis

To make the definitions and prove the lemmas detailed in Section 3.3, we need to work in a context with an observation tree with a basis and frontier.

To do this, I used a `Section`, with various context statements corresponding to the assumptions in Section 3.3.

```
Context `{Finite input, EqDecision output, Inhabited output}.
Context (T:mealy input output) `{tree T access_T}.
Context (basis:T -> bool) (basis_q0 : basis (q0 T)).
Context (basis_apart: ∀ q p, basis q -> basis p -> ¬ (q # p) -> q = p).
Context (basis_tree: ∀ q p i, delta T q i = Some p -> basis p -> basis q).
```

On the input type, we require a typeclass instance of `Finite` so we can go through all inputs. For the output type, we require typeclass instances of `EqDecision` and `Inhabited`. The `EqDecision output` instance is needed to ensure that it is decidable whether a transition should be in the basis or not. The `Inhabited output` instance is needed to construct a hypothesis when some basis states are missing outgoing transitions.

Next, we have some assumptions about the basis. These are that the initial state is part of the basis, that the basis is pairwise apart, and that the basis has the shape of a tree. Without the last assumption, a state might be part of the basis, even though it can only be reached through states that are not part of the basis. Then the hypothesis constructed in the proof of Lemma 3.3.3 would not necessarily contain the basis.

To define a hypothesis, we need a type with all the states in the basis of T. For this purpose we use a sigma type `HS`. Inhabitants of this type consist of a pair of a state q and a proof that q is in the basis. Note that `q↾e` is notation from stdpp for `exist _ q e`.

```
Definition HS :=  { q:T | basis q}.
```

Instead of defining a hypothesis in terms of Mealy machines, I defined it in terms of a transition function. This makes the definitions simpler, since the transition function is the only part of the hypothesis that is not predetermined by the observation tree in all cases.

```
Section hypothesis.
  Context (h:gmap (HS * input) (HS * output)).
  Definition Hy : mealy input output :=
    Mealy HS _ HS_fin ((q0 T) ↾ basis_q0) h.

  Definition contains_basis : Prop :=
    ∀ (q:T) (e: basis q),
    repeat_delta (q0 Hy) (access_T q) = Some (q ↾ e).

  Definition hypothesis :=
    contains_basis
    /\ complete Hy
    /\ ∀ (q p:T) (i:input) (o:output) (e:basis q),
        transition T !! (q,i) = Some (p,o) ->
        ∃ (p': Hy), transition Hy !! (q↾e:Hy,i) = Some (p',o) /\ ¬p#`p'.

  Definition consistent := hypothesis ∧ ∃ f : T->Hy, func_sim f.

  Definition leads_to_conflict `{contains_basis} : list input -> Prop :=
```

```
    λ is,
      ∃ q q1 e,
      repeat_delta (q0 Hy) is = Some (q ↾ e)
      /\ repeat_delta (q0 T) is = Some q1
      /\ q # q1.
End hypothesis.
```

4.4.1 Hypothesis construction

Lemma 3.3.3 is translated into two parts in the Coq code: the lemmas `hypothesis_existence` and `hypothesis_unique`. Before the proof of these lemmas can even start, a construction of a hypothesis is needed. This construction largely follows the construction given in the proof of Lemma 3.3.3. Because of the way a hypothesis is defined in the Coq code, only a transition function is needed here. This transition function in turn is based on a filtered list called `h_list` using the `list_to_map` function.

```
Let transition_in_basis : ((HS * input) * HS) -> Prop :=
  λ '(q1↾_, i, q2↾_),
    delta T q1 i = Some q2 /\ basis q2.

Let new_transition_fits : ((HS * input) * HS) -> Prop :=
  λ '(q1↾_, i, q2↾_),
    match delta T q1 i with
    | None => True
    | Some q1' => ¬ (q1' # q2) ∧ frontier q1'
    end.

Definition find_output : HS -> input -> output
  := λ q i, match lambda T (`q) i with
  | Some o => o
  | None => inhabitant
  end.

Definition add_output:
  ((HS * input) * HS) -> ((HS * input) * (HS * output))
  := λ '(q,i,p), ((q, i), (p, find_output q i)).

Let h_list := add_output <$> (filter
    (λ x, transition_in_basis x ∨ new_transition_fits x)
    (enum (HS * input * HS))).
```

The `transition_in_basis` function is for transitions $q \xrightarrow{i/o} q'$ where $q, a' \in S$. Turning this into a separate category makes the proofs simpler, since transitions within the basis are usually already a case that has to be handled separately.

The `new_transition_fits` function is very similar to the constraint in the proof of Lemma 3.3.3. However, it excludes transitions within the basis that are already captured by `transition_in_basis` by requiring that $\delta(q, i)$ is in the frontier. This is done to avoid duplicates and simplifies later reasoning.

The `add_output` function takes a triple and adds a fitting output to it. In case there is no outgoing transition, the inhabitant provided by the `Inhabited` typeclass is used.

A great difficulty of this definition is that it implicitly requires that the functions `transition_in_basis` and `new_transition_fits` are decidable. For the former, this is relatively straightforward. For the latter, however, we need to show that apartness is decidable. Because there are infinitely many input words that could function as a witness for apartness, this is not easily derived. Using the access function, we can find a maximum length for input words for which the semantics of a state can be defined in a tree. This length is defined below by `max_access_len`. The lemma `tree_max_len` prove that this is the maximum length for which the semantics of a state can be defined.

```
Definition max_access_len (T:mealy input output) :
  (T-> list input) -> nat :=
  λ access, list_max (map (length ∘ access) (enum T)).

Lemma rev_tree_max_len {T:mealy input output} (access:T -> list input)
                                              (tree0: tree T access) :
  ∀ (is:list input) (q:T),
  is_Some (sem q is) -> length is ≤ (max_access_len T access).
```

Now we can enumerate all input words shorter than this length. Deciding whether two states are apart then amounts to deciding whether there is a witness in that list of input words. In Coq, that means destructing an expression as follows.

```
destruct
    (existsb
        (λ is:list input, bool_decide (witness is x y))
        (lists_shorter_than (max_access_len T access_T))
        ) eqn:E.
```

In this expression, `existsb` is a boolean expression indicating whether there exists an element in a list fulfilling a requirement.

4.4.2 Hypothesis existence

The first part of Lemma 3.3.3, which states a hypothesis exists if no states in the frontier are isolated, is encoded in Coq as follows:

```
Theorem hypothesis_existence :
  (∀(q:T) (Fq:frontier q), ¬isolated q Fq) -> (∃ h, hypothesis h).
```

To prove this, we need to show three things: that `h` contains the basis, that it is complete, and that given a transition $q \xrightarrow{i/o} p$ in T, `h` has a transition from (q, i) to some state that is not apart from p.

To show that `h` contains the basis, we first show that `h` preserves all transitions that are in the basis.

```
Lemma h_preserves_transition :
  ∀ (q q1: T) (e:basis q) (e1:basis q1) (i:input) (o:output),
  transition T !! (q, i) = Some (q1, o) ->
  h !! ((q↾e):HS, i) = Some ((q1 ↾ e1):HS, o).
```

With this lemma, we can show that `h` contains the basis with structural induction on the input word access(q). This uses induction on the end of the word, using `ref_ind`.

Next, we show that `h` is complete. For a given pair of a state and an input q, i, we have three different cases to consider: the case where $\delta^T(q,i)\uparrow$, the case where $\delta^T(q,i)$ is in the basis, and the case where $\delta^T(q,i)$ is in the frontier.

For the last part, I have defined another lemma: `h_preserves_output`. This lemma states that if a transition exists in T, the output of the corresponding transition in `h` is the same.

```
Lemma h_preserves_output :
  (∀(q:T) (Fq:frontier q), ¬isolated q Fq) ->
  (∀q e i, is_Some (transition T !! (q,i)) ->
    lambda T q i = lambda (Hy h) (q↦e) i).
```

With this lemma, we can show that `h` is a hypothesis by making a case distinction on whether the result of the transition is in the basis or in the frontier.

4.4.3 Hypothesis uniqueness

The second part of Lemma 3.3.3, which states the hypothesis is unique if all frontier states are identified and the basis is complete, is encoded in Coq as follows:

```
Theorem hypothesis_unique (BC:basis_complete) (FI:frontier_identified):
  ∃ h, ∀ h0, hypothesis h0 -> h=h0.
```

In order to prove this, we first show that `h_list` can be interpreted as a function. This means that for any two pairs (x_1, y_1) and (x_2, y_2) in `h_list`, $x_1 = x_2$ implies $y_1 = y_2$.

```
Lemma h_list_functional : basis_complete -> frontier_identified ->
    ∀ x y1 y2, (x, y1) ∈ h_list -> (x, y2) ∈ h_list -> y1 = y2.
```

With the help of this lemma, we can show that every pair of an input and state only occur once as the first part of a pair in `h_list`.

```
Lemma NoDup_h_list :
  basis_complete -> frontier_identified -> NoDup h_list.*1.
```

This allows us to use the lemma `elem_of_list_to_map`, which states that a transition is in the `gmap h` if it is in `h_list`. In order to prove the lemma `hypothesis_unique`, we have to prove that the transitions of any hypothesis `h0` must be equal to that of `h`. Given an input i and a state q, we distinguish two cases: $\delta^T(q,i)$ is in the basis, or $\delta^T(q,i)$ is in the frontier. In the first case, we use the lemma `h_preserves_transition` to show that the transition in `h` is identical to the transition in T, and then we show that the transition in

`h0` must be identical to that too. In the second case, we can use the lemma `elem_of_list_to_map` to show that `h` sends (q, i) to the only state in the basis that is not apart from $\delta^T(q, i)$. The state that `h0` sends (q, i) must be the same state, since all states in the frontier are identified.

Chapter 5

Related work

As far as I could find, no work has previously been done on formalizing active automata learning algorithms. Mealy machines have been formalized in Coq [6, 4], but I could not find an implementation of Partial Mealy machines in Coq. Mealy machines have also been formalized in Isabelle, another proof assistant [18].

Apart from $L^{\#}$, several other algorithms for active automata learning of Mealy machines exist. First of all, L^* has been adapted to Mealy machines [19, 14]. The L^* algorithm uses an observation table as its main data structure. In this table, every cell corresponds to some input word. Every row represents a prefix, and every column represents a suffix. The prefixes can be compared to states in the basis and frontier, while the suffixes are similar to witnesses of apartness in $L^{\#}$.

Secondly, TTT can also be adapted to Mealy machines [10]. The TTT algorithm has a better theoretical complexity than L^*, and performs better in practice. The TTT algorithm uses a binary *discrimination tree* to keep track of the input suffixes used to discriminate between states. This allows it to use fewer output queries than L^* adaptions to discriminate between states.

Lastly, [20] describes an algorithm using an observation tree like $L^{\#}$. However, it does not prove correctness, and does not give an exact complexity. In a practical evaluation, it compares favourably to adaptions of L^* and TTT.

The algorithm formalized in this thesis, $L^{\#}$, is competitive with TTT and adaptions of L^*, and when adaptive distinguishing sequences are used, it is the fastest of these [24].

Chapter 6

Conclusions

We have discussed Mealy machines and the $L^{\#}$ algorithm. We have formalized partial Mealy machines, observation trees and apartness in Coq.

In addition, we have formalized the construction of a hypothesis, and we have proved that this results in a hypothesis and that it is unique if the basis is complete and the frontier is identified.

In hindsight, it might have been better to prove the existence and uniqueness of a mapping from the frontier to the basis with certain properties, instead of proving that the entire transition function exists and is unique. Then we could separately prove that this mapping from the frontier to the basis can be used to construct a hypothesis. This would have meant that the proofs for existence and uniqueness of the hypothesis would not have to deal with transitions within the basis, which are identical for any hypothesis.

Future work could use the work in this thesis to formalize the $L^{\#}$ algorithm itself. One way to define a teacher with an output query and an equivalence query is shown below. With this definition, there only needs to be a proof that the queries correspond to some hidden Mealy machine. This way, we can use this fact to prove that the algorithm terminates in Coq, but when we extract the program, we can also run it with other teachers.

```
Record teacher := Teacher {
  outputQuery : list input -> list output;
  equivQuery : ∀ (H:mealy input output) `(complete H), equivAns;
  teacherConsistent : ∃ (M:mealy input output),
    complete M
    /\ ∀ is : list input, Some (outputQuery is) = sem (q0 M) is
    /\ ∀ (H:mealy input output) (CH: complete H),
      match (equivQuery H CH) with
      | equiv => mealy_equiv M H
      | nonEquiv is => sem (q0 M) <> sem (q0 H)
      end
}.
```

The main difficulties of formalizing the $L^{\#}$ algorithm will probably consist of the non-determinism and the termination. One way of handling the non-determinism is to make a deterministic variant, by for example deciding

a rule with a lower number always goes first. Proving termination of the main algorithm and of the subroutines will then remain.

Bibliography

[1] Fides Aarts and Frits Vaandrager. Learning i/o automata. In *International Conference on Concurrency Theory*, pages 71–85. Springer, 2010.

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

[3] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of nfa. In *IJCAI*, volume 9, pages 1004–1009, 2009.

[4] Fritjof Bornebusch. *Coq meets CλaSH: proposing a hardware design synthesis flow that combines proof assistants with functional hardware description languages*. PhD thesis, Universität Bremen, 2021.

[5] Ana Cavalcanti and Dennis Dams. *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009, Proceedings*, volume 5850. Springer, 2009.

[6] Solange Coupet-Grimal and Line Jakubiec. Hardware verification using co-induction in coq. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, pages 91–108, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[7] Falk Howar and Bernhard Steffen. *Active Automata Learning in Practice*, pages 123–148. Springer International Publishing, Cham, 2018.

[8] Hardi Hungar and Bernhard Steffen. Behavior-based model construction. *International Journal on Software Tools for Technology Transfer*, 6(1):4–14, Jul 2004.

[9] Muhammad Naeem Irfan, Catherine Oriat, and Roland Groz. Angluin style finite state machine inference with non-optimal counterexamples. In *Proceedings of the First International Workshop on Model Inference In Testing*, pages 11–19, 2010.

[10] Malte Isberner. *Foundations of active automata learning: an algorithmic perspective*. PhD thesis, 2015.

[11] Bart Jacobs and Herman Geuvers. Relating apartness and bisimulation. *Logical Methods in Computer Science*, 17, 2021.

[12] Michael J Kearns and Umesh Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[13] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, 1995.

[14] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No.04EX940)*, pages 95–100, 2004.

[15] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

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

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

[18] Robert Sachtleben, Robert M. Hierons, Wen-ling Huang, and Jan Peleska. A mechanised proof of an adaptive state counting algorithm. In Christophe Gaston, Nikolai Kosmatov, and Pascale Le Gall, editors, *Testing Software and Systems*, pages 176–193, Cham, 2019. Springer International Publishing.

[19] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, pages 207–222, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[20] Michal Soucha and Kirill Bogdanov. Observation Tree Approach: Active Learning Relying on Testing. *The Computer Journal*, 63(9):1298–1310, 07 2019.

[21] The Coq Development Team. *The Coq Reference Manual, Release 8.15.2*. May 2022.

[22] The Iris Development Team. Coq-std++. URL: `https://gitlab.mpi-sws.org/iris/stdpp/` (accessed on August 25 2022).

[23] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. *arXiv preprint arXiv:2107.05419*, 2021.

[24] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–243. Springer, 2022.

Appendix A

Coq code

Below is the Coq code of the formalization. Note that the lemma numbers in the comments refer to the numbering in 1. It can also be found at:
https://gitlab.science.ru.nl/ssuverkropp/bachelor-thesis/-/tree/ca183462d8714edaf45d92dd115a4651896472b9

```
From stdpp Require Import fin_maps gmap finite fin option.

Record mealy (input output : Type) `{Countable input} := Mealy {
  Q :> Set;
  eqDecisionQ : EqDecision Q;
  finiteQ : Finite Q;
  q0 : Q;
  transition : gmap (Q * input) (Q * output)
}.

Arguments Q {_ _ _ _} _.
Arguments transition {_ _ _ _ _}.
Arguments q0 {_ _ _ _} _.
Arguments Mealy {_ _ _ _} _ _ _ _ _.

Section instances.
  Context `{Countable input} {output:Type} {M:mealy input output}.
  Global Instance eqDecQ : EqDecision M := eqDecisionQ input output M.
  Global Instance finQ : Finite M := finiteQ input output M.
End instances.

Definition complete {input output:Type} `{Countable input} (M : mealy
↪  input output) :=
  ∀ (q:Q M) (i:input), is_Some (transition !! (q,i)).

Section semantics.
  Context `{Countable input} {output:Type}.
  Context  {M : mealy input output}.

  Definition lambda : Q M -> input -> option output := λ q i,
    snd <$> (transition !! (q, i)).

  Definition delta : Q M -> input -> option (Q M) := λ q i,
    fst <$> (transition !! (q, i)).

  Fixpoint repeat_delta (q:Q M) (is:list input) : option (Q M) :=
  match is with
```

```
  | nil => Some q
  | i::is' => q' ← delta q i; repeat_delta q' is'
  end.

  Lemma repeat_delta_app_Some : ∀ is1 is2 q q',
    repeat_delta q is1 = Some q' ->
    repeat_delta q' is2 = repeat_delta q (is1 ++ is2).
  Proof.
  induction is1.
  - intros. inversion H0. done.
  - intros is2 q q'. simpl. destruct (delta q a). 2:done. apply IHis1.
  Qed.

  Lemma repeat_delta_app_None : ∀ is1 is2 q,
    repeat_delta q is1 = None -> repeat_delta q (is1 ++ is2) = None.
  Proof.
  induction is1.
  - done.
  - simpl. intros is2 q. destruct (delta q a). 2:done. apply IHis1.
  Qed.

  Fixpoint sem (q : Q M) (is : list input) : option (list output) :=
    match is with
    | [] => Some []
    | i :: is => '(q',o) ← transition !! (q,i); (o ::.) <$> sem q' is
    end.

  Lemma is_Some_repeat_delta_sem :
    ∀ is q, is_Some (repeat_delta q is) ↔ is_Some (sem q is).
  Proof.
  split; revert q; induction is; (try done); intros; simpl;
  [destruct H0 as (q1 & H0) | destruct H0 as (os & H0)]; simpl in H0;
  [unfold delta in H0 | unfold delta ]; destruct (transition !! (q, a));
  try done; simpl; destruct p; simpl in H0.
  - apply fmap_is_Some. apply IHis. done.
  - simpl. apply IHis. elim fmap_is_Some with (cons o) (sem q1 is).
    intros. apply H1. done.
  Qed.

  Lemma sem_length :
    ∀ is q os,
      sem q is = Some os -> length is = length os.
  Proof.
    induction is; intros.
    - inversion H0. tauto.
    - simpl in H0. destruct (transition !! (q,a)); [..|done].
      destruct p, os. simpl in H0. destruct (sem q1 is); done.
      simpl. rewrite IHis with q1 os; [tauto|].
      destruct (sem q1 is) eqn:S.
      all: simpl in H0; rewrite S in H0; inversion H0; tauto.
  Qed.
End semantics.

Section functional_simulation.
   Context `{Countable input} {output:Type}.

  Arguments mealy _ _ {_ _}.

  Definition state_equiv {M N:mealy input output} : Q M -> Q N -> Prop :=
    λ q1 q2, ∀ is, sem q1 is = sem q2 is.
  Definition mealy_equiv : relation (mealy input output) := λ M N,
    state_equiv (q0 M) (q0 N).
```

```
Definition func_sim {M N:mealy input output} : (M -> N) -> Prop :=
  λ f,
  f (q0 M) = q0 N /\
  ∀ q1 q2 i o, transition !! (q1,i) = Some (q2, o) ->
    transition !! (f q1,i) = Some (f q2, o).

Definition partial_function_inclusion {A B}:
  relation (A -> option B) := λ f g,
  ∀ a b, f a = Some b -> g a = Some b.

(*lemma 2.4*)
Lemma func_sim_sem : ∀ {M N:mealy input output} (f:M -> N),
  func_sim f -> ∀ q:M, partial_function_inclusion (sem q) (sem (f q)).
Proof.
unfold partial_function_inclusion. intros M N f FS q is. revert q.
induction is; try done. intros q os S. simpl in *.
destruct (transition !! (q, a)) eqn:T. 2: inversion S.
destruct p. destruct FS as [FS1 FS2]. rewrite FS2 with q q1 a o.
2:done. simpl in *. destruct (sem q1 is) eqn:D; inversion S.
rewrite IHis with q1 l; done.
Qed.

(* Trees *)
Definition tree (T:mealy input output) (access:T->list input) : Prop :=
  ∀ (q:T),
    repeat_delta (q0 T) (access q) = Some q
    /\ ∀ is, repeat_delta (q0 T) is = Some q -> is=access q.

Definition observation_tree (T M:mealy input output) : Prop :=
  (∃ ac, tree T ac) /\ ∃f: T->M, func_sim f.

Definition max_access_len (T:mealy input output) :
  (T-> list input) -> nat :=
  λ access, list_max (map (length ∘ access) (enum T)).

Lemma tree_max_len {T:mealy input output} (access : T -> list input)
↪  `(tree T access) :
  ∀ (is:list input) (q:T),
  (max_access_len T access) < length is -> ¬ is_Some (sem q is).
Proof.
intros. intro. apply is_Some_repeat_delta_sem in H1.
rewrite repeat_delta_app_Some with (access q) is (q0 T) q in H1;
[ | apply tree0].
destruct H1 as (q' & H1). destruct tree0 with q'. apply H3 in H1.
apply Nat.lt_lt_add_l with
  (list_max (map (length ∘ access) (enum T))) (length is) (length
  ↪  (access q)) in H0.
rewrite <- app_length in H0. rewrite H1 in H0.
apply list_max_lt with
  (map (length ∘ access) (enum T)) (length (access q')) in H0.
revert H0. apply Exists_not_Forall. apply Exists_exists.
exists ((length ∘ access) q'). split.
- apply in_map. apply elem_of_list_In. apply elem_of_enum.
- simpl. nia.
- intro. apply map_eq_nil in H5. apply elem_of_nil with q.
  rewrite <- H5. apply elem_of_enum.
Qed.

Lemma rev_tree_max_len {T:mealy input output} (access:T -> list input)
↪  (tree0: tree T access) :
  ∀ (is:list input) (q:T),
```

```
    is_Some (sem q is) -> length is ≤ (max_access_len T access).
  Proof.
  intros. apply Nat.le_ngt. intro.
  elim tree_max_len with access is q; done.
  Qed.

End functional_simulation.
Notation " x ~= y " := (state_equiv x y) (at level 70, no associativity).

Module tree_example.
  Inductive inp := a|b.
  Inductive out := A|B|C.
  Inductive Tstates := t0|t1|t2|t3|t4|t5.
  Inductive Mstates := m0|m1|m2.
  Definition fin2inp : nat -> inp := λ x, match x with
    | 0 => a
    | _ => b
    end.
  Definition inp2fin : inp -> nat := λ x, match x with
    | a => 0
    | b => 1
    end.
  Local Instance EqDecInp: EqDecision inp. solve_decision. Qed.
  Local Instance CountableInp : Countable inp.
  Proof.
  apply inj_countable' with inp2fin fin2inp. intro. destruct x; done.
  Defined.
  Local Instance EqDecisionTstates : EqDecision Tstates.
  solve_decision. Qed.
  Local Instance EqDecisionMstates : EqDecision Mstates.
  solve_decision. Qed.
  Definition fin2Tstates : (fin 6) -> Tstates := λ n,
    match n return Tstates with
    | 0%fin => t0
    | 1%fin => t1
    | 2%fin => t2
    | 3%fin => t3
    | 4%fin => t4
    | _ => t5
    end.
  Definition fin2Mstates : fin 3 -> Mstates := λ n,
    match n return Mstates with
    | 0%fin => m0
    | 1%fin => m1
    | _ => m2
    end.
  Local Instance FiniteTstates : Finite Tstates.
  Proof.
  apply surjective_finite with fin2Tstates. unfold Surj. intro.
  destruct y;
  [exists 0%fin|
    exists 1%fin|
    exists 2%fin|
    exists 3%fin|
    exists 4%fin|
    exists 5%fin].
  all:done.
  Qed.
  Local Instance FiniteMstates : Finite Mstates.
  Proof.
  apply surjective_finite with fin2Mstates. unfold Surj. intro.
  destruct y; [exists 0%fin|exists 1%fin|exists 2%fin]; done.
```

```
  Qed.

  Definition T_tr : gmap (Tstates * inp) (Tstates * out) :=
     <[(t0, a):=(t1, A)]>
    (<[(t0, b):=(t2, B)]>
    (<[(t2, a):=(t5, A)]>
    (<[(t2, b):=(t3, B)]>
    (<[(t3, a):=(t4, C)]>
    ∅)))).
  Definition M_tr : gmap (Mstates * inp) (Mstates * out) :=
     <[(m0, a):=(m0, A)]>
    (<[(m0, b):=(m1, B)]>
    (<[(m1, a):=(m0, A)]>
    (<[(m1, b):=(m2, B)]>
    (<[(m2, a):=(m1, C)]>
    (<[(m2, b):=(m2, B)]>
    ∅))))).
  Definition T : mealy inp out := Mealy Tstates _ _ t0 T_tr.
  Definition M : mealy inp out := Mealy Mstates _ _ m0 M_tr.
  Definition f : T -> M := λ t,
  match t with
    | t0|t1|t5 => m0
    | t2|t4 => m1
    | t3 => m2
  end.
  Lemma f_funcsim : func_sim f.
  Proof.
  unfold func_sim. split. done. intros. simpl in *. unfold T_tr in H.
  unfold M_tr. destruct q1, i; simpl_map; inversion H; done.
  Qed.

  Definition access_T : T -> list inp := λ t,
  match t with
    | t0 => nil
    | t1 => cons a nil
    | t2 => cons b nil
    | t3 => cons b (cons b nil)
    | t4 => cons b (cons b (cons a nil))
    | t5 => cons b (cons a nil)
  end.

  Lemma T_tree : tree T access_T.
  Proof.
  unfold tree. intros. split.
  destruct q; simpl; unfold delta; simpl; unfold T_tr;
    simplify_map_eq; tauto.
  destruct q; intros is H; simpl in H.
  all: repeat (destruct is; simpl in H; try (inversion H; tauto);
      destruct i; unfold delta in H; simpl in H; unfold T_tr in H;
      simpl_map; simpl in H).
  Qed.

  Lemma T_obs_tree : observation_tree T M.
  Proof.
  unfold observation_tree. split. exists access_T. apply T_tree.
  exists f. apply f_funcsim.
  Qed.

End tree_example.

Section apartness.
  Context `{Countable input} `{EqDecision output} {M:mealy input output}.
```

```
  Definition witness : list input -> M -> M -> Prop := λ is q1 q2,
      is_Some (sem q1 is) /\
      is_Some (sem q2 is) /\
      sem q1 is <> sem q2 is.

  Definition state_apart : M -> M -> Prop := λ q p,
    ∃ is, witness is q p.

  Global Instance witness_dec :
  ∀ (is:list input) (q1 q2:M), Decision (witness is q1 q2).
  Proof.
  solve_decision.
  Qed.

  Lemma apart_irreflexive : ∀q, ¬ (state_apart q q).
  Proof.
  intro. unfold state_apart. intro. destruct H0.
  destruct H0 as (_ & _ & H0). destruct H0. tauto.
  Qed.
End apartness.

Notation " p # q " := (state_apart p q) (at level 60, no associativity).

Section apartness_lemmas.
  Context `{Countable input, EqDecision output}.
  (* lemma 2.7 *)
  Lemma apart_func_sim :
    ∀ {T M:mealy input output} (f:T->M) (q p:T),
    func_sim f -> q # p -> ¬ f q ~= f p.
  Proof.
  intros T M f q p FS A. unfold state_apart in A.
  destruct A as (is & ((out1 & S1) & (out2 & S2) & N)).
  unfold not, state_equiv. intro H0. apply N. rewrite S1. rewrite S2.
  rewrite <- func_sim_sem with f q is out1.
  rewrite <- func_sim_sem with f p is out2. all: done.
  Qed.

  (* lemma 2.8 *)
  Lemma weak_cotransitivity :
    ∀ (M : mealy input output) (r r' q : M) (is : list input),
    (witness is r r') /\ (is_Some (sem q is)) -> (q # r) \/ (q # r').
  Proof.
  intros. destruct H0 as [W E]. destruct W as (H1 & H2 & N).
  inversion E. destruct H1 as (out1 & S1). destruct H2 as (out2 & S2).
  destruct (decide (x = out1)) as [->|]; [right | left];
  unfold state_apart; exists is; repeat split; try done.
  - rewrite H0. rewrite <- S1. tauto.
  - rewrite H0. rewrite S1. intro. inversion H1. done.
  Qed.
End apartness_lemmas.

Arguments transition {_ _ _ _} _.
Arguments lambda {_ _ _ _} _.
Arguments delta {_ _ _ _} _.

Section hypothesis_construction.
  Context `{Finite input, EqDecision output, Inhabited output}.
  Context (T:mealy input output) `{tree T access_T}.
  Context (basis:T -> bool) (basis_q0 : basis (q0 T)).
  Context (basis_apart: ∀ q p, basis q -> basis p -> ¬ (q # p) -> q = p).
```

```
Context (basis_tree: ∀ q p i, delta T q i = Some p -> basis p -> basis
↪  q).
Definition HS :=  { q:T | basis q}.
Instance HS_fin : Finite HS.
Proof.
apply sig_finite. solve_decision. apply finiteQ.
Qed.
Lemma HS_eq : ∀(q q':T) (e:basis q) (e':basis q'),
  (q↾e:HS) = (q'↾e':HS) ↔ q = q'.
Proof.
intros. split. intros. inversion H1. done.
intro. apply sig_eq_pi. intro. apply Is_true_pi. done.
Qed.

Section hypothesis.
  Context (h:gmap (HS * input) (HS * output)).
  Definition Hy : mealy input output :=
    Mealy HS _ HS_fin ((q0 T) ↾ basis_q0) h.

  Definition contains_basis : Prop :=
    ∀ (q:T) (e: basis q),
    repeat_delta (q0 Hy) (access_T q) = Some (q ↾ e).

  Definition hypothesis :=
    contains_basis
    /\ complete Hy
    /\ ∀ (q p:T) (i:input) (o:output) (e:basis q),
        transition T !! (q,i) = Some (p,o) ->
        ∃ (p': Hy), transition Hy !! (q↾e:Hy,i) = Some (p',o) /\ ¬p#`p'.

  Definition consistent := hypothesis /\ ∃ f : T->Hy, func_sim f.

  Definition leads_to_conflict `{contains_basis} : list input -> Prop :=
    λ is,
      ∃ q q1 e,
      repeat_delta (q0 Hy) is = Some (q ↾ e)
      /\ repeat_delta (q0 T) is = Some q1
      /\ q # q1.
End hypothesis.

Definition frontier (q:T) : Prop :=
  ¬ (basis q) /\
  exists q1 i, basis q1 /\ delta T q1 i = Some q.

Global Instance frontier_dec : forall (q:T), Decision (frontier q).
Proof.
intro. solve_decision.
Qed.

Definition isolated (q:T) (frontier_q:frontier q) : Prop :=
  ∀ q1:T, ¬ basis q1 \/ q # q1.

Definition identified (q:T) (frontier_q:frontier q) : Prop :=
  ∃ (q1:T) `(basis q1),
    ¬ (q # q1) /\ ∀ (q2:T) `(basis q2), (q # q2 \/ q2 = q1).

Definition basis_complete : Prop :=
  ∀ (q:T) (i:input), basis q -> is_Some (transition T !! (q, i)).

Let transition_in_basis : ((HS * input) * HS) -> Prop :=
  λ '(q1↾_, i, q2↾_),
    delta T q1 i = Some q2 /\ basis q2.
```

```
Instance tib_decision : ∀p, Decision (transition_in_basis p).
Proof.
intros. repeat (destruct p). destruct h, h0. solve_decision.
Qed.

Global Instance fin_exists_dec `{Finite A} {P: A->Prop}:
  (∀ x:A, Decision (P x)) -> Decision (∃ x:A, P x).
Proof.
intro. destruct (decide (Exists P (enum A))).
- left. apply Exists_exists in e. destruct e as (x & Inx & Px).
  exists x. tauto.
- right. intro. destruct H2. destruct n. apply Exists_exists.
  exists x. split. 2:done. apply elem_of_list_In. apply elem_of_enum.
Qed.

Fixpoint lists_shorter_than `{Finite A} (n:nat) : list (list A) :=
  match n with
  | O => [[]]
  | S m => (lists_shorter_than m) ++
      (map proj1_sig (enum {l : list A | length l = n}))
  end.

Lemma In_lists_shorter_than `{Finite A} (n:nat) :
    ∀ l: list A, length l ≤ n -> In l (lists_shorter_than n).
Proof.
intros. induction n.
- simpl. inversion H2. apply nil_length_inv in H4. left. done.
- apply in_or_app. inversion H2.
  + right. apply in_map_iff. rewrite H4. exists (l ↾ H4). split. done.
    apply elem_of_list_In. apply elem_of_enum.
  + left. apply IHn. tauto.
Qed.

Lemma existsb_false_forall {A:Type} {P: A-> bool} {l: list A} :
    existsb P l = false -> ∀ x, In x l -> ¬ P x.
Proof.
intros. intro. induction l. inversion H2. simpl in H0.
apply orb_false_elim in H1. destruct H1. inversion H2.
- rewrite H5 in H1. rewrite H1 in H3. inversion H3.
- apply IHl; done.
Qed.

Instance exists_witness_dec :
∀ x y: T, Decision (∃ is, witness is x y).
Proof.
intros.
destruct
    (existsb
        (λ is:list input, bool_decide (witness is x y))
        (lists_shorter_than (max_access_len T access_T))
        ) eqn:E.
- left. apply Is_true_eq_left in E. apply existb_True in E.
  apply Exists_exists in E. destruct E as (is & I & W). exists is.
  apply bool_decide_unpack in W. apply W.
- right. intro. destruct H1 as (is & W).
  elim tree_max_len with access_T is x.
  + done.
  + apply existsb_false_forall with is in E. exfalso. apply E.
    apply bool_decide_pack. apply W. apply In_lists_shorter_than.
    apply rev_tree_max_len with x. done. destruct W as (Sx1 & Sx & N).
    done.
```

```
  + destruct W. done.
Qed.

Let new_transition_fits : ((HS * input) * HS) -> Prop :=
  λ '(q1↾_, i, q2↾_),
    match delta T q1 i with
    | None => True
    | Some q1' => ¬ (q1' # q2) /\ frontier q1'
    end.

Instance new_transition_fits_dec :
∀ x : HS * input * HS, Decision (new_transition_fits x).
Proof.
intros. unfold new_transition_fits.
destruct x as (((q1&_)&i)&(q2&_)), (delta T (q1) i).
all: solve_decision.
Qed.

Definition find_output : HS -> input -> output
  := λ q i, match lambda T (`q) i with
  | Some o => o
  | None => inhabitant
  end.

Definition add_output:
  ((HS * input) * HS) -> ((HS * input) * (HS * output))
  := λ '(q,i,p), ((q, i), (p, find_output q i)).

Lemma add_output_preserves : ∀ q1 i q2 o p,
  (q1, i, (q2, o)) = add_output p -> p = (q1,i,q2).
Proof.
intros q1 i q2 o p A0. unfold add_output in A0.
destruct p as ((p11&p12)&p2). inversion A0. done.
Qed.

Let h_list := add_output <$> (filter
    (λ x, transition_in_basis x \/ new_transition_fits x)
    (enum (HS * input * HS))).

Let h: gmap (HS * input) (HS * output) :=
  list_to_map h_list.

Lemma h_preserves_transition :
  ∀ (q q1: T) (e:basis q) (e1:basis q1) (i:input) (o:output),
  transition T !! (q, i) = Some (q1, o) ->
  h !! ((q↾e):HS, i) = Some ((q1 ↾ e1):HS, o).
Proof.
intros. unfold h. apply elem_of_list_to_map'.
- intros. destruct x' as (q2 & o1). destruct q2 as (q2 & e2).
  apply elem_of_list_In in H2, H3. apply in_map_iff in H2,H3.
  destruct H2 as ((p1&q1')&F1&I1), H3 as ((p2&q2')&F2&I2).
  destruct p1 as (x1, i1), p2 as (x2, i2).
  unfold add_output in F1, F2. simpl in F1, F2. inversion F1.
  inversion F2. apply f_equal2. 2:done. apply HS_eq.
  rewrite H7, H8, H9 in I2. apply elem_of_list_In in I2.
  apply elem_of_list_filter in I2. destruct I2 as (D & _). destruct D.
  + unfold transition_in_basis in H2. destruct H2 as (D&_).
    unfold delta in D. rewrite H1 in D. inversion D. tauto.
  + unfold new_transition_fits in H2. unfold delta in H2. simpl in H2.
    rewrite H1 in H2. simpl in H2. destruct H2 as (_ & Fq1).
    destruct Fq1 as (Bq1 & _). elim Bq1. tauto.
- apply elem_of_list_In. apply in_map_iff.
```

```
  exists ((q ↾ e, i), q1 ↾ e1). split.
  + unfold add_output, find_output. simpl. unfold lambda.
    rewrite H1. tauto.
  + apply elem_of_list_In. apply elem_of_list_filter.
    split. 2:apply elem_of_enum. left. unfold transition_in_basis.
    split. 2:done. unfold delta. rewrite H1. tauto.
Qed.

Lemma list_to_map_prop `{EqDecision K} `{Countable K} {A:Type} :
  ∀ (P:A->Prop) (l:list (K*A)) (k:K),
  let M := list_to_map l:gmap K A in
  (∀ a, (k,a) ∈ l -> P a) -> is_Some (M !! k) ->
  ∃ a, M !! k = Some a /\ P a.
Proof.
intros. destruct H3 as (a & H3). exists a. split. done. apply H2.
apply elem_of_list_to_map_2. unfold M in H3. tauto.
Qed.

Lemma h_contains_basis : contains_basis h.
Proof.
unfold contains_basis. intros. remember (access_T q) as is.
revert Heqis e. revert q. induction is using rev_ind.
- intros. simpl. apply f_equal. apply HS_eq. pose proof tree0 q.
  destruct H1. rewrite <- Heqis in H1. inversion H1. done.
- intros. destruct (repeat_delta (q0 T) is) eqn:RDis.
  cut (Is_true (basis q1)). intro e1.
  rewrite <- repeat_delta_app_Some with is [x] (q0 (Hy h)) (q1 ↾ e1).
  + destruct tree0 with q. rewrite <- Heqis in H1.
    rewrite <- repeat_delta_app_Some with is [x] (q0 T) q1 in H1.
    2:done. simpl in H1. unfold delta in H1.
    destruct (transition T !! (q1, x)) eqn:tq1. 2:done.
    destruct p. simpl in H1. inversion H1. simpl in H4.
    rewrite H4 in tq1. simpl. unfold delta. simpl.
    rewrite h_preserves_transition with q1 q e1 e x o; done.
  + apply IHis. destruct tree0 with q1. apply H2. done.
  + destruct tree0 with q. apply basis_tree with q x. 2:done.
    rewrite <- Heqis in H1.
    rewrite <- repeat_delta_app_Some with is [x] (q0 T) q1 in H1.
    2:done. simpl in H1. destruct (delta T q1 x); tauto.
  + destruct tree0 with q. rewrite <- Heqis in H1.
    rewrite repeat_delta_app_None with is [x] (q0 T) in H1; done.
Qed.

Lemma h_complete :
  (∀(q:T) (Fq:frontier q), ¬isolated q Fq) -> complete (Hy h).
Proof.
unfold complete. intros. simpl. unfold is_Some.
destruct (h !! (q,i)) eqn:H2. exists p; tauto. exfalso.
apply not_elem_of_list_to_map in H2. apply H2.
apply elem_of_list_fmap. destruct (transition T !! (`q, i)) eqn:Tqi.
destruct p, (basis q1) eqn:e1.
- apply Is_true_true_2 in e1. exists ((q, i), (q1 ↾ e1, o)).
  split. done. apply elem_of_list_In. apply in_map_iff.
  exists (q,i,q1 ↾ e1). split.
  { unfold add_output. simpl. repeat (apply f_equal2); try tauto.
    unfold find_output. unfold lambda. rewrite Tqi. tauto. }
  apply elem_of_list_In. apply elem_of_list_filter.
  split. 2:apply elem_of_enum. left. destruct q as (q&e).
  split. 2:tauto. unfold delta. simpl in Tqi. rewrite Tqi. tauto.
- assert (frontier q1) as Fq1.
  { split. apply Is_true_false. tauto. exists (`q). exists i.
    split. destruct q. tauto. unfold delta. rewrite Tqi. tauto. }
```

```
  specialize H1 with q1 Fq1. unfold isolated in H1.
  assert (∃ q2, ¬ (¬ basis q2 \/ q1 # q2)).
  { apply Exists_finite.
    replace (λ x : T, ¬ (¬ basis x \/ q1 # x))
        with (not ∘ λ x, ¬ basis x \/ q1 # x).
    2:reflexivity. apply not_Forall_Exists. solve_decision. intro.
    apply H1. rewrite Forall_finite in H3. tauto. }
  destruct H3 as (q2 & H3). apply Decidable.not_or in H3.
  destruct H3 as (H3 & NA). destruct (basis q2) eqn:Bq2.
  2:naive_solver. apply Is_true_true in Bq2.
  exists (q, i, ((q2↾Bq2), o)). split. done. apply elem_of_list_fmap.
  exists (q,i,q2 ↾ Bq2). split.
  + unfold add_output. simpl. unfold find_output. unfold lambda.
    rewrite Tqi. tauto.
  + apply elem_of_list_filter. split. 2:apply elem_of_enum.
    right. unfold new_transition_fits. unfold delta. simpl.
    destruct q as (q&e). simpl in Tqi. rewrite Tqi. split; tauto.
- exists ((q, i), (q, inhabitant)). split. done.
  apply elem_of_list_fmap. exists (q,i,q). split.
  + unfold add_output. simpl. unfold find_output. unfold lambda.
    rewrite Tqi. tauto.
  + apply elem_of_list_filter. split. 2:apply elem_of_enum.
    right. unfold new_transition_fits. unfold delta. simpl.
    destruct q as (q&e). simpl in Tqi. rewrite Tqi. done.
Qed.

Lemma h_preserves_output :
  (∀(q:T) (Fq:frontier q), ¬isolated q Fq) ->
  (∀q e i, is_Some (transition T !! (q,i)) ->
    lambda T q i = lambda (Hy h) (q↾e) i).
Proof.
intros. destruct H2 as (p&tq). destruct p. unfold lambda. rewrite tq.
simpl. destruct (basis q1) eqn:e1.
+ apply Is_true_eq_left in e1.
  rewrite h_preserves_transition with q q1 e e1 i o; tauto.
+ assert (frontier q1).
  { split. apply Is_true_false. tauto. exists q, i.
    split; auto. unfold delta. rewrite tq. tauto. }
  unfold h. assert (H4:= H1 q1 H2). unfold isolated in H4. symmetry.
  apply fmap_Some. apply list_to_map_prop.
  * intros. apply elem_of_list_fmap in H3.
    destruct H3 as ((p, q2) & A & _). destruct p as (qe, i2).
    unfold add_output in A. simpl in A. inversion A. simpl.
    unfold find_output, lambda. simplify_eq. simpl. rewrite tq. tauto.
  * apply h_complete. tauto.
Qed.

(* Lemma 3.6 part 1 *)
Theorem hypothesis_existence :
  (∀(q:T) (Fq:frontier q), ¬isolated q Fq) -> (∃ h, hypothesis h).
Proof.
intros FI. exists h. repeat split.
apply h_contains_basis. apply h_complete. tauto.

intros q p i o e tqT. destruct (basis p) eqn:Bp.
- apply Is_true_eq_left in Bp. exists (p↾Bp). split_and.
  2: apply apart_irreflexive. simpl.
  rewrite h_preserves_transition with q p e Bp i o. tauto. tauto.
- assert (frontier p) as Fp.
  { split. apply Is_true_false. tauto. exists q,i.
    split. done. unfold delta. rewrite tqT. tauto. }
  destruct (h_complete FI (q↾e:HS) i). destruct x as (p' & o').
```

```
  exists p'. split.
  + assert (lambda (Hy h) (q↾e) i = Some o').
    { unfold lambda. rewrite H1. tauto. }
    rewrite <- h_preserves_output in H2. unfold lambda in H2.
    rewrite tqT in H2. simpl in H2. inversion H2. tauto. tauto.
    exists (p,o). tauto.
  + simpl in H1. unfold h in H1. apply elem_of_list_to_map_2 in H1.
    apply elem_of_list_fmap in H1. destruct H1 as (((q1&i1)&p1)&AO&IL).
    inversion AO. simplify_eq. simpl in *. simplify_eq.
    apply elem_of_list_filter in IL. destruct IL as (IL&_).
    destruct IL.
    * destruct p1, H1. simpl in H1. unfold delta in H1.
      rewrite tqT in H1. inversion H1. apply apart_irreflexive.
    * unfold new_transition_fits in H1. unfold delta in H1.
      simpl in H1. rewrite tqT in H1. simpl in H1.
      destruct p1 as (p1&e'), H1. tauto.
Qed.

Definition frontier_identified :=
  (∀(q:T) (Fq:frontier q), identified q Fq).

Lemma h_list_functional : basis_complete -> frontier_identified ->
    ∀ x y1 y2, (x, y1) ∈ h_list -> (x, y2) ∈ h_list -> y1 = y2.
Proof.
intros BC FI x y1 y2 IL1 IL2. unfold h_list in IL1, IL2.
destruct x as ((q&e) & i), y1 as (q1 & o1), y2 as (q2 & o2).
apply elem_of_list_fmap in IL1,IL2.
destruct IL1 as (y1&AO1&IL1), IL2 as (y2&AO2&IL2).
pose proof (add_output_preserves (q↾e) i q1 o1 y1 AO1).
pose proof (add_output_preserves (q↾e) i q2 o2 y2 AO2).
simplify_eq. apply f_equal2.
- unfold basis_complete in  BC. destruct (BC q i e) as ((q'&o) & Tqi).
  apply elem_of_list_filter in IL1,IL2.
  destruct IL1 as (IL1&_), IL2 as (IL2&_).
  destruct IL1, IL2;
  unfold transition_in_basis, new_transition_fits, delta in H1,H2.
  + destruct q1 as (q1&e1), q2 as (q2&e2).
    apply HS_eq. destruct H1,H2. rewrite H1 in H2. inversion H2. done.
  + exfalso. simpl in H2. rewrite Tqi in H1,H2. simpl in H1,H2.
    destruct q1 as (q1&e1), H1 as (H1&_). inversion H1. simplify_eq.
    destruct q2 as (q2&e2), H2 as (_&(H2&_&_)). done.
  + exfalso. simpl in H1. rewrite Tqi in H1,H2. simpl in H1,H2.
    destruct q2 as (q2&e2), H2 as (H2&_). inversion H2. simplify_eq.
    destruct q1 as (q1&e1), H1 as (_&(H1&_&_)). done.
  + simpl in H1,H2. rewrite Tqi in H1,H2.
    destruct q1 as (q1&e1), q2 as (q2,e2). simpl in H1,H2.
    destruct H1 as (NA1&F), H2 as (NA2&_). apply HS_eq.
    destruct FI with q' as (x&_&_&D). tauto.
    destruct (D q1 e1), (D q2 e2); try done. simplify_eq. tauto.
- unfold add_output in AO1,AO2. inversion AO1. inversion AO2. done.
Qed.

Lemma NoDup_h_list :
  basis_complete -> frontier_identified -> NoDup h_list.*1.
Proof.
intros BC FI. apply NoDup_fmap_fst. apply h_list_functional; done.
apply NoDup_fmap.
- unfold Inj. intros p1 p2 AO.
  destruct p1 as ((x1&i1)&y1), p2 as ((x2&i2)&y2).
  unfold add_output in AO. simpl in AO. inversion AO. tauto.
- apply list.NoDup_filter. apply NoDup_enum.
Qed.
```

```
  (* Lemma 3.6 part 2 *)
  Theorem hypothesis_unique (BC:basis_complete) (FI:frontier_identified):
    ∃ h, ∀ h0, hypothesis h0 -> h=h0.
  Proof.
  exists h. intros h0 Hh0. apply map_eq. intro i.
  unfold hypothesis in Hh0. destruct Hh0 as (CB & C & O), i as (Hq & i).
  destruct Hq as (q & e), (BC q i e) as ((q1,o) & Tqi).
  destruct (basis q1) eqn:Bq1.
  - apply Is_true_true in Bq1.
    rewrite h_preserves_transition with q q1 e Bq1 i o. 2:done.
    unfold contains_basis in CB.
    assert (lambda (Hy h0) (q ↾ e) i = Some o).
    { destruct (O q q1 i o e Tqi). destruct H1. unfold lambda.
      rewrite H1. done. }
    assert (repeat_delta (q ↾ e:Q (Hy h0)) [i] = Some (q1↾Bq1:HS)).
    { rewrite repeat_delta_app_Some
        with (access_T q) [i] (q0 (Hy h0)) (q↾e).
      assert (access_T q ++ [i] = access_T q1). apply tree0.
      rewrite <- repeat_delta_app_Some with (access_T q) [i] (q0 T) q.
      unfold repeat_delta. unfold delta. rewrite Tqi. tauto.
      apply tree0. rewrite H2. apply CB. apply CB. }
    unfold repeat_delta in H2. destruct (delta (Hy h0) (q↾e) i) eqn:D;
    inversion H2. rewrite H4 in D. unfold delta in D.
    unfold lambda in H1.
    destruct (transition (Hy h0) !! (q ↾ e:(Q (Hy h0)), i)) eqn:DH.
    2:inversion D. simpl in DH. rewrite DH. simpl in H1,D. destruct p.
    inversion H1. inversion D. tauto.
  - assert (frontier q1) as Fq1.
    { split. apply Is_true_false. tauto. exists q. exists i.
      split. tauto. unfold delta. rewrite Tqi. tauto. }
    destruct (FI q1 Fq1) as (q2 & e2 & NA & I).
    assert (h !! (q ↾ e:HS, i) = Some (q2↾e2:HS, o)).
    { unfold h. apply elem_of_list_to_map. apply (NoDup_h_list BC FI).
      unfold h_list. apply elem_of_list_fmap.
      exists (q ↾ e, i, q2 ↾ e2). split.
      + unfold add_output. simpl. repeat apply f_equal2; try done.
        unfold find_output, lambda. simpl. rewrite Tqi. done.
      + apply elem_of_list_filter. split. 2: apply elem_of_enum.
        right. unfold new_transition_fits. simpl. unfold delta.
        rewrite Tqi. done.
    }
    rewrite H1. destruct (O q q1 i o e Tqi) as (q3 & Tr2 & NA2).
    simpl in Tr2. rewrite Tr2. apply f_equal. apply f_equal2. 2:tauto.
    destruct q3 as (q3 & e3). simpl in NA2. destruct (I q3 e3).
    + done.
    + apply HS_eq. done.
  Qed.
End hypothesis_construction.

Section teacher.
  Context `{Finite input} (output : Type).

  Inductive equivAns :=
    | equiv : equivAns
    | nonEquiv : list input -> equivAns.

  Record teacher := Teacher {
    outputQuery : list input -> list output;
    equivQuery : ∀ (H:mealy input output) `(complete H), equivAns;
    teacherConsistent : ∃ (M:mealy input output),
      complete M
```

```
      /\ ∀ is : list input, Some (outputQuery is) = sem (q0 M) is
      /\ ∀ (H:mealy input output) (CH: complete H),
        match (equivQuery H CH) with
        | equiv => mealy_equiv M H
        | nonEquiv is => sem (q0 M) <> sem (q0 H)
        end
  }.
End teacher.
```