Optimizing automata learning via monads

GERCO VAN HEERDT, MATTEO SAMMARTINO, and ALEXANDRA SILVA, University College London

Automata learning has been successfully applied in the verification of hardware and software. The size of the automaton model learned is a bottleneck for scalability, and hence optimizations that enable learning of compact representations are important. This paper exploits monads, both as a mathematical structure and a programming construct, to design, prove correct, and implement a wide class of such optimizations. The former perspective on monads allows us to develop a new algorithm and accompanying correctness proofs, building upon a general framework for automata learning based on category theory. The new algorithm is parametric on a monad, which provides a rich algebraic structure to capture non-determinism and other side-effects. We show that our approach allows us to uniformly capture existing algorithms, develop new ones, and add optimizations. The latter perspective allows us to effortlessly translate the theory into practice: we provide a Haskell library implementing our general framework, and we show experimental results for two specific instances: non-deterministic and weighted automata.

Additional Key Words and Phrases: automata, learning, side-effects, monads, algebras

1 INTRODUCTION

The increasing complexity of software and hardware systems calls for new scalable methods to design, verify, and continuously improve systems. Black-box inference methods aim at building models of running systems by observing their response to certain queries. This reverse engineering process is very amenable for automation and allows for fine-tuning the precision of the model depending on the properties of interest, which is important for scalability.

One of the most successful instances of black-box inference is automata learning, which has been used in various verification tasks, ranging from finding bugs in implementations of network protocols [de Ruiter and Poll 2015] to rejuvenating legacy software [Schuts et al. 2016]. Vaandrager [2017] has recently written a comprehensive overview of the widespread use of automata learning in verification.

A limitation in automata learning is that the models of real systems can become too large to be handled by tools. This demands compositional methods and techniques that enable compact representation of behaviors.

In this paper, we show how monads can be used to add optimizations to learning algorithms in order to obtain compact representations. We will use as playground for our approach the well known $\mathsf{L}^\star$ algorithm [Angluin 1987], which learns a minimal deterministic finite automaton (DFA) accepting a regular language by interacting with a *teacher*, i.e., an oracle that can reply to specific queries about the target language. Monads allow us to take an abstract approach, in which category theory is used to devise an optimized learning algorithm and a generic correctness proof for a broad class of compact models. Monads also allow us to straightforwardly implement the algorithm in Haskell via the corresponding programming constructs.

The inspiration for this work is quite concrete: it is a well-known fact that non-deterministic finite automata (NFAs) can be much smaller than deterministic ones for a regular language. The subtle point is that given a regular language, there is a canonical deterministic automaton accepting it—the minimal one—but there might be many "minimal" non-deterministic automata accepting the same language. This raises a challenge for learning algorithms: which non-deterministic automaton should the algorithm learn? To overcome this, Bollig et al. [2009] developed a version of Angluin's

Authors' address: Gerco van Heerdt, gerco.heerdt@ucl.ac.uk; Matteo Sammartino, m.sammartino@ucl.ac.uk; Alexandra Silva, alexandra.silva@ucl.ac.uk, University College London.

$\mathsf{L}^\star$ algorithm, which they called $\mathsf{NL}^\star$, in which they use a particular class of NFAs, namely *Residual Finite State Automata* (RFSAs), which do admit minimal canonical representatives. Though $\mathsf{NL}^\star$ indeed is a first step in incorporating a more compact representation of regular languages, there are several questions that remain to be addressed. We tackle them in this paper.

DFAs and NFAs are formally connected by the subset construction. Underpinning this construction is the rich algebraic structure of languages and of the state space of the DFA obtained by determinizing an NFA. The state space of a determinized DFA—consisting of subsets of the state space of the original NFA—has a join-semilattice structure. Moreover, this structure is preserved in language acceptance: if there are subsets U and V, then the language of $U \cup V$ is the union of the languages of the first two. Formally, the function that assigns to each state its language is a join-semilattice map, since languages themselves are just sets of words and have a lattice structure. And languages are even richer: they have the structure of complete atomic Boolean algebras. This leads to several questions: Can we exploit this structure and have even more compact representations? What if we slightly change the setting and look at weighted languages over a semiring, which have the structure of a semimodule (or vector space, if the semiring is a field)?

The latter question is strongly motivated by the widespread use of weighted languages and corresponding *weighted finite automata* (WFAs) in verification, from the formal verification of quantitative properties [Chatterjee et al. 2008; Droste and Gastin 2005; Kuperberg 2014], to probabilistic model-checking [Baier et al. 2009], to the verification of on-line algorithms [Aminof et al. 2010].

Our key insight is that the algebraic structures mentioned above are in fact algebras for a monad T. In the case of join-semilattices this is the powerset monad, and in the case of vector spaces it is the free vector space monad. These monads can be used to define a notion of T-automaton, with states having the structure of an algebra for the monad T, which generalizes non-determinism as a side-effect. From T-automata we can derive a compact, equivalent version by taking as states a set of *generators* and transferring the algebraic structure of the original state space to the transition structure of the automaton.

This general perspective enables us to generalize $\mathsf{L}^\star$ to a new algorithm $\mathsf{L}^\star_T$, which learns compact automata featuring non-determinism and other side-effects captured by a monad. Moreover, $\mathsf{L}^\star_T$ incorporates further optimizations arising from the monadic representation, which lead to more scalable algorithms.

Besides the theoretical aspects, we devote large part of this paper to implementation and experimental evaluation. Monads are key for us to faithfully translate theory into practice. We provide a library that implements all aspects of our general framework, making use of Haskell monads.[1] For any monad, the library allows the programmer to obtain a basic, correct-by-construction instance of the algorithm and of its optimized versions for free. This enables the programmer to experiment with different optimizations with minimal effort. Our library also allows the programmer to redefine some basic operations, if a more efficient version is available, in order to make the algorithm more amenable to real-world usage. For instance, generators can be computed efficiently in the vector space case via Gaussian elimination.

One of the main challenges in applying Angluin-style algorithms to real-world systems is implementing the teacher. In fact, it is often the case that exact answers to certain queries are not available. In these cases the teacher often resorts to *random testing* [e.g., Aarts et al. 2013; Chalupar et al. 2014; Cho et al. 2010], with an unavoidable trade-off in terms of model accuracy (see [Vaandrager 2017] for a detailed discussion on this issue). Our library provides support for both exact and approximate teachers, along with a basic implementation that works for any monad.

[1] The code is provided as supplementary material.

Interestingly, the exact teacher relies on *bisimulation up to context* [Rot et al. 2013; Sangiorgi 1998], which exploits the monad structure to efficiently determine bisimulation.

2 OVERVIEW AND CONTRIBUTIONS

In this section, we give an overview of our approach and highlight our main contribution. We start by explaining the original $\mathsf{L}^\star$ algorithm. We then discuss the challenges in adapting the algorithm to learn automata with side-effects, illustrating them through a concrete example—NFAs.

2.1 $\mathsf{L}^\star$ algorithm

The $\mathsf{L}^\star$ algorithm learns the minimal DFA accepting a language $\mathcal{L} \subseteq A^*$ over a finite alphabet A. The algorithm assumes the existence of a *minimally adequate teacher*, which is an oracle that can answer two types of queries:

- **Membership queries**: given a word $w \in A^*$, does w belong to $\mathcal{L}$?
- **Equivalence queries**: given a *hypothesis* DFA $\mathcal{H}$, does $\mathcal{H}$ accept $\mathcal{L}$? If not, the teacher will return a *counterexample*, i.e., a word incorrectly accepted or rejected by $\mathcal{H}$.

The algorithm incrementally builds an *observation table*. The table is made of two parts: a top part, with rows ranging over a finite set $S \subseteq A^*$; and a bottom part, with rows ranging over $S \cdot A$ (i.e., words of the form sa, with $s \in S$ and $a \in A$). Columns range over a finite $E \subseteq A^*$. For each $u \in S \cup S \cdot A$ and $v \in E$, the corresponding cell in the table contains 1 if and only if $uv \in \mathcal{L}$. Intuitively, each row u contains enough information to fully identify the Myhill-Nerode equivalence class of u with respect to an approximation of the target language—rows with the same content are considered members of the same equivalence class. Cells are filled in by the algorithm via membership queries.

As an example, and to set notation, consider the table below over $A = \{a, b\}$. It shows that $\mathcal{L}$ contains the word aa and does not contain the words ε (the empty word), a, b, ba, aaa, and baa.

		E		
		ε	a	aa
S	ε	0	0	1
$S \cdot A$	a	0	1	0
	b	0	0	0

$$\mathrm{row}_{\mathrm{t}} \colon S \to 2^E \qquad \mathrm{row}_{\mathrm{t}}(u)(v) = 1 \iff uv \in \mathcal{L}$$

$$\mathrm{row}_{\mathrm{b}} \colon S \to (2^E)^A \qquad \mathrm{row}_{\mathrm{b}}(u)(a)(v) = 1 \iff uav \in \mathcal{L}$$

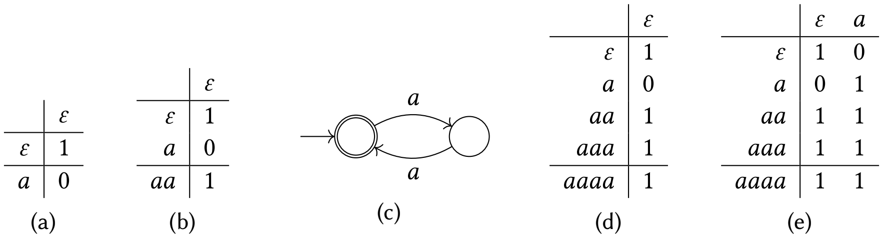
We use the functions $\mathrm{row}_{\mathrm{t}}$ and $\mathrm{row}_{\mathrm{b}}$ to describe the top and bottom parts of the table, respectively. Notice that S and $S \cdot A$ may intersect. For the sake of conciseness, when tables are depicted, elements in the intersection are only shown in the top part.

A key idea of the algorithm is to construct a hypothesis DFA from the different rows in the table. The construction is the same as that of the minimal DFA from the Myhill-Nerode equivalence, and exploits the correspondence between table rows and Myhill-Nerode equivalence classes. The state space of the hypothesis DFA is given by the set $H = \{\mathrm{row}_{\mathrm{t}}(s) \mid s \in S\}$. Note that there may be multiple rows with the same content, but they result in a single state, as they all belong to the same Myhill-Nerode equivalence class. The initial state is $\mathrm{row}_{\mathrm{t}}(\varepsilon)$, and we use the ε column to determine whether a state is accepting: $\mathrm{row}_{\mathrm{t}}(s)$ is accepting whenever $\mathrm{row}_{\mathrm{t}}(s)(\varepsilon) = 1$. The transition function is defined as $\mathrm{row}_{\mathrm{t}}(s) \xrightarrow{a} \mathrm{row}_{\mathrm{b}}(s)(a)$. (Notice that the continuation is drawn from the bottom part of the table). For the hypothesis automaton to be well-defined, ε must be in S and E, and the table must satisfy two properties:

- **Closedness** states that each transition actually leads to a state of the hypothesis. That is, the table is closed if for all $t \in S$ and $a \in A$ there is $s \in S$ such that $\mathrm{row}_{\mathrm{t}}(s) = \mathrm{row}_{\mathrm{b}}(t)(a)$.
- **Consistency** states that there is no ambiguity in determining the transitions. That is, the table is consistent if for all $s_1, s_2 \in S$ such that $\mathrm{row}_{\mathrm{t}}(s_1) = \mathrm{row}_{\mathrm{t}}(s_2)$ we have $\mathrm{row}_{\mathrm{b}}(s_1) = \mathrm{row}_{\mathrm{b}}(s_2)$.

```
$S, E \leftarrow \{\varepsilon\}$
repeat
    while the table is not closed or not consistent
        if the table is not closed
            find $t \in S, a \in A$ such that $\mathrm{row_b}(t)(a) \neq \mathrm{row_t}(s)$ for all $s \in S$
            $S \leftarrow S \cup \{ta\}$
        if the table is not consistent
            find $s_1, s_2 \in S$, $a \in A$, and $e \in E$ such that
                $\mathrm{row_t}(s_1) = \mathrm{row_t}(s_2)$ and $\mathrm{row_b}(s_1)(a)(e) \neq \mathrm{row_b}(s_2)(a)(e)$
            $E \leftarrow E \cup \{ae\}$
    Construct the hypothesis $\mathcal{H}$ and submit it to the teacher
    if the teacher replies no, with a counterexample $z$
        $S \leftarrow S \cup \mathrm{prefixes}(z)$
until the teacher replies yes
return $\mathcal{H}$
```

Fig. 1. L★ algorithm.



(a)

	ε
ε	1
a	0

(b)

	ε
ε	1
a	0
aa	1

(c)

(d)

	ε
ε	1
a	0
aa	1
aaa	1
$aaaa$	1

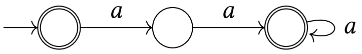
(e)

	ε	a
ε	1	0
a	0	1
aa	1	1
aaa	1	1
$aaaa$	1	1

Fig. 2. Example run of L★ on $\mathcal{L} = \{w \in \{a\}^* \mid |w| \neq 1\}$.

The algorithm updates the sets S and E to satisfy these properties, constructs a hypothesis, submits it in an equivalence query, and, when given a counterexample, refines the hypothesis. This process continues until the hypothesis is correct. The algorithm is shown in Figure 1.

Example Run. We now run the algorithm with the target language $\mathcal{L} = \{w \in \{a\}^* \mid |w| \neq 1\}$. The minimal DFA accepting $\mathcal{L}$ is

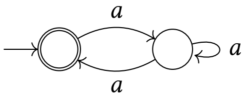
$\mathcal{M} =$  (1)

Initially, $S = E = \{\varepsilon\}$. We build the observation table given in Figure 2a. This table is not closed, because the row with label a, having 0 in the only column, does not appear in the top part of the table: the only row ε has 1. To fix this, we add the word a to the set S. Now the table (Figure 2b) is closed and consistent, so we construct the hypothesis that is shown in Figure 2c and pose an equivalence query. The teacher replies *no* and informs us that the word aaa should have been accepted. L★ handles a counterexample by adding all its prefixes to the set S. We only have to add aa and aaa in this case. The next table (Figure 2d) is closed, but not consistent: the rows ε and aa both have value 1, but their extensions a and aaa differ. To fix this, we prepend the continuation a to the column ε on which they differ and add $a \cdot \varepsilon = a$ to E. This distinguishes $\mathrm{row_t}(\varepsilon)$ from $\mathrm{row_t}(aa)$, as seen in the next table in Figure 2e. The table is now closed and consistent, and the new hypothesis automaton is precisely the correct one $\mathcal{M}$.

As mentioned, the hypothesis construction approximates the theoretical construction of the minimal DFA, which is unique up to isomorphism. That is, for $S = E = A^*$ the relation that identifies words of S having the same value in row_t is precisely the Myhill-Nerode's right congruence.

2.2 Learning non-deterministic automata

As it is well known, NFAs can be smaller than the minimal DFA for a given language. For example, the language $\mathcal{L}$ above is accepted by the NFA

$\mathcal{N} =$  (2)

which is smaller than the minimal DFA $\mathcal{M}$. Though in this example, which we chose for simplicity, the state reduction is not massive, it is known that in general NFAs can be exponentially smaller than the minimal DFA [Kozen 2012]. This reduction of the state space is enabled by a side-effect—non-determinism, in this case.

Learning NFAs can lead to a substantial gain in space complexity, but it is challenging. The main difficulty is that NFAs do not have a canonical minimal representative: there may be several non-isomorphic state-minimal NFAs accepting the same language, which poses problems for the development of the learning algorithm. To overcome this, Bollig et al. [2009] proposed to use a particular class of NFAs, namely RFSAs, which do admit minimal canonical representatives. However, their ad-hoc solution for NFAs does not extend to other automata, such as weighted or alternating. In this paper we present a solution that works for any side-effect, specified as a monad.

The crucial observation underlying our approach is that the language semantics of an NFA is defined in terms of its determinization, i.e., the DFA obtained by taking sets of states of the NFA as state space. In other words, this DFA is defined over an algebraic structure induced by the powerset, namely a *join semilattice* (JSL) whose join operation is set union. This automaton model does admit minimal representatives, which leads to the key idea for our algorithm: learning NFAs as automata over JSLs. In order to do so, we use an extended table where rows have a JSL structure, defined as follows. The join of two rows is given by an element-wise or, and the bottom element is the row containing only zeroes. More precisely, the new table consists of the two functions

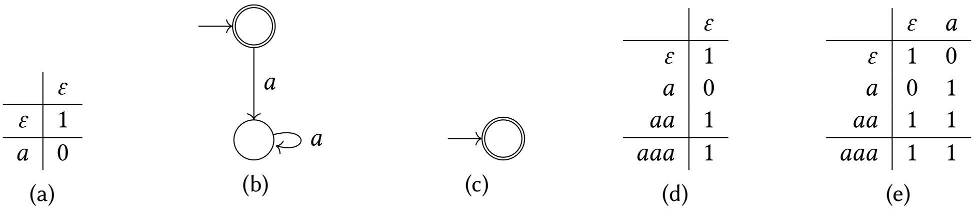
$$\mathsf{row}_\mathsf{t}^\sharp \colon \mathcal{P}(S) \to 2^E \qquad \mathsf{row}_\mathsf{b}^\sharp \colon \mathcal{P}(S) \to (2^E)^A$$

given by $\mathsf{row}_\mathsf{t}^\sharp(U) = \bigvee\{\mathsf{row}_\mathsf{t}(s) \mid s \in U\}$ and $\mathsf{row}_\mathsf{b}^\sharp(U)(a) = \bigvee\{\mathsf{row}_\mathsf{b}(s)(a) \mid s \in U\}$. Formally, these functions are JSL homomorphisms, and they induce the following general definitions:

- The table is *closed* if for all $U \subseteq S$, $a \in A$ there is $U' \subseteq S$ such that $\mathsf{row}_\mathsf{t}^\sharp(U') = \mathsf{row}_\mathsf{b}^\sharp(U)(a)$.
- The table is *consistent* if for all $U_1, U_2 \subseteq S$ s.t. $\mathsf{row}_\mathsf{t}^\sharp(U_1) = \mathsf{row}_\mathsf{t}^\sharp(U_2)$ we have $\mathsf{row}_\mathsf{b}^\sharp(U_1) = \mathsf{row}_\mathsf{b}^\sharp(U_2)$.

We remark that our algorithm does not actually store the whole extended table, which can be quite large. It only needs to store the original table over S because all other rows in $\mathcal{P}(S)$ are freely generated and can be computed as needed, with no additional membership queries. The only lines in Figure 1 that need to be adjusted are lines 5 and 8, where closedness and consistency are replaced with the new notions given above. Moreover, $\mathcal{H}$ is now built from the extended table.

Optimizations. In this paper we also present two optimizations to our algorithm. For the first one, note that the state space of the hypothesis constructed by the algorithm can be very large since it encodes the entire algebraic structure. We show that we can extract a *minimal set of generators*



	ε
ε	1
a	0

(a)

(b)

(c)

	ε
ε	1
a	0
aa	1
aaa	1

(d)

	ε	a
ε	1	0
a	0	1
aa	1	1
aaa	1	1

(e)

Fig. 3. Example run of the $\mathsf{L}^\star$ adaptation for NFAs on $\mathcal{L} = \{w \in \{a\}^* \mid |w| \neq 1\}$.

from the table and compute a *succinct hypothesis* in the form of an automaton with side-effects, without any algebraic structure. For JSLs, this consists in only taking rows that are not the join of other rows, i.e., the *join-irreducibles*. By applying this optimization to this specific case, we essentially recover the learning algorithm of Bollig et al. [2009]. The second optimization is a generalization of the optimized counterexample handling method of Rivest and Schapire [1993], originally intended for $\mathsf{L}^\star$ and DFAs. It consists in processing counterexamples by adding a single *suffix* of the counterexample to E, instead of adding all prefixes of the counterexample to S. This can avoid the algorithm posing a large number of membership queries.

Example Revisited. We now run the new algorithm on the language $\mathcal{L} = \{w \in \{a\}^* \mid |w| \neq 1\}$ considered earlier. Starting from $S = E = \{\varepsilon\}$, the observation table (Figure 3a) is immediately closed and consistent. (It is closed because we have $\mathsf{row}^\sharp_\mathsf{t}(\{a\}) = \mathsf{row}^\sharp_\mathsf{t}(\emptyset)$.) This gives the JSL hypothesis shown in Figure 3b, which leads to an NFA hypothesis having a single state that is initial, accepting, and has no transitions (Figure 3c). The hypothesis is obviously incorrect, and the teacher may supply us with counterexample aa. Adding prefixes a and aa to S leads to the table in Figure 3d. The table is again closed, but not consistent: $\mathsf{row}^\sharp_\mathsf{t}(\{a\}) = \mathsf{row}^\sharp_\mathsf{t}(\emptyset)$, but $\mathsf{row}^\sharp_\mathsf{b}(\{a\})(a) = \mathsf{row}^\sharp_\mathsf{t}(\{aa\}) \neq \mathsf{row}^\sharp_\mathsf{t}(\emptyset) = \mathsf{row}^\sharp_\mathsf{b}(\emptyset)(a)$. Thus, we add a to E. The resulting table (Figure 3e) is closed and consistent. We note that row aa is the union of other rows: $\mathsf{row}^\sharp_\mathsf{t}(\{aa\}) = \mathsf{row}^\sharp_\mathsf{t}(\{\varepsilon, a\})$ (i.e., it is not a join-irreducible), and therefore can be ignored when building the succinct hypothesis. This hypothesis has two states, ε and a, and indeed it is the correct one $\mathcal{N}$.

2.3 Contributions and road map of the paper

After some preliminary notions in Section 3, our main contributions are presented as follows:

- In Section 4, we develop a general algorithm $\mathsf{L}^\star_T$, which generalizes the NFA one presented in Section 2.2 to an arbitrary *monad* T capturing side-effects, and we provide a general correctness proof for our algorithm.
- In Section 5, we describe the first optimization and prove its correctness.
- In Section 6 we describe the second optimization. We also show how it can be combined with the one of Section 5, and how it can lead to a further small optimization, where the consistency check on the table is dropped.
- In Section 7 we show how $\mathsf{L}^\star_T$ can be applied to several automata models, highlighting further case-specific optimizations when available.
- In Section 8 we describe our library and explain in detail how it can be instantiated to NFAs and WFAs. The implementation of monads for these two cases is non-trivial, due to specific Haskell requirements. We also give efficient versions of both instances. To the best of our knowledge, we are the first ones to implement an Angluin-style learning algorithm for WFAs, and to provide optimizations for it.

- Finally, in Section 9 we describe experimental results for the non-deterministic and weighted cases, comparing all the optimizations enabled by our library. In particular, for NFAs we show that the Rivest and Schapire optimization, not available to Bollig et al. [2009], leads to an improvement in the number of membership queries, as happens in the DFA case.

3 PRELIMINARIES

In this section we define a notion of T-automaton, a generalization of non-deterministic finite automata parametric in a monad T. We assume familiarity with basic notions of category theory: functors (in the category **Set** of sets and functions) and natural transformations.

Side-effects and different notions of non-determinism can be conveniently captured as a *monad*. A monad $T = (T, \eta, \mu)$ is a triple consisting of an endofunctor T on **Set** and two natural transformations: a *unit* $\eta\colon \mathtt{Id} \Rightarrow T$ and a *multiplication* $\mu\colon T^2 \Rightarrow T$, which satisfy the compatibility laws $\mu \circ \eta_T = \mathrm{id}_T = \mu \circ T\eta$ and $\mu \circ \mu_T = \mu \circ T\mu$.

Example 3.1 (Monads). An example of a monad is the triple $(\mathcal{P}, \{-\}, \bigcup)$, where $\mathcal{P}$ denotes the powerset functor associating a collection of subsets to a set, $\{-\}$ is the singleton operation, and $\bigcup$ is just union of sets. Another example is the triple $(V(-), e, m)$, where $V(X)$ is the free semimodule (over a semiring $\mathbb{S}$) over X, namely $\{\varphi \mid \varphi\colon X \to \mathbb{S} \text{ having finite support}\}$. The support of a function $\varphi\colon X \to \mathbb{S}$ is the set of $x \in X$ such that $\varphi(x) \neq 0$. Then $e\colon X \to V(X)$ is the characteristic function for each $x \in X$, and $m\colon V(V(X)) \to V(X)$ is defined for $\varphi \in V(V(X))$ and $x \in X$ as $m(\varphi)(x) = \sum_{\psi \in V(X)} \varphi(\psi) \times \psi(x)$.

Given a monad T, a T-algebra is a pair (X, h) consisting of a carrier set X and a function $h\colon TX \to X$ such that $h \circ \mu_X = h \circ Th$ and $h \circ \eta_X = \mathrm{id}_X$. A T-homomorphism between two T-algebras (X, h) and (Y, k) is a function $f\colon X \to Y$ such that $f \circ h = k \circ Tf$. The abstract notion of T-algebra instantiates to expected notions, as illustrated in the following example.

Example 3.2 (Algebras for a monad). The $\mathcal{P}$-algebras are the (complete) join-semilattices, and their homomorphisms are join-preserving functions. If $\mathbb{S}$ is a field, V-algebras are vector spaces, and their homomorphisms are linear maps.

We will often refer to a T-algebra (X, h) as X if h is understood or if its specific definition is irrelevant. Given a set X, (TX, μ_X) is a T-algebra called the *free T-algebra* on X. One can build algebras pointwise for some operations. For instance, if Y is a set and (X, x) a T-algebra, then we have a T-algebra (X^Y, f), where $f\colon T(X^Y) \to X^Y$ is given by $f(W)(y) = (x \circ T(\mathrm{ev}_y))(W)$ and $\mathrm{ev}_y\colon X^Y \to X$ by $\mathrm{ev}_y(g) = g(y)$. If U and V are T-algebras and $f\colon U \to V$ is a T-algebra homomorphism, then the image $\mathtt{img}(f)$ of f is a T-algebra, with the T-algebra structure inherited from V.

The following proposition connects algebra homomorphisms from the free T-algebra on a set U to an algebra V with functions $U \to V$. We will make use of this later in the section.

Proposition 3.3. *Given a set U and a T-algebra (V, v), there is a bijective correspondence between T-algebra homomorphisms $TU \to V$ and functions $U \to V$: for a T-algebra homomorphism $f\colon TU \to V$, define $f^\dagger = f \circ \eta\colon U \to V$; for a function $g\colon U \to V$, define $g^\sharp = v \circ Tg\colon TU \to V$. Then $g^\sharp$ is a T-algebra homomorphism called the* free T-extension of *g, and we have $f^{\dagger\sharp} = f$ and $g^{\sharp\dagger} = g$.*

We now have all the ingredients to define our notion of automaton with side-effects and their language semantics. We fix a monad (T, η, μ) with T preserving finite sets, as well as a T-algebra O that models outputs of automata.

Definition 3.4 (T-automaton). A *T-automaton* is a quadruple $(Q, \delta\colon Q \to Q^A, \mathsf{out}\colon Q \to O, \mathsf{init} \in Q)$, where the *state space* Q is a T-algebra, the *transition map* δ and *output map* out are T-algebra homomorphisms, and init is the *initial state.*

Example 3.5. DFAs are $\mathtt{Id}$-automata when $O = 2 = \{0, 1\}$ is used to distinguish accepting from rejecting states. For the more general case of O being any set, DFAs generalize into *Moore automata.*

Example 3.6. Recall that $\mathcal{P}$-algebras are JSLs, and their homomorphisms are join-preserving functions. In a $\mathcal{P}$-automaton, Q is equipped with a join operation, and Q^A is a join-semilattice with pointwise join: $(f \vee g)(a) = f(a) \vee g(a)$ for $a \in A$. Since the automaton maps preserve joins, we have, in particular, $\delta(q_1 \vee q_2)(a) = \delta(q_1)(a) \vee \delta(q_2)(a)$. One can represent an NFA over a set of states S as a $\mathcal{P}$-automaton by taking $Q = (\mathcal{P}(S), \bigcup)$ and $O = 2$, the Boolean join-semilattice with the *or* operation as its join. Let $\mathsf{init} \subseteq S$ be the set of initial states and $\mathsf{out}\colon \mathcal{P}(Q) \to 2$ and $\delta\colon \mathcal{P}(S) \to \mathcal{P}(S)^A$ the respective extensions (Proposition 3.3) of the NFA's output and transition functions. The resulting $\mathcal{P}$-automaton is precisely the determinized version of the NFA.

More generally, an automaton with side-effects given by a monad T always represents a T-automaton with a free state space: by applying Proposition 3.3, we have the following.

Proposition 3.7. *A T-automaton of the form $((TX, \mu_X), \delta, \mathsf{out}, \mathsf{init})$, for any set X, is completely defined by the set X with the element $\mathsf{init} \in TX$ and functions*

$$\delta^\dagger : X \to (TX)^A \qquad\qquad \mathsf{out}^\dagger : X \to O.$$

We call such a T-automaton a *succinct* automaton, which we sometimes identify with the representation $(X, \delta^\dagger, \mathsf{out}^\dagger, \mathsf{init})$.

A *(generalized) language* is a function $\mathcal{L}\colon A^* \to O$. For every T-automaton we have an *observability* and a *reachability* map, telling respectively which state is reached by reading a given word and which language each state recognizes.

Definition 3.8 (Reachability/observability maps). The *reachability map* of a T-automaton $\mathcal{A}$ with state space Q is a function $r_\mathcal{A} : A^* \to Q$ inductively defined as follows: $r_\mathcal{A}(\varepsilon) = \mathsf{init}$ and $r_\mathcal{A}(ua) = \delta(r_\mathcal{A}(u))(a)$. The *observability map* of $\mathcal{A}$ is a function $o_\mathcal{A} : Q \to O^{A^*}$ inductively defined as follows: $o_\mathcal{A}(q)(\varepsilon) = \mathsf{out}(q)$ and $o_\mathcal{A}(q)(av) = o_\mathcal{A}(\delta(q)(a))(v)$.

The *language accepted by* $\mathcal{A}$ is the function $\mathcal{L}_\mathcal{A} : A^* \to O$ given by $\mathcal{L}_\mathcal{A} = o_\mathcal{A}(\mathsf{init}) = \mathsf{out}_\mathcal{A} \circ r_\mathcal{A}$.

Example 3.9. For an NFA $\mathcal{A}$ represented as a $\mathcal{P}$-automaton, as seen in Example 3.6, $o_\mathcal{A}(q)$ is the language of q in the traditional sense. Notice that q, in general, is a set of states: $o_\mathcal{A}(q)$ takes the union of languages of singleton states. The set $\mathcal{L}_\mathcal{A}$ is the language accepted by the initial states, i.e., the language of the whole NFA. The reachability map $r_\mathcal{A}(u)$ returns the set of states reached via all possible paths reading u.

Given a language $\mathcal{L}\colon A^* \to O$, there exists a (unique) *minimal T-automaton* $\mathcal{M}_\mathcal{L}$ accepting $\mathcal{L}$. Its existence follows from general facts see [see, e.g., van Heerdt 2016].

Definition 3.10 (Minimal T-automaton for $\mathcal{L}$). Let $t_\mathcal{L} : A^* \to O^{A^*}$ be the function giving the *residual languages* of $\mathcal{L}$, namely $t_\mathcal{L}(u) = \lambda v.\mathcal{L}(uv)$. The minimal T-automaton $\mathcal{M}_\mathcal{L}$ accepting $\mathcal{L}$ has state space $M = \mathtt{img}(t^\sharp_\mathcal{L})$, initial state $\mathsf{init} = t_\mathcal{L}(\varepsilon)$, and T-algebra homomorphisms $\mathsf{out}\colon M \to O$ and $\delta\colon M \to M^A$ given by $\mathsf{out}(t^\sharp_\mathcal{L}(U)) = \mathcal{L}(U)$ and $\delta(t^\sharp_\mathcal{L}(U))(a)(v) = t^\sharp_\mathcal{L}(U)(av)$.

In the following, we will also make use of the *minimal Moore automaton* accepting $\mathcal{L}$. Although this always exists—it is defined by instantiating Definition 3.10 with $T = \mathsf{Id}$—it need not be finite.

The following property says that finiteness of Moore automata and of T-automata accepting the same language are intimately related.

Proposition 3.11. *The minimal Moore automaton accepting $\mathcal{L}$ is finite if and only if the minimal T-automaton accepting $\mathcal{L}$ is finite.*

4 A GENERAL ALGORITHM

In this section we introduce our extension of $\mathsf{L}^\star$ to learn automata with side-effects. The algorithm is parametric in the notion of side-effect, represented as the monad T, and is therefore called $\mathsf{L}^\star_T$. We fix a language $\mathcal{L} \colon A^* \to O$ that is to be learned, and we assume that there is a finite T-automaton accepting $\mathcal{L}$. This assumption generalizes the requirement of $\mathsf{L}^\star$ that $\mathcal{L}$ is regular (i.e., accepted by a specific class of T-automata, see Example 3.5).

An observation table consists of a pair of functions

$$\mathrm{row}_{\mathrm{t}} \colon S \to O^E \qquad\qquad \mathrm{row}_{\mathrm{b}} \colon S \to (O^E)^A$$

given by $\mathrm{row}_{\mathrm{t}}(s)(e) = \mathcal{L}(se)$ and $\mathrm{row}_{\mathrm{b}}(s)(a)(e) = \mathcal{L}(sae)$, where $S, E \subseteq A^*$ are finite sets with $\varepsilon \in S \cap E$. For $O = 2$, we recover exactly the $\mathsf{L}^\star$ observation table. The key idea for $\mathsf{L}^\star_T$ is defining closedness and consistency over the free T-extensions of those functions.

Definition 4.1 (Closedness and Consistency). The table is *closed* if for all $U \in T(S)$ and $a \in A$ there exists a $U' \in T(S)$ such that $\mathrm{row}_{\mathrm{t}}^{\sharp}(U') = \mathrm{row}_{\mathrm{b}}^{\sharp}(U)(a)$. The table is *consistent* if for all $U_1, U_2 \in T(S)$ such that $\mathrm{row}_{\mathrm{t}}^{\sharp}(U_1) = \mathrm{row}_{\mathrm{t}}^{\sharp}(U_2)$ we have $\mathrm{row}_{\mathrm{b}}^{\sharp}(U_1) = \mathrm{row}_{\mathrm{b}}^{\sharp}(U_2)$.

For closedness, we do not need to check all elements of $T(S) \times A$ against elements of $T(S)$, but only those of $S \times A$, thanks to the following result.

Lemma 4.2. *If for all $s \in S$ and $a \in A$ there is $U \in T(S)$ such that $\mathrm{row}_{\mathrm{t}}^{\sharp}(U) = \mathrm{row}_{\mathrm{b}}(s)(a)$, then the table is closed.*

Proof. Let $m \colon \mathtt{img}(\mathrm{row}_{\mathrm{t}}^{\sharp}) \hookrightarrow O^E$ be the embedding of the image of $\mathrm{row}_{\mathrm{t}}^{\sharp}$ into its codomain. According to van Heerdt et al. [2017], the definition of closedness given in Definition 4.1 amounts to requiring the existence of a T-algebra homomorphism close making the following diagram commute:

$$\begin{tikzcd}
T(S) \arrow[d, dashed, "\mathrm{close}"'] \arrow[dr, "\mathrm{row}_{\mathrm{b}}^{\sharp}"] & \\
\mathtt{img}(\mathrm{row}_{\mathrm{t}}^{\sharp})^A \arrow[r, "m^A"'] & (O^E)^A
\end{tikzcd} \tag{3}$$

It is easy to see that the hypothesis of this lemma corresponds to requiring the existence of a function close′ making the diagram below on the left in **Set** commute.

$$\begin{tikzcd}
S \arrow[d, dashed, "\mathrm{close}'"'] \arrow[dr, "\mathrm{row}_{\mathrm{b}}"] & \\
\mathtt{img}(\mathrm{row}_{\mathrm{t}}^{\sharp})^A \arrow[r, "m^A"'] & (O^E)^A
\end{tikzcd}
\qquad
\begin{tikzcd}
T(S) \arrow[d, "T(\mathrm{close}')"'] \arrow[dr, "T(\mathrm{row}_{\mathrm{b}})"] & \\
T(\mathtt{img}(\mathrm{row}_{\mathrm{t}}^{\sharp})^A) \arrow[r, "T(m^A)"'] \arrow[d] & T((O^E)^A) \arrow[d] \\
\mathtt{img}(\mathrm{row}_{\mathrm{t}}^{\sharp})^A \arrow[r, "m^A"'] & (O^E)^A
\end{tikzcd}$$

This diagram can be made into a diagram of T-algebra homomorphisms as on the right, where the compositions of the left and right legs give respectively $\mathrm{close}'^{\sharp}$ and $\mathrm{row}_{\mathrm{b}}^{\sharp}$. This diagram commutes

```
1   $S, E \leftarrow \{\varepsilon\}$
2   repeat
3       while the table is not closed or not consistent
4           if the table is not closed
5               find $s \in S$, $a \in A$ such that $\mathsf{row}_\mathsf{b}(s)(a) \neq \mathsf{row}_\mathsf{t}^\sharp(U)$ for all $U \in T(S)$
6               $S \leftarrow S \cup \{sa\}$
7           if the table is not consistent
8               find $U_1, U_2 \in T(S)$, $a \in A$, and $e \in E$ such that
                    $\mathsf{row}_\mathsf{t}^\sharp(U_1) = \mathsf{row}_\mathsf{t}^\sharp(U_2)$ and $\mathsf{row}_\mathsf{b}^\sharp(U_1)(a)(e) \neq \mathsf{row}_\mathsf{b}^\sharp(U_2)(a)(e)$
9               $E \leftarrow E \cup \{ae\}$
10      Construct the hypothesis $\mathcal{H}$ and submit it to the teacher
11      if the teacher replies no, with a counterexample $z$
12          $S \leftarrow S \cup \mathsf{prefixes}(z)$
13  until the teacher replies yes
14  return $\mathcal{H}$
```

Fig. 4. Adaptation of $\mathsf{L}^\star$ for T-automata.

because the top triangle commutes by functoriality of T, and the bottom square commutes by m^A being a T-algebra homomorphism. Therefore we have that (3) commutes for $\mathsf{close} = \mathsf{close}'^\sharp$. □

Example 4.3. For NFAs represented as $\mathcal{P}$-automata, the properties are as presented in Section 2.2. Recall that for $T = \mathcal{P}$ and $O = 2$, the Boolean join-semilattice, $\mathsf{row}_\mathsf{t}^\sharp$ and $\mathsf{row}_\mathsf{b}^\sharp$ describe a table where rows are labeled by subsets of S. Then we have, for instance, $\mathsf{row}_\mathsf{t}^\sharp(\{s_1, s_2\})(e) = \mathsf{row}_\mathsf{t}(s_1)(e) \vee \mathsf{row}_\mathsf{t}(s_2)(e)$, i.e., $\mathsf{row}_\mathsf{t}^\sharp(\{s_1, s_2\})(e) = 1$ if and only if $\mathcal{L}(s_1 e) = 1$ or $\mathcal{L}(s_2 e) = 1$. Closedness amounts to check whether each row in the bottom part of the table is the join of a set of rows in the top part. Consistency amounts to check whether, for all sets of rows $U_1, U_2 \subseteq S$ in the top part of the table whose joins are equal, the joins of rows $U_1 \cdot \{a\}$ and $U_2 \cdot \{a\}$ in the bottom part are also equal, for all $a \in A$.

If closedness and consistency hold, we can define a hypothesis T-automaton $\mathcal{H}$. Its state space is $H = \mathtt{img}(\mathsf{row}_\mathsf{t}^\sharp)$, $\mathsf{init} = \mathsf{row}_\mathsf{t}(\varepsilon)$, and output and transition maps are given by:

$$\mathsf{out} \colon H \to O \qquad\qquad \mathsf{out}(\mathsf{row}_\mathsf{t}^\sharp(U)) = \mathsf{row}_\mathsf{t}^\sharp(U)(\varepsilon)$$

$$\delta \colon H \to H^A \qquad\qquad \delta(\mathsf{row}_\mathsf{t}^\sharp(U)) = \mathsf{row}_\mathsf{b}^\sharp(U).$$

The correctness of this definition follows from the abstract treatment of van Heerdt et al. [2017], instantiated to the category of T-algebras and their homomorphisms.

We can now give our algorithm $\mathsf{L}_T^\star$. In the same way as for the example in Section 2, we only have to adjust lines 5 and 8 in Figure 1. The resulting algorithm is shown in Figure 4.

Correctness. Correctness for $\mathsf{L}_T^\star$ amounts to proving that, for any target language $\mathcal{L}$, the algorithm terminates returning the minimal T-automaton $\mathcal{M}_\mathcal{L}$ accepting $\mathcal{L}$. As in the original $\mathsf{L}^\star$ algorithm, we only need to prove that the algorithm terminates, that is, that only finitely many hypotheses are produced. Correctness follows from termination, since line 13 causes the algorithm to terminate only if the hypothesis automaton coincides with $\mathcal{M}_\mathcal{L}$.

In order to show termination, we argue that the state space H of the hypothesis increases while the algorithm loops, and that H cannot be larger than M, the state space of $\mathcal{M}_\mathcal{L}$. In fact, when a closedness defect is resolved (line 6), a row that was not previously found in the image of $\mathsf{row}_\mathsf{t}^\sharp \colon T(S) \to O^E$ is added, so the set H grows larger. When a consistency defect is resolved (line 9), two previously equal rows become distinguished, which also increases the size of H.

As for counterexamples, adding their prefixes to S (line 11) creates a consistency defect, which will be fixed during the next iteration, causing H to increase. This is due to the following result, which says that the counterexample z has a prefix that violates consistency.

Proposition 4.4. *If $z \in A^*$ is such that $\mathcal{L}_\mathcal{H}(z) \neq \mathcal{L}(z)$ and* $\mathsf{prefixes}(z) \subseteq S$*, then there are a prefix ua of z, with $u \in A^*$ and $a \in A$, and $U \in T(S)$ such that* $\mathsf{row}_\mathsf{t}(u) = \mathsf{row}_\mathsf{t}^\sharp(U)$ *and* $\mathsf{row}_\mathsf{b}(u)(a) \neq \mathsf{row}_\mathsf{b}^\sharp(U)(a)$.

Proof. Note that

$$\begin{aligned}
\mathsf{row}_\mathsf{t}(z)(\varepsilon) &= \mathcal{L}(z) && \text{(definition of } \mathsf{row}_\mathsf{t}\text{)}\\
&\neq \mathcal{L}_\mathcal{H}(z) && \text{(assumption)}\\
&= \mathsf{out}_\mathcal{H}(r_\mathcal{H}(z)) && \text{(Definition of } \mathcal{L}_\mathcal{H}\text{)}\\
&= r_\mathcal{H}(z)(\varepsilon) && \text{(definition of } \mathsf{out}_\mathcal{H}\text{),}
\end{aligned}$$

so $\mathsf{row}_\mathsf{t}(z) \neq r_\mathcal{H}(z)$. Let $p \in A^*$ be the smallest prefix of z satisfying $\mathsf{row}_\mathsf{t}(p) \neq r_\mathcal{H}(p)$. We have $\mathsf{row}_\mathsf{t}(\varepsilon) = \mathsf{init}_\mathcal{H} = r_\mathcal{H}(\varepsilon)$, so $p \neq \varepsilon$ and therefore $p = ua$ for certain $u \in A^*$ and $a \in A$. Let $S' \subset S$ be the set from which $\mathcal{H}$ was constructed—recall that we added $\mathsf{prefixes}(z)$ to S after constructing $\mathcal{H}$. Choose any $U \in T(S')$ such that $\mathsf{row}_\mathsf{t}^\sharp(U) = r_\mathcal{H}(u)$, which is possible because H is the image of $\mathsf{row}_\mathsf{t}^\sharp$ restricted to the domain $T(S')$. By the minimality property of p we have $\mathsf{row}_\mathsf{t}(u) = r_\mathcal{H}(u) = \mathsf{row}_\mathsf{t}^\sharp(U)$. Furthermore,

$$\begin{aligned}
\mathsf{row}_\mathsf{b}(u)(a) &= \mathsf{row}_\mathsf{t}(ua) && \text{(definitions of } \mathsf{row}_\mathsf{t} \text{ and } \mathsf{row}_\mathsf{b}\text{)}\\
&\neq r_\mathcal{H}(ua) && (ua = p \text{ and } \mathsf{row}_\mathsf{t}(p) \neq r_\mathcal{H}(p))\\
&= \delta_\mathcal{H}(r_\mathcal{H}(u))(a) && \text{(definition of } r_\mathcal{H}\text{)}\\
&= \delta_\mathcal{H}(\mathsf{row}_\mathsf{t}^\sharp(U))(a) && (r_\mathcal{H}(u) = \mathsf{row}_\mathsf{t}^\sharp(U))\\
&= \mathsf{row}_\mathsf{b}^\sharp(U)(a) && \text{(definition of } \delta_\mathcal{H}\text{).} \qquad \square
\end{aligned}$$

Now, note that, by increasing S or E, the hypothesis state space H never decreases in size. Moreover, for $S = A^*$ and $E = A^*$, $\mathsf{row}_\mathsf{t}^\sharp = t_\mathcal{L}^\sharp$, as defined in Definition 6.2. Therefore, since H and M are defined as the images of $\mathsf{row}_\mathsf{t}^\sharp$ and $t_\mathcal{L}^\sharp$, respectively, the size of H is bounded by that of M. Since H increases while the algorithm loops, the algorithm must terminate and thus is correct.

We note that the RFSA learning algorithm of Bollig et al. does not terminate using this counterexample processing method [Bollig et al. 2008, Appendix F]. This is due to their notion of consistency being weaker than ours: we have shown that progress is guaranteed because a consistency defect, in our sense, is created using this method.

Query complexity. The complexity of automata learning algorithms is usually measured in terms of the number of both membership and equivalence queries asked, as it is common to assume that computations within the algorithm are insignificant compared to evaluating the system under analysis in real-world applications. The complexity of answering the queries themselves is not considered, as it depends on the implementation of the teacher, which the algorithm abstracts from.

Notice that, as the table is a T-algebra homomorphism, asking membership queries for rows labeled by words in S is enough to determine all other rows, for which queries need not be asked. We measure the query complexities in terms of the number of states n of the minimal Moore automaton, the number of states t of the minimal T-automaton, the size k of the alphabet, and the length m of the longest counterexample. Note that t cannot be smaller than n, but it can be much bigger. For example, when $T = \mathcal{P}$, t may be in $\mathcal{O}(2^n)$.[2]

The maximum number of closedness defects fixed by the algorithm is n, as a closedness defect for the setting with algebraic structure is also a closedness defect for the setting without that structure. The maximum number of consistency defects fixed by the algorithm is t, as fixing a consistency defect distinguishes two rows that were previously identified. Since counterexamples lead to consistency defects, this also means that the algorithm will not pose more than t equivalence queries. A word is added to S when fixing a closedness defect, and $\mathcal{O}(m)$ words are added to S when processing a counterexample. The number of rows that we need to fill using queries is therefore in $\mathcal{O}(tmk)$. The number of columns added to the table is given by the number of times a consistency defect is fixed and thus in $\mathcal{O}(t)$. Altogether, the number of membership queries is in $\mathcal{O}(t^2mk)$.

5 SUCCINCT HYPOTHESES

We now describe the first of two optimizations, which is enabled by the use of monads. Our algorithm produces hypotheses that can be quite large, as their state space is the image of $\mathsf{row}_\mathsf{t}^\sharp$, which has the whole set $T(S)$ as its domain. For instance, when $T = \mathcal{P}$, $T(S)$ is exponentially larger than S. We show how we can compute *succinct* hypotheses, whose state space is given by a subset of S. We start by defining sets of *generators for the table*.

Definition 5.1. A set $S' \subseteq S$ is a *set of generators for the table* whenever for all $s \in S$ there is $U \in T(S')$ such that $\mathsf{row}_\mathsf{t}(s) = \mathsf{row}_\mathsf{t}^\sharp(U)$.[3]

Intuitively, U is the decomposition of s into a “combination” of generators. When $T = \mathcal{P}$, S' generates the table whenever each row can be obtained as the join of a set of rows labeled by S'. Explicitly: for all $s \in S$ there is $\{s_1, \ldots, s_n\} \subseteq S'$ such that $\mathsf{row}_\mathsf{t}(s) = \mathsf{row}_\mathsf{t}^\sharp(\{s_1, \ldots, s_n\}) = \mathsf{row}_\mathsf{t}(s_1) \vee \cdots \vee \mathsf{row}_\mathsf{t}(s_n)$.

Recall that $\mathcal{H}$, with state space H, is the hypothesis automaton for the table. The existence of generators S' allows us to compute a T-automaton with state space $T(S')$ equivalent to $\mathcal{H}$. We call this the *succinct hypothesis*, although $T(S')$ may be larger than H. Proposition 3.7 tells us that the succinct hypothesis can be represented as an automaton with side-effects in T that has S' as its state space. This results in a lower space complexity when storing the hypothesis.

We now show how the succinct hypothesis is computed. Observe that, if generators S' exist, $\mathsf{row}_\mathsf{t}^\sharp$ factors through the restriction of itself to $T(S')$. Denote this latter function $\widehat{\mathsf{row}_\mathsf{t}}^\sharp$. Since we have $T(S') \subseteq T(S)$, the image of $\widehat{\mathsf{row}_\mathsf{t}}^\sharp$ coincides with $\mathtt{img}(\mathsf{row}_\mathsf{t}^\sharp) = H$, and therefore the surjection restricting $\widehat{\mathsf{row}_\mathsf{t}}^\sharp$ to its image has the form $e \colon T(S') \to H$. Any right inverse $i \colon H \to T(S')$ of the function e (that is, $e \circ i = \mathrm{id}_H$, but whereas e is a T-algebra homomorphism, i need not be one) yields a succinct hypothesis as follows.

Definition 5.2 (Succinct Hypothesis). The *succinct hypothesis* is the following T-automaton $\mathcal{S}$: its state space is $T(S')$, its initial state is $\mathsf{init} = i(\mathsf{row}_\mathsf{t}(\varepsilon))$, and we define

$$\mathsf{out}^\dagger \colon S' \to O \qquad \mathsf{out}^\dagger(s) = \mathsf{row}_\mathsf{t}(s)(\varepsilon)$$
$$\delta^\dagger \colon S' \to T(S')^A \qquad \delta^\dagger(s)(a) = i(\mathsf{row}_\mathsf{b}(s)(a)).$$

[2] This can be seen from the language $\{a^p\}$, for some $p \in \mathbb{N}$ and a singleton alphabet $\{a\}$. Its residual languages are $\emptyset$ and $\{a^i\}$ for all $0 \leq i \leq p$, which means the minimal DFA accepting the language has $p + 2$ states. However, the residual languages w.r.t. sets of words are all the subsets of $\{\varepsilon, a, aa, \ldots, a^p\}$—hence, the minimal T-automaton has 2^{p+1} states.

[3] Here and hereafter we assume that $T(S') \subseteq T(S)$, and more generally that T preserves inclusion maps. To eliminate this assumption, one could take the inclusion map $f \colon S' \hookrightarrow S$ and write $\mathsf{row}_\mathsf{t}^\sharp(T(f)(U))$ instead of $\mathsf{row}_\mathsf{t}^\sharp(U)$.

This definition is inspired by that of a *scoop*, due to Arbib and Manes [1975].

Proposition 5.3. *Any succinct hypothesis of $\mathcal{H}$ accepts the same language as $\mathcal{H}$.*

The proof can be found in the appendix. We now give a simple procedure to compute a *minimal* set of generators, that is, a set S' such that no proper subset is a set of generators. This generalizes a procedure defined by Angluin et al. [2015] for non-deterministic, universal, and alternating automata.

Proposition 5.4. *The following algorithm returns a minimal set of generators for the table:*

```
$S' \leftarrow S$
while there are $s \in S'$ and $U \in T(S' \setminus \{s\})$ s.t. $\mathsf{row}_t^\sharp(U) = \mathsf{row}_t(s)$
    $S' \leftarrow S' \setminus \{s\}$
return $S'$
```

The proof can be found in the appendix. Determining whether U as in the algorithm given in Proposition 5.4 exists, one can always naively enumerate all possibilities, using that T preserves finite sets. This is what we call the basic algorithm. For specific algebraic structures, one may find more efficient methods, as we show in the following example.

Example 5.5. Consider again the powerset monad $T = \mathcal{P}$. We now exemplify two ways of computing succinct hypotheses, which are inspired by the definitions of canonical RFSAs [Denis et al. 2002]. The basic idea is to start from a deterministic automaton and to remove states that are equivalent to a set of other states. The algorithm given in Proposition 5.4 computes a minimal S' that only contains labels of rows that are not the join of other rows. (In case two rows are equal, only one of their labels is kept.) In other words, as mentioned in Section 2, S' contains labels of join-irreducible rows. To concretize the algorithm efficiently, we use a method introduced by Bollig et al. [2009], which essentially exploits the natural order on the JSL of table rows. In contrast to the basic exponential algorithm, this results in a polynomial one.[4] Bollig et al. determine whether a row is a join of other rows by comparing the row just to the join of rows below it. Like them, we make use of this also to compute right inverses of e, for which we will formalize the order.

The function $e\colon \mathcal{P}(S') \to H$ tells us which sets of rows are equivalent to a single state in H. We show two right inverses $H \to \mathcal{P}(S')$ for it. The first one,

$$i_1(h) = \{s \in S' \mid \mathsf{row}_t(s) \le h\},$$

stems from the construction of the *canonical RFSA* of a language [Denis et al. 2002]. Here we use the order $a \le b \iff a \vee b = b$ induced by the JSL structure. The resulting construction of a succinct hypothesis was first used by Bollig et al. [2009]. This succinct hypothesis has a "maximal" transition function, meaning that no more transitions can be added without changing the language of the automaton.

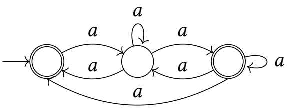
The second inverse is

$$i_2(h) = \{s \in S' \mid \mathsf{row}_t(s) \le h \text{ and for all } s' \in S' \text{ s.t. } \mathsf{row}_t(s) \le \mathsf{row}_t(s') \le h \\ \text{we have } \mathsf{row}_t(s) = \mathsf{row}_t(s')\},$$

resulting in a more economical transition function, where some redundancies are removed. This corresponds to the *simplified canonical RFSA* Denis et al. [2002].

[4] When we refer to computational complexities, as opposed to query complexities, they are in terms of the sizes of S, E, and A.

Example 5.6. Consider again the powerset monad $T = \mathcal{P}$, and recall the table in Figure 3e. When $S' = S$, the right inverse given by i_1 yields the succinct hypothesis shown below.



Note that $i_1(\mathrm{row}_t(aa)) = \{\varepsilon, a, aa\}$. When instead taking i_2, the succinct hypothesis is just the DFA (1) because $i_2(\mathrm{row}_t(aa)) = \{aa\}$. Rather than constructing a succinct hypothesis directly, our algorithm first reduces the set S'. In this case, we note that $\mathrm{row}_t(aa) = \mathrm{row}_t^\sharp(\{\varepsilon, a\})$, so we can remove aa from S'. Now i_1 and i_2 coincide and produce the NFA (2). Minimizing the set S' in this setting essentially comes down to determining what Bollig et al. [2009] call the *prime* rows of the table.

Remark 5.7. The algorithm in Proposition 5.4 implicitly assumes an order in which elements of S are checked. Although the algorithm is correct for any such order, different orders may give results that differ in size.

6 OPTIMIZED COUNTEREXAMPLE HANDLING

The second optimization we give generalizes the counterexample processing method due to Rivest and Schapire [1993], which improves the worst case complexity of the number of membership queries needed in L*. Maler and Pnueli [1995] proposed to add all suffixes of the counterexample to the set E instead of adding all prefixes to the set S. This eliminates the need for consistency checks in the deterministic setting. The method by Rivest and Schapire finds a *single* suffix of the counterexample and adds it to E. This suffix is chosen in such a way that it either distinguishes two existing rows or creates a closedness defect, both of which imply that the hypothesis automaton will grow.

The main idea is finding the distinguishing suffix via the hypothesis automaton $\mathcal{H}$. Given a word $u \in A^*$, let q_u be the state in $\mathcal{H}$ reached by reading u, i.e., $q_u = r_{\mathcal{H}}(u)$. For each $q \in H$, we pick any $U_q \in T(S)$ that yields q according to the table, i.e., such that $\mathrm{row}_t^\sharp(U_q) = q$. Then for a counterexample z we have that the residual language w.r.t. U_{q_z} does not "agree" with the residual language w.r.t. z.

The above intuition can be formalized as follows. Let $\mathcal{R}\colon A^* \to O^{A^*}$ be given by $\mathcal{R}(u) = t_{\mathcal{L}}^\sharp(U_{q_u})$ for all $u \in A^*$, the residual language computation. We have the following technical lemma, saying that a counterexample z distinguishes the residual languages $t_{\mathcal{L}}(z)$ and $\mathcal{R}(z)$.

Lemma 6.1. *If $z \in A^*$ is such that $\mathcal{L}_{\mathcal{H}}(z) \neq \mathcal{L}(z)$, then $t_{\mathcal{L}}(z)(\varepsilon) \neq \mathcal{R}(z)(\varepsilon)$.*

Proof. We have

$$\begin{aligned}
t_{\mathcal{L}}(z)(\varepsilon) &= \mathcal{L}(z) && \text{(definition of } t_{\mathcal{L}}\text{)}\\
&\neq \mathcal{L}_{\mathcal{H}}(z) && \text{(assumption)}\\
&= (\mathrm{out}_{\mathcal{H}} \circ r_{\mathcal{H}})(z) && \text{(definition of } \mathcal{L}_{\mathcal{H}}\text{)}\\
&= r_{\mathcal{H}}(z)(\varepsilon) && \text{(definition of } \mathrm{out}_{\mathcal{H}}\text{)}\\
&= q_z(\varepsilon) && \text{(definition of } q_z\text{)}\\
&= \mathrm{row}_t^\sharp(U_{q_z})(\varepsilon) && \text{(definition of } U_{q_z}\text{)}\\
&= t_{\mathcal{L}}^\sharp(U_{q_z})(\varepsilon) && \text{(definitions of } \mathrm{row}_t \text{ and } t_{\mathcal{L}}\text{)}\\
&= \mathcal{R}(z)(\varepsilon) && \text{(definition of } \mathcal{R}\text{).}
\end{aligned}$$

□

We assume that $U_{q_\varepsilon} = \eta(\varepsilon)$. For a counterexample z, we then have $\mathcal{R}(\varepsilon)(z) = t_{\mathcal{L}}(\varepsilon)(z) \neq \mathcal{R}(z)(\varepsilon)$. While reading z, the hypothesis automaton passes a sequence of states $q_{u_0}, q_{u_1},q_{u_2},\ldots,q_{u_n}$, where $u_0 = \epsilon$, $u_n = z$, and $u_{i+1} = u_i a$ for some $a \in A$ is a prefix of z. If z were correctly classified by $\mathcal{H}$, all residuals $\mathcal{R}(u_i)$ would classify the remaining suffix v of z, i.e., such that $z = u_i v$, in the same way. However, the previous lemma tells us that, for a counterexample z, this is not case, meaning that for some suffix v we have $\mathcal{R}(ua)(v) \neq \mathcal{R}(u)(av)$. In short, this inequality is discovered along a transition in the path to z.

Corollary 6.2. *If $z \in A^*$ is such that $\mathcal{L}_{\mathcal{H}}(z) \neq \mathcal{L}(z)$, then there are $u, v \in A^*$ and $a \in A$ such that $uav = z$ and $\mathcal{R}(ua)(v) \neq \mathcal{R}(u)(av)$.*

To find such a decomposition efficiently, Rivest and Schapire use a binary search algorithm. We conclude with the following result that turns the above property into the elimination of a closedness witness. That is, given a counterexample z and the resulting decomposition uav from the above corollary, we show that, while currently $\mathsf{row}_\mathsf{t}^\sharp(U_{q_{ua}}) = \mathsf{row}_\mathsf{b}^\sharp(U_{q_u})(a)$, after adding v to E we have $\mathsf{row}_\mathsf{t}^\sharp(U_{q_{ua}})(v) \neq \mathsf{row}_\mathsf{b}^\sharp(U_{q_u})(a)(v)$. (To see that the latter follows from the proposition below, note that for all $U \in T(S)$ and $e \in E$, $\mathsf{row}_\mathsf{t}^\sharp(U)(e) = t_{\mathcal{L}}^\sharp(U)(e)$ and for each $a' \in A$, $\mathsf{row}_\mathsf{b}^\sharp(U)(a')(e) = t_{\mathcal{L}}^\sharp(U)(a'e)$, by the definition of those maps.) The inequality means that either we have a closedness defect, or there still exists some $U \in T(S)$ such that $\mathsf{row}_\mathsf{t}^\sharp(U) = \mathsf{row}_\mathsf{b}^\sharp(U_{q_u})(a)$. In this case, the rows $\mathsf{row}_\mathsf{t}^\sharp(U)$ and $\mathsf{row}_\mathsf{t}^\sharp(U_{q_{ua}})$ have become distinguished by adding v, which means that the size of H has been increased. We know that a closedness defect leads to an increase in the size of H, so in any case we make progress.

Proposition 6.3. *If $z \in A^*$ is such that $\mathcal{L}_{\mathcal{H}}(z) \neq \mathcal{L}(z)$, then there are $u, v \in A^*$ and $a \in A$ such that $\mathsf{row}_\mathsf{t}^\sharp(U_{q_{ua}}) = \mathsf{row}_\mathsf{b}^\sharp(U_{q_u})(a)$ and $t_{\mathcal{L}}^\sharp(U_{q_{ua}})(v) \neq t_{\mathcal{L}}^\sharp(U_{q_u})(av)$.*

Proof. By Corollary 6.2 we have $u, v \in A^*$ and $a \in A$ such that $\mathcal{R}(ua)(v) \neq \mathcal{R}(u)(av)$. This directly yields the inequality by the definition of $\mathcal{R}$. Furthermore,

$$\begin{aligned}
\mathsf{row}_\mathsf{t}(U_{q_{ua}}) &= q_{ua} && \text{(definition of } U_{q_{ua}}\text{)}\\
&= r_{\mathcal{H}}(ua) && \text{(definition of } q_{ua}\text{)}\\
&= \delta_{\mathcal{H}}(r_{\mathcal{H}}(u))(a) && \text{(definition of } r_{\mathcal{H}}\text{)}\\
&= \delta_{\mathcal{H}}(q_u)(a) && \text{(definition of } q_u\text{)}\\
&= \delta_{\mathcal{H}}(\mathsf{row}_\mathsf{t}^\sharp(U_{q_u}))(a) && \text{(definition of } U_{q_u}\text{)}\\
&= \mathsf{row}_\mathsf{b}^\sharp(U_{q_u})(a) && \text{(definition of } \delta_{\mathcal{H}}\text{).}
\end{aligned}$$

□

We now show how to combine this optimized counterexample processing method with the succinct hypothesis optimization from Section 5. Recall that the succinct hypothesis $\mathcal{S}$ is based on a right inverse $i\colon H \to T(S')$ of $e\colon T(S') \to H$. Choosing such an i is equivalent to choosing U_q for each $q \in H$. We then redefine $\mathcal{R}$ using the reachability map of the succinct hypothesis. Specifically, $\mathcal{R}(u) = t_{\mathcal{L}}^\sharp(r_{\mathcal{S}}(u))$ for all $u \in A^*$.

Unfortunately, there is one complication. We assumed earlier that $U_{q_\varepsilon} = \eta(\varepsilon)$, or more specifically $\mathcal{R}(\varepsilon)(z) = \mathcal{L}(z)$. This now may be impossible because we do not even necessarily have $\varepsilon \in S'$. We show next that if this equality does not hold, then there are two rows that we can distinguish by adding z to E. Thus, after testing whether $\mathcal{R}(\varepsilon)(z) = \mathcal{L}(z)$, we either add z to E (if the test fails) or proceed with the original method.

Proposition 6.4. *If $z \in A^*$ is such that $\mathcal{R}(\varepsilon)(z) \neq \mathcal{L}(z)$, then $\mathrm{row}_{\mathrm{t}}^{\sharp}(\mathrm{init}_{\mathcal{S}}) = \mathrm{row}_{\mathrm{t}}(\varepsilon)$ and $t_{\mathcal{L}}^{\sharp}(\mathrm{init}_{\mathcal{S}})(z) \neq t_{\mathcal{L}}(\varepsilon)(z)$.*

Proof. We have $\mathrm{row}_{\mathrm{t}}^{\sharp}(\mathrm{init}_{\mathcal{S}}) = \mathrm{row}_{\mathrm{t}}^{\sharp}(i(\mathrm{row}_{\mathrm{t}}(\varepsilon))) = \mathrm{row}_{\mathrm{t}}(\varepsilon)$ by the definitions of $\mathrm{init}_{\mathcal{S}}$ and i, and

$$\begin{aligned}
t_{\mathcal{L}}^{\sharp}(i(\mathrm{row}_{\mathrm{t}}(\varepsilon)))(z) &= t_{\mathcal{L}}^{\sharp}(\mathrm{init}_{\mathcal{S}})(z) && \text{(definition of } \mathrm{init}_{\mathcal{S}}\text{)} \\
&= t_{\mathcal{L}}^{\sharp}(r_{\mathcal{S}}(\varepsilon))(z) && \text{(definition of } r_{\mathcal{S}}\text{)} \\
&= \mathcal{R}(\varepsilon)(z) && \text{(definition of } \mathcal{R}\text{)} \\
&\neq \mathcal{L}(z) && \text{(assumption)} \\
&= t_{\mathcal{L}}(\varepsilon)(z) && \text{(definition of } t_{\mathcal{L}}\text{).} \qquad \square
\end{aligned}$$

To see that the original method still works, we prove the analogue of Lemma 6.1 for the new definition of $\mathcal{R}$.

Lemma 6.5. *If $z \in A^*$ is such that $\mathcal{L}_{\mathcal{S}}(z) \neq \mathcal{L}(z)$ and $\mathcal{R}(\varepsilon)(z) = \mathcal{L}(z)$, then $\mathcal{R}(\varepsilon)(z) \neq \mathcal{R}(z)(\varepsilon)$.*

Proof. We have

$$\begin{aligned}
\mathcal{R}(\varepsilon)(z) &= \mathcal{L}(z) && \text{(assumption)} \\
&\neq \mathcal{L}_{\mathcal{S}}(z) && \text{(counterexample)} \\
&= (\mathrm{out}_{\mathcal{S}} \circ r_{\mathcal{S}}^{\dagger})(z) && \text{(definition of } \mathcal{L}_{\mathcal{S}}\text{)} \\
&= (\mathrm{row}_{\mathrm{t}}^{\sharp} \circ r_{\mathcal{S}}^{\dagger})(z)(\varepsilon) && \text{(definition of } \mathrm{out}_{\mathcal{S}}\text{)} \\
&= t_{\mathcal{L}}^{\sharp}(r_{\mathcal{S}}^{\dagger}(z))(\varepsilon) && \text{(definition of } \mathrm{row}_{\mathrm{t}}^{\sharp}\text{)} \\
&= \mathcal{R}(z)(\varepsilon) && \text{(definition of } \mathcal{R}\text{).} \qquad \square
\end{aligned}$$

Corollary 6.6. *If $z \in A^*$ is such that $\mathcal{L}_{\mathcal{S}}(z) \neq \mathcal{L}(z)$ and $\mathcal{R}(\varepsilon)(z) = \mathcal{L}(z)$, then there are $u, v \in A^*$ and $a \in A$ such that $uav = z$ and $\mathcal{R}(ua)(v) \neq \mathcal{R}(u)(av)$.*

Now we are ready to prove the analogue of Proposition 6.3.

Proposition 6.7. *If $z \in A^*$ is such that $\mathcal{L}_{\mathcal{S}}(z) \neq \mathcal{L}(z)$ and $\mathcal{R}(\varepsilon)(z) = \mathcal{L}(z)$, then there are $u, v \in A^*$ and $a \in A$ such that $\mathrm{row}_{\mathrm{t}}^{\sharp}(r_{\mathcal{S}}^{\dagger}(ua)) = \mathrm{row}_{\mathrm{b}}^{\sharp}(r_{\mathcal{S}}^{\dagger}(u))(a)$ and $t_{\mathcal{L}}^{\sharp}(r_{\mathcal{S}}^{\dagger}(ua))(v) \neq t_{\mathcal{L}}^{\sharp}(r_{\mathcal{S}}^{\dagger}(u))(av)$.*

Proof. Let u, a, and v be as in Corollary 6.6. Thus,

$$t_{\mathcal{L}}^{\sharp}(r_{\mathcal{S}}^{\dagger}(ua))(v) = \mathcal{R}(ua)(v) \neq \mathcal{R}(u)(av) = t_{\mathcal{L}}^{\sharp}(r_{\mathcal{S}}^{\dagger}(u))(av).$$

Furthermore, since for all $s \in S$ and $b \in A$ we have

$$\begin{aligned}
((\mathrm{row}_{\mathrm{t}}^{\sharp})^{A} \circ \delta_{\mathcal{S}}^{\dagger})(s)(b) &= \mathrm{row}_{\mathrm{t}}^{\sharp}(\delta_{\mathcal{S}}^{\dagger}(s)(b)) \\
&= (\mathrm{row}_{\mathrm{t}}^{\sharp} \circ i)(\mathrm{row}_{\mathrm{b}}(s)(b)) && \text{(definition of } \delta_{\mathcal{S}}^{\dagger}\text{)} \\
&= \mathrm{row}_{\mathrm{b}}(s)(b) && \text{(definition of } i\text{),}
\end{aligned}$$

it follows that $(\mathrm{row}_{\mathrm{t}}^{\sharp})^{A} \circ \delta_{\mathcal{S}} = \mathrm{row}_{\mathrm{b}}^{\sharp}$. Therefore,

$$\begin{aligned}
\mathrm{row}_{\mathrm{t}}^{\sharp}(r_{\mathcal{S}}^{\dagger}(ua)) &= \mathrm{row}_{\mathrm{t}}^{\sharp}(\delta_{\mathcal{S}}(r_{\mathcal{S}}^{\dagger}(u))(a)) && \text{(definition of } r_{\mathcal{S}}^{\dagger}\text{)} \\
&= ((\mathrm{row}_{\mathrm{t}}^{\sharp})^{A} \circ \delta_{\mathcal{S}})(r_{\mathcal{S}}^{\dagger}(u))(a) \\
&= \mathrm{row}_{\mathrm{b}}^{\sharp}(r_{\mathcal{S}}^{\dagger}(u))(a). && \qquad \square
\end{aligned}$$

Example 6.8. Recall the succinct hypothesis $\mathcal{S}$ from Figure 3c for the table in Figure 2a. Note that $S' = S$ cannot be further reduced. The hypothesis is based on the right inverse $i \colon H \to \mathcal{P}(S)$ of $e \colon \mathcal{P}(S) \to H$ given by $i(\mathsf{row}_\mathsf{t}(\varepsilon)) = \{\varepsilon\}$ and $i(\mathsf{row}_\mathsf{t}^\sharp(\emptyset)) = \emptyset$. This is the only possible right inverse because e is bijective. For the prefixes of the counterexample aa we have $r_\mathcal{S}(\varepsilon) = \{\varepsilon\}$ and $r_\mathcal{S}(a) = r_\mathcal{S}(aa) = \emptyset$. Note that $t_\mathcal{L}^\sharp(\{\varepsilon\})(aa) = 1$ while $t_\mathcal{L}(\emptyset)(a) = t_\mathcal{L}(\emptyset)(\varepsilon) = 0$. Thus, $\mathcal{R}(\varepsilon)(aa) \neq \mathcal{R}(a)(a)$. Adding a to E would indeed create a closedness defect.

Query complexity. Again, we measure the membership and equivalence query complexities in terms of the number of states n of the minimal Moore automaton, the number of states t of the minimal T-automaton, the size k of the alphabet, and the length m of the longest counterexample.

A counterexample now gives an additional column instead of a set of rows, and we have seen that this leads to either a closedness defect or to two rows being distinguished. Thus, the number of equivalence queries is still at most t, and the number of columns is still in $\mathcal{O}(t)$. However, the number of rows that we need to fill using membership queries is now in $\mathcal{O}(nk)$. This means that a total of $\mathcal{O}(tnk)$ membership queries is needed to fill the table.

Apart from filling the table, we also need queries to analyze counterexamples. The binary search algorithm mentioned after Corollary 6.2 requires for each counterexample $\mathcal{O}(\log m)$ computations of $\mathcal{R}(x)(y)$ for varying words x and y. Let r be the maximum number of queries required for a single such computation. Note that for $u, v \in A^*$, and letting $\alpha \colon TO \to O$ be the algebra structure on O, we have

$$\mathcal{R}(u)(v) = \alpha(T(\mathsf{ev}_v \circ t_\mathcal{L})(U_{q_u}))$$

for the original definition of $\mathcal{R}$ and

$$\mathcal{R}(u)(v) = \alpha(T(\mathsf{ev}_v \circ t_\mathcal{L})(r_\mathcal{S}^\dagger(u)))$$

in the succinct hypothesis case. Since the restricted map $T(\mathsf{ev}_v \circ t_\mathcal{L}) \colon TS \to TO$ is completely determined by $\mathsf{ev}_v \circ t_\mathcal{L} : S \to O$, r is at most $|S|$, which is bounded by n in this optimized algorithm. For some examples (see for instance the writer automata in Section 7), we even have $r = 1$. The overall membership query complexity is $\mathcal{O}(tnk + tr \log m)$.

Dropping Consistency. We described the counterexample processing method based around Proposition 6.3 in terms of the succinct hypothesis $\mathcal{S}$ rather than the actual hypothesis $\mathcal{H}$ by showing that $\mathcal{R}$ can be defined using $\mathcal{S}$. Since the definition of the succinct hypothesis does not rely on the property of consistency to be well-defined, this means we could drop the consistency check from the algorithm altogether. We can still measure progress in terms of the size of the set H, but it will not be the state space of an actual hypothesis during intermediate stages. This observation also explains why Bollig et al. [2009] are able to use a weaker notion of consistency in their algorithm. Interestingly, they exploit the canonicity of their choice of succinct hypotheses to arrive at a polynomial membership query complexity that does not involve the factor t.

7 EXAMPLES

In this section we list several examples that can be seen as T-automata and hence learned via an instance of $\mathsf{L}_T^\star$. We remark that, since our algorithm operates on finite structures (recall that T preserves finite sets), for each automaton type one can obtain a basic, correct-by-construction instance of $\mathsf{L}_T^\star$ for free, by just plugging the concrete definition of the monad into the abstract algorithm. However, we note that this is not how $\mathsf{L}_T^\star$ is intended to be used in a real-world context. Instead, it should be seen as an abstract specification of the operations each concrete implementation needs to perform, or, in other words, as a template for real implementations.

For each instance below, we discuss whether certain operations admit a more efficient implementation than the basic one, based on the specific algebraic structure induced by the monad. We also mention related algorithms from the literature. Due to our general treatment, the optimizations of Sections 5 and 6 apply to all of these instances.

Non-deterministic automata. As discussed before, non-deterministic automata are $\mathcal{P}$-automata with a free state space, provided that $O = 2$ is equipped with the "or" operation as its $\mathcal{P}$-algebra structure. We also mentioned that, as Bollig et al. [2009] showed, there is a polynomial time algorithm to check whether a given row is the join of other rows. This gives an efficient method for handling closedness straight away. Moreover, as shown in Example 5.5, it allows for an efficient construction of the succinct hypothesis. Unfortunately, checking for consistency defects seems to require a number of computations exponential in the number of rows. We recall that Bollig et al. [2009] use an ad-hoc version of consistency which cannot be easily captured in our framework. However, as explained at the end of Section 6, we can in fact drop consistency altogether.

Universal automata. Just like non-deterministic automata, universal automata can be seen as $\mathcal{P}$-automata with a free state space. The difference, however, is that the $\mathcal{P}$-algebra structure on $O = 2$ is dual: it is given by the "and" rather than the "or" operation. Universal automata accept a word when all paths reading that word are accepting. One can dualize the optimized specific algorithms for the case of non-deterministic automata. This is precisely what Angluin et al. [2015] have done.

Partial automata. Consider the *maybe monad* Maybe, given by $\mathsf{Maybe}(X) = 1 + X$, with natural transformations having components $\eta_X : X \longrightarrow 1 + X$ and $\mu_X : 1 + 1 + X \longrightarrow 1 + X$ defined in the standard way. Partial automata with states X can be represented as Maybe-automata with state space $\mathsf{Maybe}(X) = 1 + X$, where there is an additional *sink state*, and output algebra $O = \mathsf{Maybe}(1) = 1 + 1$. Here the left value is for rejecting states, including the sink one. The transition map $\delta : 1 + X \longrightarrow (1 + X)^A$ represents an undefined transition as one going to the sink state. The algorithm $\mathsf{L}^{\star}_{\mathsf{Maybe}}$ is mostly like $\mathsf{L}^{\star}$, except that implicitly the table has an additional row with zeroes in every column. Since the monad only adds a single element to each set, there is no need to optimize the basic algorithm for this specific case.

Weighted automata. Recall from Section 3 the *free semimodule monad* V, sending a set X to the free semimodule over a finite semiring $\mathbb{S}$. Weighted automata over a set of states X can be represented as V-automata whose state space is the semimodule $V(X)$, the output function $\mathsf{out} : V(X) \longrightarrow \mathbb{S}$ assigns a weight to each state, and the transition map $\delta : V(X) \longrightarrow V(X)^A$ sends each state and each input symbol to a linear combination of states. The obvious semimodule structure on $\mathbb{S}$ extends to a pointwise structure on the potential rows of the table. The basic algorithm loops over all linear combinations of rows to check closedness and over all pairs of combinations of rows to check consistency. This is an extremely expensive operation. If $\mathbb{S}$ is a field, a row can be decomposed into a linear combination of other rows in polynomial time using standard techniques from linear algebra. As a result, there are efficient procedures for checking closedness and constructing succinct hypotheses. It was shown by van Heerdt et al. [2017] that consistency in this setting is equivalent to closedness of the transpose of the table. This trick is due to Bergadano and Varricchio [1996], who first studied learning of weighted automata.

Alternating automata. We use the characterization of alternating automata due to Bertrand [2017]. Recall that, given a partially ordered set $(P, \leq)$, an *upset* is a subset U of P such that, if $x \in U$ and $x \leq y$, then $y \in U$. Given $Q \subseteq P$, we write $\uparrow\! Q$ for the *upward closure* of Q, that is the smallest upset of P containing Q. We consider the monad A that maps a set X to the set of all upsets of $\mathcal{P}(X)$.

Its unit is given by $\eta_X(x) = \uparrow\{\{x\}\}$ and its multiplication by

$$\mu_X(U) = \{V \subseteq X \mid \exists_{W \in U} \forall_{Y \in W} \exists_{Z \in Y} Z \subseteq V\}.$$

The sets of sets in $\mathsf{A}(X)$ can be seen as DNF formulae over elements of X, where the outer powerset is disjunctive and the inner one is conjunctive. Accordingly, we define an algebra structure $\beta \colon \mathsf{A}(2) \to 2$ on the output set 2 by letting $\beta(U) = 1$ if $\{1\} \in U$, 0 otherwise. Alternating automata with states X can be represented as A-automata with state space $\mathsf{A}(X)$, output map $\mathrm{out} \colon \mathsf{A}(X) \to 2$, and transition map $\delta \colon \mathsf{A}(X) \to \mathsf{A}(X)^A$, sending each state to a DNF formula over X. The only difference with the usual definition of alternating automata is that $\mathsf{A}(X)$ is not the full set $\mathcal{PP}(X)$, which would not give a monad in the desired way. However, for each formula in $\mathcal{PP}(X)$ there is an equivalent one in $\mathsf{A}(X)$.

An adaptation of $\mathsf{L}^\star$ for alternating automata was introduced by Angluin et al. [2015] and further investigated by Berndt et al. [2017]. The former found that given a row $r \in 2^E$ and a set of rows $X \subseteq 2^E$, r is equal to a DNF combination of rows from X (where logical operators are applied component-wise) if and only if it is equal to the combination defined by

$$Y = \{\{x \in X \mid x(e) = 1\} \mid e \in E \wedge r(e) = 1\}.$$

In our setting, we can reuse this idea to efficiently find closedness defects and to construct the hypothesis. Notice that, even though the monad A formally requires the use of DNF formulae representing upsets, in the actual implementation we can use smaller formulae, e.g., Y above instead of its upward closure. In fact, it is easy to check that DNF combinations of rows are invariant under upward closure. As with non-deterministic and universal automata, we do not know of an efficient way to ensure consistency. As in the existing algorithms mentioned above, we could drop it altogether.

Writer automata. The examples considered so far involve existing classes of automata. To further demonstrate the generality of our approach, we introduce a new (as far as we know) type of automaton, which we call *writer automaton.*

The *writer monad* `Writer`$(X) = \mathbb{M} \times X$ for a finite monoid $\mathbb{M}$ has a unit $\eta_X \colon X \to \mathbb{M} \times X$ given by adding the unit e of the monoid, $\eta_X(x) = (e, x)$, and a multiplication $\mu_X \colon \mathbb{M} \times \mathbb{M} \times X \to \mathbb{M} \times X$ given by performing the monoid multiplication, $\mu_X(m_1, m_2, x) = (m_1 m_2, x)$. In Haskell, the writer monad is used for such tasks as collecting successive log messages, where the monoid is given by the set of sets or lists of possible messages and the multiplication adds a message.

The algebras for this monad are sets Q equipped with an $\mathbb{M}$-action. One may take the output object to be the set $\mathbb{M}$ with the monoid multiplication as its action. `Writer`-automata with a free state space can be represented as deterministic automata that have an element of $\mathbb{M}$ associated with each transition. The semantics of these is that the encountered $\mathbb{M}$-elements multiply along paths and finally multiply with the output of the last state to produce the actual output.

The basic learning algorithm is already of polynomial time complexity. In fact, to determine whether a given row is a combination of rows in the table, i.e., whether it is given by a monoid value applied to one of the rows in the table, one simply tries all of these values. This allows us to check for closedness, to minimize the generators, and to construct the succinct hypothesis, in polynominal time. Consistency involves comparing all ways of applying monoid values to rows and, for each comparison, at most $|A|$ further comparisons between one-letter extensions. The total number of comparisons is clearly polynomial in $|\mathbb{M}|$, $|S|$ and $|A|$.

8 IMPLEMENTATION

We have implemented the general $\mathsf{L}^\star_T$ algorithm in Haskell, taking full advantage of the monads provided by its standard library. Apart from the high-level implementation, our library provides

- a basic implementation for weighted automata over a finite semiring, with a polynomial time variation for the case where the semiring is a field[5];
- an implementation for non-deterministic automata that has polynomial time implementations for ensuring closedness and constructing the hypothesis, but not for ensuring consistency;
- a variation on the previous algorithm that uses the notion of consistency defined by Bollig et al. [2009];
- instantiations of the basic algorithm to the monad being $(-)+E$, for E a finite set of exceptions, and `Writer`, both of which result in polynomial time algorithms;

In this section we describe the main structure and ingredients of our library. After recalling monads in Haskell in Section 8.1, we start with the formalization of automata in Section 8.2. We then introduce teachers in Section 8.3 before exploring the actual learning algorithm in Section 8.4. We give details for the non-deterministic and weighted case, whose monads deserve a closer analysis.

8.1 Monads

We note that a monad in Haskell is specified as a *Kleisli triple* $(T, \eta, (-)^\sharp)$, where T assigns to every set X a set TX, η consists of a component $\eta_X : X \to TX$ for each set X, and $(-)^\sharp$ provides for each function $f : X \to TY$ an extension $f^\sharp : TX \to TY$. These need to satisfy

$$f^\sharp \circ \eta = f \qquad\qquad \eta^\sharp = \mathrm{id} \qquad\qquad (g^\sharp \circ f)^\sharp = g^\sharp \circ f^\sharp.$$

Kleisli triples are in a one-to-one correspondence with monads. On both sides of this correspondence we have the same T and η, which for a Kleisli triple are turned into a functor with a natural transformation by setting $Tf = (\eta \circ f)^\sharp$. Furthermore, $(-)^\sharp$ and μ are obtained from each other by $f^\sharp = \mu \circ Tf$ and $\mu = \mathrm{id}^\sharp$. Indeed, under this correspondence the $(-)^\sharp$ operation is a specific instance of the extension operation defined for a monad, with the T-algebra codomain restricted to free T-algebras. In Haskell, the η of the Kleisli triple is written `return`, and, given $f : X \to TY$ and $x \in TX$, $f^\sharp(x)$ is written `x >>= f` and referred to as the *bind* operation. Furthermore, for any $f : X \to Y$, Tf is given by `fmap f`.

Some basic **Set** monads cannot directly be written down in Haskell because their definition can only be given on types equipped with an equality check, or, for reasons of efficiency, a total order. For example, the `Set` type provided by `Data.Set` comes with a `union` function that has the following signature:

```
union :: Ord a => Set a -> Set a -> Set a
```

One will have to use unions in one way or another in defining the bind of the powerset monad. However, since this bind needs to be of type

```
(>>=) :: Set a -> (a -> Set b) -> Set b
```

and does not assume an `Ord` instance on `b`, the powerset monad cannot be defined in this way.

One solution is to delay the monadic computations in a wrapper type whose constructors are used to define a monad instance: the free monad. Specifically, we endow the *freer monad* of Kiselyov and Ishii [2015] with a constraint parameter:

```
data CFree c m a where
    Return :: a -> CFree c m a
    Bind :: (c b) => m b -> (b -> CFree c m a) -> CFree c m a
```

[5] Despite the assumption in the present paper that the monad preserves finite set, our implementation can learn weighted automata over infinite fields and thus implements the general algorithm introduced by Bergadano and Varricchio [1996], which was studied in a categorical context by Jacobs and Silva [2014] and van Heerdt et al. [2017].

Such a constrained free monad was first defined by George Giorgidze, but only for the specific case where m is Set and c is Ord.[6] On the constrained free monad we can define a complete Monad instance:

```
instance Monad (CFree c m) where
    return = Return
    f >>= g = case f of
        Return a -> g a
        Bind s h -> s `Bind` (h >=> g)
```

This is the same code as used by Kiselyov and Ishii [2015], but we note that on the last line, since s is the first argument of Bind in f, we know that the appropriate constrained needed to invoke Bind on the right-hand side, with again s as its first argument, is satisfied.

Finally, if there is a monad that is defined only on types satisfying a certain constraint, then we can convert from our free monad type with that constraint back to the actual "monad":

```
class ConstrainedMonad c m | m -> c where
    constrainedReturn :: (c a) => a -> m a
    constrainedBind :: (c a, c b) => m b -> (b -> m a) -> m a

unCFree :: (ConstrainedMonad c m, c a) => CFree c m a -> m a
unCFree f = case f of
    Return a -> constrainedReturn a
    Bind s g -> s `constrainedBind` (unCFree . g)
```

Note that operations such as equality checks for CFree c m use unCFree to delegate the operation to whatever is defined for m. This means that in code that abstracts from the monad we seem to be working with m as a monad.

As an example, the Set "monad" becomes

```
instance ConstrainedMonad Ord Set where
    constrainedReturn = Set.singleton
    s `constrainedBind` f = Set.unions [f a | a <- Set.toList s]
```

We may then use CFree Ord Set as the monad.

To implement the free semimodule monad in Haskell, we use the Map type from Data.Map. Note that the monad will be defined in the first argument for that type, so we need to create an auxiliary type to swap the arguments.

```
newtype Linear s k = Linear {fromLinear :: Map k s}
```

Defining the monad again requires Ord constraints.

```
instance (Semiring s, Eq s) => ConstrainedMonad Ord (Linear s) where
    constrainedReturn a = Linear $ Map.singleton a mempty
    l `constrainedBind` f = Linear .
        foldl' (\m (k, s) -> ladd m . lscale s . fromLinear $ f k) Map.empty .
        Map.toList . lminimize $ fromLinear l
```

The function lscale scales a map by an element from the semiring; ladd adds two maps together. Both operations are pointwise. The monad we can use is CFree Ord (Linear s).

8.2 Automata

We model an automaton as a simple deterministic automaton.

[6] https://hackage.haskell.org/package/set-monad-0.2.0.0

```
data Aut a o q = Aut {
    initial :: q,
    delta :: q -> a -> q,
    out :: q -> o }
```

For such automata, we can easily implement reachability and language functions, as well as bisimulation. Bisimulation is used to realize exact equivalence queries for the teachers that hold an automaton accepting the language to be learned. To optimize for the monad in the same way the learning algorithm is optimized, we use *bisimulation up to context* [Rot et al. 2013; Sangiorgi 1998].

```
bisimT :: (Eq o) => ((t q, t r) -> [(t q, t r)] -> Bool) ->
    [a] -> Aut a o (t q) -> Aut a o (t r) -> Maybe [a]
```

Here `t` represents the monad that we optimize for. Up to context means that, when considering a pair `p :: (t q, t r)` of next states and the current relation `b :: [(t q, t r)]`, the pair `p` does not need to be added to the relation if it can be obtained as a combination of the elements of `b`, using the free algebra structures of `t q` and `t r`. The first argument of `bisimT` is a function that should determine this. Because of this abstraction, we do not actually need to constrain `t` to be a monad here. For the `Identity` monad, one can simply use `elem` as the first argument. The second argument is the alphabet.

Succinct automata optimized by a monad `t` enjoy a more concrete representation involving maps.

```
data SAut t a o q = SAut {
    sinitial :: t q,
    sdelta :: Map q (Map a (t q)),
    sout :: Map q o }
```

This is the type of the automata that the $\mathtt{L}^\star_T$ implementation learns. The concrete representation allows the automaton to be displayed and exported. Of course, one can determinize a succinct automaton using `t`-algebras for `a -> t q` and `o`.

```
det :: (Monad t, Ord q, Ord a) =>
    Alg t (a -> t q) -> Alg t o -> SAut t a o q -> Aut a o (t q)
```

The type `Alg t x` is defined to be `t x -> x`. We allow an arbitrary algebra on `a -> t q` rather than assuming the component `t (a -> t q) -> a -> t (t q)` of the distributive law used in earlier sections because this allows us to run the delayed monadic computations discussed earlier, which would otherwise pile up and cause serious performance issues.

8.3 Teaching

A teacher in our implementation is an object that comprises membership and equivalence functions. It also records the alphabet.

```
data Teacher s a o q = Teacher {
    membership :: [a] -> s o,
    equivalence :: Aut a o q -> s (Maybe [a]),
    alphabet :: [a] }
```

`Teacher` objects are parameterized by a monad `s` that serves a different purpose than optimizing the learning algorithm: it is the monad of side-effects allowed by the implementation of queries. Whereas the `Identity` monad suffices for a predefined automaton, one may have to use the `IO` monad to interact with an actual black-box system. By allowing an arbitrary monad rather than assuming the `IO` monad, we are able to build features such as query counters and a cache on top of any teacher through the use of *monad transformers*. A monad transformer provides for any monad a new monad into which the original one can be embedded. For example, the `StateT x s` monad adds a state with values in x to an existing monad s. This is the transformer that enables the addition of query counters and a cache to a teacher:

```
countTeacher :: (Monad s) =>
    Teacher s a o q -> Teacher (StateT (Int, Int) s) a o q
cacheTeacher :: (Monad s, Ord a) =>
    Teacher s a o q -> Teacher (StateT (Map [a] o) s) a o q
```

The most basic teacher holds an automaton that it uses to determine membership and equivalence, the latter of which is implemented through bisimulation.

```
autTeacherT :: (Monad s, Eq o) => ((t q, t r) -> [(t q, t r)] -> Bool) ->
    [a] -> Aut a o (t q) -> Teacher s a o (t r)
```

It implements a Teacher for any monad s because it does not have any side-effects.

We also provide a teacher that implements equivalence queries through random testing.

```
randomTeacher :: (Monad s, Eq o) => Int -> State StdGen [a] ->
    [a] -> ([a] -> s o) -> Teacher (StateT StdGen s) a o q
```

Its first argument is the number of tests per equivalence query, while the second argument samples test words: StdGen is a random number generator. Once more we use the StateT monad transformer, in this case to add a random number generator state to the monad s that the membership query function, which is the last argument, may use. This query function is used both for membership queries and for generated test queries. Note that this particular teacher does not give any guarantees on the validity of positive responses to equivalence queries. We do also provide the random sampling teacher suggested by Angluin [1987], which guarantees that on a positive answer the hypothesis is *probably approximately correct*, a notion introduced by Valiant [1984].

```
pacTeacher :: (Monad s, Eq o) => Double -> Double -> State StdGen [a] ->
    [a] -> ([a] -> s o) -> Teacher (StateT (Int, StdGen) s) a o q
```

Here the first argument is the accuracy ϵ, while the second one is the confidence ∂. Both should be values between 0 and 1. If $d\colon A^* \to [0,1]$ is the distribution represented by the third argument (converting between Haskell types and sets for convenience) and $l_1, l_2\colon A^* \to O$ are the languages of the hypothesis and the target, the guarantee is that, with probability at least $1 - \partial$, $\sum_{u \in A^*, l_1(u) \neq l_2(u)} d(u) \leq \epsilon$. Compared to randomTeacher, an Int has been added to the state because the number of tests depends on the number of equivalence queries that have already been asked.

8.4 Learning

We define a Learner type that allows us to switch between variations on $\mathsf{L}^\star_T$ and to optimize certain specific procedures.

```
data Learner t a o = Learner {
    decomposeRow :: ObservationTable a o -> [[a]] -> [o] -> Maybe (t [a]),
    consistencyDefect :: Maybe (ObservationTable a o -> Maybe [a]),
    ceh :: CEHandler }
```

The function decomposeRow takes an observation table, a list of labels l, and a row r, and determines whether r can be obtained as a combination of the rows with labels in l. If this is the case, it returns the combination, which has type t [a]. This function is used to check closedness, to minimize the labels used as states for the hypothesis, and to construct the hypothesis. If consistencyDefect is set to Nothing, it indicates that consistency should be solved by solving closedness for what we call the *transpose* of the table (swapping S and E and reversing their words while considering the reverse of the target language as the target language); otherwise, it contains a function that given an observation table produces a new column to fix one of its consistency defects, unless the table is already consistent. Solving closedness for the transpose of the table always ensures consistency, but in general it may add more columns than necessary. Lastly, `CEHandler` is a type that enumerates our adaptations of the three counterexample handling methods: the original one by Angluin [1987], the one by Maler and Pnueli [1995], and the one by Rivest and Schapire [1993].

To enable basic implementations of `decomposeRow` and `consistencyDefect` that work for any monad T (preserving finite sets), we need to be able to loop over the values of TS. In order to facilitate this, there is a class `Concrete f` whose only member function turns a list of values of any type into a list of values with type `f` applied to that type. It is intended to be the concrete application of a functor to a set (represented as a list). We provide the functions `lazyDecomposeRow` and `lazyConsistencyDefect`, both conditioned with a `Concrete t` constraint, which directly enable a basic version of the learning algorithm.

To optimize the algorithm in a specific setting, a programmer only has to adjust these two functions. We provide such optimized functions for the cases of non-deterministic and weighted automata (over a field). Regarding the former case, we provide `crfsaDecompose` and `scrfsaDecompose`, which are essentially the right inverses corresponding to the canonical and simplified canonical RFSA, respectively, as explained in Example 5.5. Our optimized weighted algorithm uses Gaussian elimination in a function called `gaussianDecomposeRow` and solves consistency by solving closedness for the transpose of the table, a method readily available regardless of the monad.

Enabling our adaptation of the counterexample handling method due to Rivest and Schapire requires an additional condition. Recall that this method requires us to pose membership queries for combinations of words, which can be done by extending the membership query function (the language) of type `[a] -> o` to one of type `t [a] -> o` using the algebra structure defined on `o`. However, our membership query function actually has type `[a] -> s o`, and there is no reason to assume any interaction between `s` and `t`. As a workaround, we will assume an instance of `Supported` for the monad `t`, where `Supported` is a class defined as follows:

```
class Supported f where
    supp :: (Ord a) => f a -> [a]
```

Given any `u :: f a` and `g :: a -> b`, we require `supp u` to be such that the computation of `fmap g u` only evaluates `g` on the elements of `supp u`. Naturally, we want `supp u` to be as small as possible: it should contain exactly those elements of type `a` that are present in `u`. As an example, recall that the free semimodule monad with values in a semiring `s` can be defined on a type `a` as `Map a s`, where we identify a missing value for an element with that element being assigned zero. Given `u :: Map a s`, `supp u` is given by the keys of the map `u` that are assigned a non-zero value.

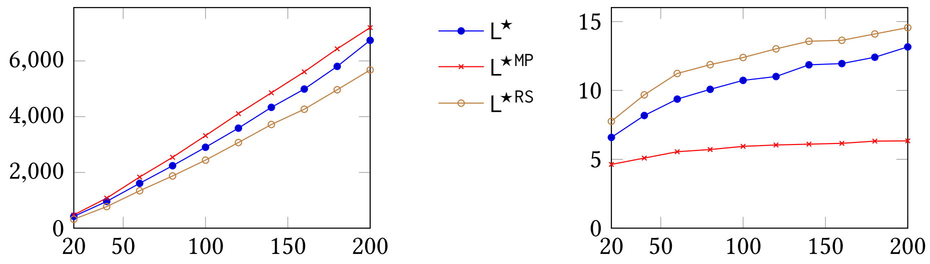
Using the instance for a monad `t`, the membership query function can be extended by querying the words in the support of a given element of `t [a]` sequentially, constructing a partial membership query function defined only on that support, and evaluating the extension of that function. This method works because we assume that the side-effects exhibited by `s` do not influence future membership queries.

Finally, our general $\mathsf{L}^\star_T$ implementation has the following signature:

```
lStarT :: (Monad s, Monad t, Supported t, Ord a, Eq o) =>
    Alg t (a -> t [a]) -> Alg t o ->
    Teacher s a o (t [a]) -> Learner t a o -> s (SAut t a o [a])
```

9 EXPERIMENTS

In this section we analyze the performance, in terms of number of queries, of several variations of our algorithm by running them on randomly generated WFAs, NFAs, and plain Moore automata. Our aim is to show the effect of exploiting the right monad and of using our adapted optimized



(a) Membership queries

(b) Equivalence queries

Fig. 5. $\mathsf{L}^\star$ variations on random DFAs.

counterexample handling method. The experiments are run using the implementation discussed in Section 8. In all cases we use an alphabet of size 3. Random Moore automata are generated by choosing for each state an output and further for each input symbol a next state using uniform distributions. The WFAs are over the field of size 5. Here the outputs are chosen in the same way, and for each pair of states and each input symbol, we create a transition symbol from the first to the second state with a random weight chosen uniformly. We take the average of 100 iterations for each of the sizes for which we generate automata. Membership query results in tables will be rounded to whole numbers. We use bisimulation to find counterexamples in all experiments, exploiting the fact that the target automaton will be known. We cache membership queries so that the counts exclude duplicates.

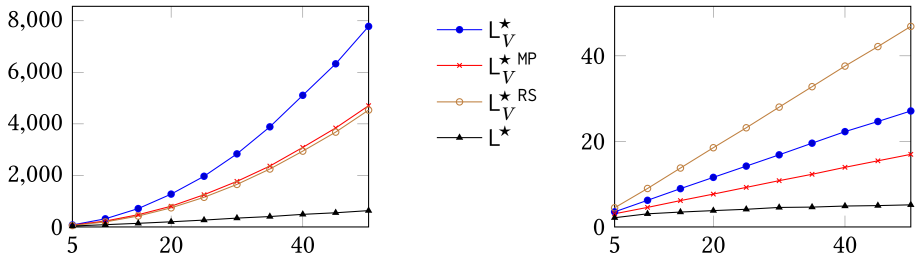
For reference, Figure 5 compares $\mathsf{L}^\star$ and the two counterexample handling variations by Maler and Pnueli (denoted MP) and by Rivest and Schapire (denoted RS), on randomly generated DFAs of size 20 through 200 with increments of 20. Compared to $\mathsf{L}^\star$, both $\mathsf{L}^{\star\mathsf{MP}}$ and $\mathsf{L}^{\star\mathsf{RS}}$ remove the need for consistency checks. Interestingly, whereas $\mathsf{L}^{\star\mathsf{RS}}$ compared to $\mathsf{L}^\star$ improves in membership queries and worsens in equivalence queries, the situation is reversed for $\mathsf{L}^{\star\mathsf{MP}}$.

9.1 $\mathsf{L}^\star_V$

In Table 1 we compare the performance of $\mathsf{L}^\star$ with that of $\mathsf{L}^\star_V$. (Recall that V is the free vector space monad.) Here $\mathsf{L}^\star$ is the obvious generalization of the original $\mathsf{L}^\star$ algorithm to learn Moore automata—DFAs with outputs in an arbitrary set, which here is the field with five elements. Thus, as opposed to $\mathsf{L}^\star_V$, $\mathsf{L}^\star$ ignores the vector space structure on the output set. In both cases we consider the three different counterexample handling methods. The algorithms are run on randomly generated WFAs of sizes 1 through 4. As expected, each $\mathsf{L}^\star_V$ variation provides a massive gain over the corresponding $\mathsf{L}^\star$ variation in terms of membership queries, and a more modest one in terms of equivalence

	MQs						EQs					
Size	$\mathsf{L}^\star$	$\mathsf{L}^\star_V$	$\mathsf{L}^{\star\mathsf{MP}}$	$\mathsf{L}^{\star\ \mathsf{MP}}_V$	$\mathsf{L}^{\star\mathsf{RS}}$	$\mathsf{L}^{\star\ \mathsf{RS}}_V$	$\mathsf{L}^\star$	$\mathsf{L}^\star_V$	$\mathsf{L}^{\star\mathsf{MP}}$	$\mathsf{L}^{\star\ \mathsf{MP}}_V$	$\mathsf{L}^{\star\mathsf{RS}}$	$\mathsf{L}^{\star\ \mathsf{RS}}_V$
1	10	4	10	4	10	4	1.00	1.00	1.00	1.00	1.00	1.00
2	105	15	154	15	104	11	1.86	1.73	1.86	1.73	1.86	1.73
3	845	27	1003	27	844	24	2.84	2.10	2.16	2.14	3.00	2.81
4	5570	50	7904	50	5567	40	3.71	2.88	2.90	2.83	3.97	3.78

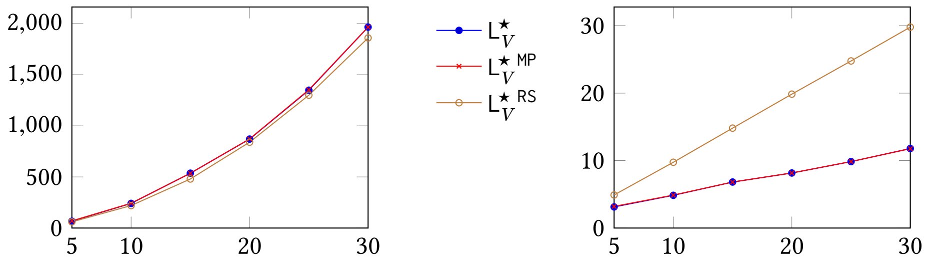
Table 1. $\mathsf{L}^\star$ variations and $\mathsf{L}^\star_V$ variations on random WFAs.



(a) Membership queries

(b) Equivalence queries

Fig. 6. L$^{\star}_{V}$ variations and L* on random Moore automata.



(a) Membership queries

(b) Equivalence queries

Fig. 7. L$^{\star}_{V}$ variations on random WFAs.

queries. Comparing the results of the L* variations, we see that the membership query results of L* and L$^{\star\mathsf{RS}}$ are extremely close together. Other than that, the ordering of the counterexample handling methods is the same as with the DFA experiments. The L$^{\star}_{V}$ variations will be compared in more detail later.

Now we run L* and variations of L$^{\star}_{V}$ on randomly generated Moore automata of sizes 5 through 50 with increments of 5. We chose to compare the L$^{\star}_{V}$ variations only to L* because of its average performance in between L$^{\star\mathsf{MP}}$ and L$^{\star\mathsf{RS}}$ as seen in Figure 5. The results are shown in Figure 6. We see that, in terms of membership queries, both RS and MP counterexample handling methods improve over the one by Angluin in this setting, and MP performs best in terms of either query type. In these experiments, L* performs much better than the algorithms that attempt to take advantage of the non-existent vector space structure. Together with the results in Table 1, this is consistent with the findings of Angluin et al. [2015]: they found that for DFAs and non-deterministic, universal, and alternating automata, the adaptation of L* that takes advantage of the exact type of structure of the randomly generated target automata performs the best.

Figure 7 illustrates the performance of L$^{\star}_{V}$ variations on randomly generated WFAs. Here we generated WFAs of sizes 5 through 30 with increments of 5. We emphasize that in Table 1 we could not go beyond size 4, because of performance issues with L*. There is hardly any difference between the use of Angluin's counterexample handling method and MP, neither in terms of membership queries, nor in terms of equivalence queries. Interestingly, while the RS method performs worse than the other methods in terms of equivalence queries, as usual, it provides no significant gain in

	MQs					EQs				
Size	$\mathsf{L}^\star$	$\mathsf{NL}^{\star\mathsf{MP}}$	$\mathsf{NL}^{\star\mathsf{MP}-}$	$\mathsf{NL}^{\star\mathsf{RS}}$	$\mathsf{NL}^{\star\mathsf{RS}-}$	$\mathsf{L}^\star$	$\mathsf{NL}^{\star\mathsf{MP}}$	$\mathsf{NL}^{\star\mathsf{MP}-}$	$\mathsf{NL}^{\star\mathsf{RS}}$	$\mathsf{NL}^{\star\mathsf{RS}-}$
4	138	79	82	55	54	3.75	3.01	3.64	3.59	4.64
8	1792	666	729	389	381	10.38	6.52	8.83	9.37	14.10
12	11130	2467	2701	1331	1286	18.93	11.08	14.92	17.88	27.61
16	38256	5699	6240	3036	2999	28.81	15.75	22.59	27.29	45.53

Table 2. $\mathsf{L}^\star$ and $\mathsf{NL}^\star$ variations on random NFAs.

terms of membership queries. We ran these experiments also with the variations on the MP and RS algorithms where we drop the consistency checks. In both cases the differences were negligible.

9.2 NL*

We now consider learning algorithms for NFAs. To generate random NFAs, we use the strategy introduced by Tabakov and Vardi [2005] with a transition density of 1.25, meaning that for each input symbol there are on average 1.25 transitions originating from each state. According to Tabakov and Vardi, this density results in the largest equivalent minimal DFAs. Like Tabakov and Vardi, we let half of the states be accepting. We ran several variations of $\mathsf{L}^\star$ and $\mathsf{NL}^\star$ on randomly generated NFAs of sizes 4 through 16 with increments of 4. The results are shown in Table 2. Here $\mathsf{NL}^{\star\mathsf{MP}}$ refers to the original algorithm by Bollig et al. [2009], with their notion of consistency; $\mathsf{NL}^{\star\mathsf{RS}}$ is the same algorithm, but using the counterexample handling method that we adapted from Rivest and Schapire's. The variations $\mathsf{NL}^{\star\mathsf{MP}-}$ and $\mathsf{NL}^{\star\mathsf{RS}-}$ drop the consistency checks altogether. Unfortunately, doing the full consistency check was not computationally feasible. As expected, the $\mathsf{NL}^\star$ algorithms yield a great improvement over $\mathsf{L}^\star$ in terms of membership queries, and in most cases they also improve in terms of equivalence queries. This was already observed by Bollig et al. The exception is $\mathsf{NL}^{\star\mathsf{RS}-}$, which, despite having the best membership query results, requires by far the most equivalence queries. As happened to $\mathsf{L}^\star$ on DFAs, switching within $\mathsf{NL}^\star$ from the MP to the RS counterexample handling method improves the performance in terms of membership queries and worsens it in terms of equivalence queries. Dropping consistency altogether turns out to increase both query numbers.

10 CONCLUSION

We have presented $\mathsf{L}^\star_T$, a general adaptation of $\mathsf{L}^\star$ that uses monads to learn an automaton with algebraic structure, as well as a method for finding a succinct equivalent based on its generators. Furthermore, we adapted the optimized counterexample handling method of Rivest and Schapire [1993] to this setting and discussed instantiations to non-deterministic, universal, partial, weighted, alternating, and writer automata. We have provided a prototype implementation in Haskell, using which we obtained experimental results confirming that exploiting the algebraic structure reduces the number of queries posed. The results also reveal that the best counterexample handling method depends on the type of automata considered and the algebraic structure exploited by the algorithm. We found that there is a significant gain in membership queries compared to the $\mathsf{NL}^\star$ algorithm by Bollig et al. [2009] when using our adapted optimized counterexample handling method.

Related Work. This paper builds on and extends the theoretical toolkit of van Heerdt [2016]; van Heerdt et al. [2017], who are developing a categorical automata learning framework (CALF) in which learning algorithms can be understood and developed in a structured way.

An adaptation of $\mathsf{L}^\star$ that produces NFAs was first developed by Bollig et al. [2009]. Their algorithm learns a special subclass of NFAs consisting of RFSAs, which were introduced by Denis et al. [2002]. Angluin et al. [2015] unified algorithms for NFAs, universal automata, and alternating automata, the latter of which was further improved by Berndt et al. [2017]. We are able to provide a more general framework, which encompasses and goes beyond those classes of automata. Moreover, we study optimized counterexample handling, which Angluin et al. [2015]; Berndt et al. [2017]; Bollig et al. [2009] do not consider.

The algorithm for weighted automata over a (not necessarily finite) field was studied in a category theoretical context by Jacobs and Silva [2014] and elaborated on by van Heerdt et al. [2017]. The algorithm itself was introduced by Bergadano and Varricchio [1996]. The present paper provides the first, correct-by-construction implementation of the algorithm. The theory of succinct automata used for our hypotheses is based on the work of Arbib and Manes [1975], revamped to more recent category theory.

Our library is currently a prototype, which is not intended to compete with a state-of-the-art tool such as LearnLib [Isberner et al. 2015] or other automata learning libraries like libalf [Bollig et al. 2010]. Our Haskell implementation does not provide the computational efficiency achieved by LearnLib, which furthermore includes the TTT-algorithm with its optimized data structure that replaces the observation table by a tree [Isberner et al. 2014]. Such optimization is ad-hoc for DFAs, and an extension to other classes of automata is not trivial. First steps in this direction have been done by [van Heerdt et al. 2017], who have studied the tree data structure in a more general setting. We intend to further pursue investigation in this direction, in order to allow for optimized data structures in a future version of our library. We note that, although libalf supports NFAs, none of the existing tools and libraries offers the flexibility of our library, in terms of available optimizations and classes of models that can be learned.

Future Work. Whereas our general algorithm effortlessly instantiates to monads that preserve finite sets, a major challenge lies in investigating monads that do not enjoy this property. In fact, although the algorithm for weighted automata generalizes to an infinite field [Jacobs and Silva 2014; van Heerdt et al. 2017], for an infinite semiring in general we cannot guarantee termination. This is because a finitely generated semimodule may have an infinite chain of strict submodules. Intuitively, this means that while fixing closedness defects increases the size of the hypothesis state space semimodule, an infinite number of steps may be needed to resolve all closedness defects. There are however subclasses of semirings for which a generalization should be possible, e.g., Noetherian or, more generally, proper semirings, which were recently studied by Milius [2017]. Moreover, we expect that $\mathsf{L}^\star_T$ can be generalized from the category of sets to *locally finitely presentable* categories.

As a result of the correspondence between learning and conformance testing [Berg et al. 2005; van Heerdt et al. 2017], it should be possible to include in our framework the W-method [Chow 1978], which is often used in case studies deploying $\mathsf{L}^\star$ [e.g., Chalupar et al. 2014; de Ruiter and Poll 2015]. We defer a thorough investigation of conformance testing to future work.

REFERENCES

Fides Aarts, Joeri de Ruiter, and Erik Poll. 2013. Formal models of bank cards for free. In *ICSTW*. IEEE Computer Society, 461–468.

Benjamin Aminof, Orna Kupferman, and Robby Lampert. 2010. Reasoning about online algorithms with weighted automata. *ACM Trans. Algorithms* 6, 2 (2010), 28:1–28:36.

Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inform. Comput.* 75 (1987), 87–106.

Dana Angluin, Sarah Eisenstat, and Dana Fisman. 2015. Learning regular languages via alternating automata. In *IJCAI*. 3308–3314.

Michael A. Arbib and Ernest G. Manes. 1975. Fuzzy machines in a category. *Bulletin of the AMS* 13 (1975), 169–210.

Christel Baier, Marcus Größer, and Frank Ciesinski. 2009. Model checking linear-time properties of probabilistic systems. In *Handbook of Weighted automata*.

Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. 2005. On the correspondence between conformance testing and regular inference. In *FASE*, Vol. 3442. 175–189.

Francesco Bergadano and Stefano Varricchio. 1996. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM J. Comput.* 25 (1996), 1268–1280.

Sebastian Berndt, Maciej Liśkiewicz, Matthias Lutter, and Rüdiger Reischuk. 2017. Learning Residual Alternating Automata. In *AAAI*. 1749–1755.

Meven Bertrand. 2017. Coalgebraic Determinization of Alternating Automata. (2017). http://jurriaan.creativecode.org/wp-content/uploads/2017/10/alt.pdf.

Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. 2008. *Angluin-Style Learning of NFA (Research Report LSV-08-28)*. Technical Report. ENS Cachan.

Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. 2009. Angluin-Style Learning of NFA. In *IJCAI*, Vol. 9. 1004–1009.

Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R Piegdon. 2010. libalf: The automata learning framework. In *CAV*. 360–364.

Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. 2014. Automated Reverse Engineering using Lego®. In *WOOT*.

Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. 2008. Quantitative Languages. In *CSL*. 385–400.

Chia Yuan Cho, Domagoj Babić, Eui Chul Richard Shin, and Dawn Song. 2010. Inference and Analysis of Formal Models of Botnet Command and Control Protocols. In *CCS*. ACM, 426–439.

Tsun S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Software Eng.* 4 (1978), 178–187.

Joeri de Ruiter and Erik Poll. 2015. Protocol state fuzzing of TLS implementations. In *USENIX Security*. 193–206.

François Denis, Aurélien Lemay, and Alain Terlutte. 2002. Residual finite state automata. *Fundamenta Informaticae* 51 (2002), 339–368.

Manfred Droste and Paul Gastin. 2005. Weighted Automata and Weighted Logics. In *ICALP*. 513–525.

Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification (LNCS)*, Vol. 8734. 307–322.

Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-Source LearnLib. In *CAV (LNCS)*, Vol. 9206. 487–495.

Bart Jacobs and Alexandra Silva. 2014. Automata Learning: A Categorical Perspective. In *Horizons of the Mind*, Vol. 8464. 384–406.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 94–105.

Dexter C. Kozen. 2012. *Automata and computability*. Springer Science & Business Media.

Denis Kuperberg. 2014. Linear Temporal Logic for Regular Cost Functions. *Logical Methods in Computer Science* 10, 1 (2014).

Oded Maler and Amir Pnueli. 1995. On the Learnability of Infinitary Regular Sets. *Inform. and Comput.* 118 (1995), 316–326.

Stefan Milius. 2017. Proper functors. (2017). Submitted, copy obtained in personal communication.

Ronald L. Rivest and Robert E. Schapire. 1993. Inference of Finite Automata Using Homing Sequences. *Inform. Comput.* 103 (1993), 299–347.

Jurriaan Rot, Marcello Bonsangue, and Jan Rutten. 2013. Coalgebraic bisimulation-up-to. In *SOFSEM*. 369–381.

Davide Sangiorgi. 1998. On the bisimulation proof method. *Mathematical Structures in Computer Science* 8 (1998), 447–479.

Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. 2016. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In *IFM*, Vol. 9681. 311–325.

Deian Tabakov and Moshe Y Vardi. 2005. Experimental evaluation of classical automata constructions. In *LPAR*, Vol. 5. 396–411.

Frits W. Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (2017), 86–95.

Leslie G. Valiant. 1984. A theory of the learnable. *Commun. ACM* 27 (1984), 1134–1142.

Gerco van Heerdt. 2016. *An Abstract Automata Learning Framework*. Master's thesis. Radboud University Nijmegen.

Gerco van Heerdt, Matteo Sammartino, and Alexandra Silva. 2017. CALF: Categorical Automata Learning Framework. In *CSL*. 29:1–29:24.

A OMITTED PROOFS

Proposition 3.11. *The minimal Moore automaton accepting $\mathcal{L}$ is finite if and only if the minimal T-automaton accepting $\mathcal{L}$ is finite.*

Proof. The left to right implication is proved by freely generating a T-automaton from the Moore one via the monad unit, and by recalling that T preserves finite sets. The resulting T-automaton accepts $\mathcal{L}$ and is finite, therefore any of its quotients, including the minimal T-automaton accepting $\mathcal{L}$, is finite. Analogously, the right to left implication follows by forgetting the algebraic structure of the T-automaton: this yields a finite Moore automaton accepting $\mathcal{L}$. □

Proposition 5.3. *Any succinct hypothesis of $\mathcal{H}$ accepts the same language as $\mathcal{H}$.*

Proof. Assume a right inverse $i\colon H \longrightarrow T(S')$ of $e\colon T(S') \longrightarrow H$. We first prove $o_{\mathcal{H}} \circ e = o_{\mathcal{S}}$, by induction on the length of words. For all $U \in T(S')$, we have

$$\begin{aligned}
o_{\mathcal{H}}(e(U))(\varepsilon) &= \mathsf{out}_{\mathcal{H}}(e(U)) && \text{(definition of } o_{\mathcal{H}}\text{)}\\
&= \mathsf{out}_{\mathcal{H}}(\mathsf{row}^{\sharp}_{\mathsf{t}}(U)) && \text{(definition of } e\text{)}\\
&= \mathsf{row}^{\sharp}_{\mathsf{t}}(U)(\varepsilon) && \text{(definition of } \mathsf{out}_{\mathcal{H}}\text{)}\\
&= \mathsf{out}_{\mathcal{S}}(U) && \text{(definition of } \mathsf{out}_{\mathcal{S}}\text{)}\\
&= o_{\mathcal{S}}(U)(\varepsilon) && \text{(definition of } o_{\mathcal{S}}\text{).}
\end{aligned}$$

Now assume that for a given $v \in A^*$ and all $U \in T(S')$ we have $o_{\mathcal{H}}(e(U))(v) = o_{\mathcal{S}}(U)(v)$. Then, for all $U \in T(S')$ and $a \in A$,

$$\begin{aligned}
o_{\mathcal{H}}(e(U))(av) &= o_{\mathcal{H}}(\delta_{\mathcal{H}}(e(U))(a))(v) && \text{(definition of } o_{\mathcal{H}}\text{)}\\
&= o_{\mathcal{H}}(\delta_{\mathcal{H}}(\mathsf{row}^{\sharp}_{\mathsf{t}}(U))(a))(v) && \text{(definition of } e\text{)}\\
&= o_{\mathcal{H}}(\mathsf{row}^{\sharp}_{\mathsf{b}}(U)(a))(v) && \text{(definition of } \delta_{\mathcal{H}}\text{)}\\
&= (o_{\mathcal{H}} \circ e \circ i)(\mathsf{row}^{\sharp}_{\mathsf{b}}(U)(a))(v) && (e \circ i = \mathsf{id}_H)\\
&= (o_{\mathcal{S}} \circ i)(\mathsf{row}^{\sharp}_{\mathsf{b}}(U)(a))(v) && \text{(induction hypothesis)}\\
&= o_{\mathcal{S}}(\delta_{\mathcal{S}}(U)(a))(v) && \text{(definition of } \delta_{\mathcal{S}}\text{)}\\
&= o_{\mathcal{S}}(U)(av) && \text{(definition of } o_{\mathcal{S}}\text{).}
\end{aligned}$$

From this we see that

$$\begin{aligned}
o_{\mathcal{S}}(\mathsf{init}_{\mathcal{S}}) &= (o_{\mathcal{S}} \circ i \circ \mathsf{row}_{\mathsf{t}})(\varepsilon) && \text{(definition of } \mathsf{init}_{\mathcal{S}}\text{)}\\
&= (o_{\mathcal{H}} \circ e \circ i \circ \mathsf{row}_{\mathsf{t}})(\varepsilon) && (o_{\mathcal{H}} \circ e = o_{\mathcal{S}})\\
&= (o_{\mathcal{H}} \circ \mathsf{row}_{\mathsf{t}})(\varepsilon) && (e \circ i = \mathsf{id}_H)\\
&= o_{\mathcal{H}}(\mathsf{init}_{\mathcal{H}}) && \text{(definition of } \mathsf{init}_{\mathcal{H}}\text{).}
\end{aligned}$$

□

Proposition 5.4. *The following algorithm returns a minimal set of generators for the table:*

```
$S' \leftarrow S$
while there are $s \in S'$ and $U \in T(S' \setminus \{s\})$ s.t. $\mathsf{row}^{\sharp}_{\mathsf{t}}(U) = \mathsf{row}_{\mathsf{t}}(s)$
    $S' \leftarrow S' \setminus \{s\}$
return $S'$
```

Proof. Minimality is obvious, as S' not being minimal would make the loop guard true.

We prove that the returned set is a set of generators. For clarity, we denote by $d_{S'}\colon S \longrightarrow T(S')$ the function associated with a set of generators S'. The main idea is incrementally building $d_{S'}$ while building S'. In the first line, S is a set of generators, with $d_S = \eta_S \colon S \longrightarrow T(S)$. For the loop, suppose S' is a set of generators. If the loop guard is false, the algorithm returns the set of generators S'. Otherwise, suppose there are there are $s \in S'$ and $U \in T(S' \setminus \{s\})$ such that $\mathsf{row}^\sharp_{\mathsf{t}}(U) = \mathsf{row}_{\mathsf{t}}(s)$. Then there is a function

$$f \colon S' \to T(S' \setminus \{s\}) \qquad\qquad f(s') = \begin{cases} U & \text{if } s' = s \\ \eta(s') & \text{if } s' \neq s \end{cases}$$

that satisfies $\mathsf{row}_{\mathsf{t}}(s') = \mathsf{row}^\sharp_{\mathsf{t}}(f(s'))$ for all $s' \in S'$, from which it follows that $\mathsf{row}^\sharp_{\mathsf{t}}(U') = \mathsf{row}^\sharp_{\mathsf{t}}(f^\sharp(U'))$ for all $U' \in T(S')$. Then we can set $d_{S' \setminus \{s\}}$ to $f^\sharp \circ d_{S'} \colon S \longrightarrow T(S' \setminus \{s\})$ because $\mathsf{row}_{\mathsf{t}}(s') = \mathsf{row}^\sharp_{\mathsf{t}}(d_{S' \setminus \{s\}}(s'))$ for all $s' \in S$. Therefore, $S' \setminus \{s\}$ is a set of generators. $\square$