Active Learning for Extended Finite State Machines[1][2]

Sofia Cassel* and Falk Howar† and Bengt Jonsson* and Bernhard Steffen‡

*Dept. of Information Technology, Uppsala University, Sweden, `{sofia.cassel|bengt.jonsson}@it.uu.se`
†IPSSE, TU Clausthal, Germany, `falk.howar@tu-clausthal.de`
‡Chair for Programming Systems, TU Dortmund, Germany, `steffen@cs.tu-dortmund.de`

Abstract. We present a black-box active learning algorithm for inferring extended finite state machines (EFSM)s by dynamic black-box analysis. EFSMs can be used to model both data flow and control behavior of software and hardware components. Different dialects of EFSMs are widely used in tools for model-based software development, verification, and testing. Our algorithm infers a class of EFSMs called *register automata*. Register automata have a finite control structure, extended with variables (registers), assignments, and guards. Our algorithm is parameterized on a particular *theory*, i.e., a set of operations and tests on the data domain that can be used in guards.

Key to our learning technique is a novel learning model based on so-called *tree queries*. The learning algorithm uses tree queries to infer symbolic data constraints on parameters, e.g., sequence numbers, time stamps, identifiers, or even simple arithmetic. We describe sufficient conditions for the properties that the symbolic constraints provided by a tree query in general must have to be usable in our learning model. We also show that, under these conditions, our framework induces a generalization of the classical Nerode equivalence and canonical automata construction to the symbolic setting. We have evaluated our algorithm in a black-box scenario, where tree queries are realized through (black-box) testing. Our case studies include connection establishment in TCP and a priority queue from the Java Class Library.

1. Introduction

The model-based approach to development, verification, and testing of software systems is a key path towards efficient development of reliable software systems. Models of software components are the basis for model checking [CGP01], model-based test generation [BJK+04, UL07], for composition of components [Arb04], for checking correctness of API usage [BBC+06], etc. However, the application of model-based methods is hampered by the current lack of adequate models and specifications for most actual systems, largely due to the significant manual effort needed for constructing models.

To address the problem of nonexisting or outdated models of component behavior, techniques for automatically generating such models are being developed. These techniques can be based on static analysis, dynamic analysis, or a combination thereof. In this paper, we are interested in a particular dynamic analysis

---

technique, *active automata learning* [Ang87, RS93], using which we can infer automata models that represent the dynamic behavior of a software or hardware component. Active automata learning can be deployed in a *black-box* setting, i.e., where we only have access to a component's interface, and where information about a component's behavior is obtained by sending input to it and observing what output it produces.

Mature techniques, based on active automata learning, are available for generating finite-state models that describe *control flow*, i.e., possible orderings of interactions between a component and its environment [HHNS02, HNS03, ABL02, SL07]. Their usefulness has been demonstrated, e.g., for mining APIs [ABL02], supporting model-based testing [HHNS02, WBDP10] and conformance testing [AKT+12], and for analyzing security protocols [SL07, GIO12]. Perhaps the most well-known algorithm for inferring finite automata is L^* [Ang87], which has been implemented in the LearnLib framework [IHS15]. However, in many situations it is crucial for models to also be able to describe *data flow*, i.e., constraints on data parameters that are passed when the component interacts with its environment, as well as the mutual influence between control flow and data flow. For instance, models of protocol components must describe how different parameter values in sequence numbers, identifiers, etc. influence the control flow, and vice versa.

In order to capture both control flow and data flow aspects of component behavior (as well as their mutual influence), finite state machines can be, and commonly are, equipped with variables. Variables can store the values of data parameters; they can influence control flow by means of guards, and the control flow can cause variable updates. Finite state machines with variables are often called *extended finite state machines* (EFSMs). Different dialects of EFSMs are successfully used in tools for model-based testing (such as ConformiQ Qtronic [Hui07], which produces high-quality test suites), web service composition [BPT10], model-based development [GHP02], and by software model checkers to formally verify properties of all program behaviors [JM09]. However, there is still no general dynamic analysis technique for inferring EFSM models with guards and assignments to variables. Existing techniques have restrictions: some limit the available operations on data to comparisons for equality, while others require significant manual effort (e.g., [AJUV14]), and/or rely on access to source code (e.g., [BB13]).

Contribution. In this paper, we present a framework for generating EFSM models of black-box components using dynamic analysis. Using active automata learning, we can infer concise models of components with, e.g., sequence numbers, time stamps, or other variables that are manipulated using moderately complex arithmetic operations and relations. We infer a class of behavioral models called *register automata* (RAs)–essentially, a restricted form of EFSMs. An RA has a finite control structure, extended with variables (registers) that can store values from a potentially infinite domain, and transition guards that compare data parameters to registers. Our contribution is an active automata learning algorithm for RAs. The algorithm is parameterized on a particular *theory*, i.e., a set of operations and tests on the data domain that can be used in guards. It is the first fully automated technique that can, in principle, be combined with any theory and generate full RA models with variables, guards, and operations.

Our framework can be used to generate informative models of the dynamic behavior of software components. As described above, such models are an essential basis for model-based test generation, service composition, protocol analysis, and several other activites in model-based development Our framework assumes knowledge about a component's static interface (e.g., the names of method calls and types of parameters), and a theory which captures the relations between data parameters that influence the component's control flow.

Technically, we generate RAs using active automata learning, following the general principles of the L^* algorithm [Ang87] for inferring finite state machines. We call our algorithm SL^* (for *Symbolic L^**), because it generalizes the L^* algorithm to the symbolic level.

The L^* algorithm is based on the *Nerode equivalence*. It views words as concatenations of prefixes and suffixes. Prefixes constitute equivalence classes of the Nerode equivalence, and suffixes distinguish words in different equivalence classes. The L^* algorithm maintains an increasing set of prefixes and an increasing set of suffixes until there is at least one prefix in each equivalence class of the language, and enough suffixes to distinguish all inequivalent prefixes. A final automaton that accepts the language is then constructed with each state corresponding to a Nerode-equivalence class.

As the basis for the SL^* algorithm, we define a novel symbolic version of the Nerode equivalence. In a generated register automaton, locations are still identified by prefixes, but distinguished by *symbolic decision trees*. A symbolic decision tree summarizes the relations between data parameter(s) in a given set of suffixes and a given prefix. It models how such relations affect whether continuations of the prefix should be accepted by the automaton or not. Constructing a symbolic decision tree for a prefix is called making a *tree query*.

Tree queries are used in place of membership queries: if two prefixes have isomorphic symbolic decision trees, they lead to the same location in the automaton. Thus, prefixes are compared at a symbolic level in our algorithm.

The concept of organizing prefixes and suffixes in a tree and use this tree a basis for constructing an EFSM that models relations between data parameters is entirely novel. We describe sufficient conditions for the properties that the symbolic constraints provided by a tree query in general must have to be usable in our learning model. We also show that, under these conditions, our framework induces a generalization of the classical Nerode equivalence and canonical automata construction to the symbolic setting.

We have implemented our algorithm and equipped it with a range of theories for different types of data values and relations between them. We consider integers with addition ($+$), equalities ($=$), and inequalities ($<$, $>$). We use our implementation to demonstrate the application of SL^* in a series of experiments, comprising time-stamps, sequence numbers, simple integer arithmetic, the connection establishment in TCP, and the priority queue from the Java Class Library. We discuss the results, including possible ways to optimize them in the future.

Related work. The problem of inferring behavioral models from implementations has been addressed in a number of different ways. Dynamic analysis approaches that combine automata learning techniques with methods for inferring constraints on data are the most closely related to our work. The pattern they follow is typically similar to CEGAR (counterexample-guided abstraction refinement): a sequence of models is refined in a process that is usually monotonic and converges to a fixpoint. All the approaches, however, suffer from limitations with respect to capturing the mutual influence of data flow and control flow on each other, and/or in what relations can be expressed between data parameters.

In white-box scenarios, access to the source code is presumed, so domain knowledge, manual abstractions, and/or symbolic execution can be used. White-box inference based on active automata learning (AAL) has been explored in several works. AAL has been combined with predicate abstraction [ACMN05] to infer interface specifications of Java classes, and with CEGAR [HJM05] to infer interface specifications as finite-state automata without data parameters. In [XSL+13], AAL is combined with support vector machines to infer constraints on data parameters; in [GRR12], AAL is combined with symbolic execution to recover guards from the analyzed system, producing DFA models where labels are guards over parameters of alphabet symbols. The authors of [BB13], also use AAL with symbolic execution, to infer transducers with memory. The memory of the transducers is, however, limited to a bounded-size window over past inputs.

In black-box scenarios, an early method for inferring EFSM-like models is [LMP08], where models are generated from execution traces by combining passive automata learning with the Daikon tool [EPG+07]. Since constraints on data parameters are only created for individual traces, there is no way to model the influence of data values on subsequent control flow.

Other approaches use AAL to infer data constraints from tests: In [AJUV14], a manually supplied abstraction on the data domain makes it possible to apply finite-state active automata learning techniques to the test cases. The approach has been successfully used in practical applications [ASV10, AdRP13], but a drawback is that a priori insight into the target component's behavior is required, making it not quite black-box. In [HSM11], automated (alphabet) refinement is used. Since the presented approach works at the level of concrete representative inputs, the resulting models have no symbolic interpretation but are rather minimal concrete representative systems. In [MM14], AAL is used to learn symbolic automata, and counterexamples used to refine transitions (representing equivalence classes in the language of the symbolic automata). The goal is to handle very large alphabets without having to store values in registers.

[IHS14] provides an overview of the extension of active automata learning from DFAs to register automata with tests for equality. For example, in [AHK+12, BHLM13] and our own previous work [HSJC12], EFSM models are inferred where equality tests are the only allowed operations on data. In our earlier work [BJR08], we generated a symbolic automaton from a finite automaton over a large data domain, potentially causing scalability problems for implementations. The authors of [BHLM13] infer EFSMs that they claim to be incomparable with register automata, and that can represent components where data parameters are 'globally fresh', i.e., never before seen or stored since the last reset of the component.

We have previously developed black-box techniques for generating (register automaton) models that combine control-flow and data, where the only operation on data is comparison for equality [HSJC12, HIS+12]. This approach does not need a priori supplied abstractions and can capture the interplay between control and data accurately. We have successfully applied it to generate models of container-like interfaces (such

as sets, stacks, queues, etc. [HIS⁺12], but the approach can not handle models with arbitrary relations or operations on the data values. In this paper, we generalize this approach and present - for the first time - a framework that can be parameterized by a number of different *theories* (operations on a data domain) to infer canonical models of components.

Outline. We introduce and illustrate the main ideas of our work, by means of a small example, in Section 2. In Section 3, we introduce theories and data languages, followed by register automata in Section 4. In Section 5, we define symbolic decision trees and canonical tree oracles. In Section 6, we introduce our version of the Nerode equivalence, and show how it can be used to construct canonical automata. In Section 7, we define canonical tree oracles for some common theories. We present the details of our learning algorithm in Section 8, and Section 9 presents the results of applying it to a series of small components. a small series of experiments. Here, we also briefly describe the implementation of a teacher for our learning framework. Conclusions then follow in Section 10.

2. Main Ideas

In this section, we introduce the main ideas behind our extension to the L^* algorithm. First, we briefly discuss the general problem of inferring an automaton model of a component from test cases. We describe the classic active automata learning approach to solving the problem. Then, we introduce a particular class of components that cannot be adequately modeled as finite automata but instead require more expressive formalisms, such as EFSMs. We use an example to illustrate particular issues that arise when modeling these components. Finally, we describe our active learning framework, and how it generalizes active automata learning to the symbolic setting.

We are interested in the problem of generating automata models of components from test cases. This problem can be formulated in terms of language inference. To generate finite-state models of components, we can use existing active automata learning algorithms for inference of regular languages. Active automata learning is often represented as a series of interactions between a Teacher and a Learner, where the Learner makes queries that the Teacher answers. In active automata learning (e.g. L^* [Ang87]), an automaton model of a black-box component is inferred by making a set of *membership queries*. By making a membership query, the Learner asks whether a particular sequence of input symbols should be accepted or rejected. It corresponds to executing a test case on the component. When the Learner has built a hypothesis automaton, it is submitted for an *equivalence query*, which consists in asking whether the hypothesis is equivalent to the language of the black-box component. If the reply is negative, a counterexample word is returned that highlights a difference between the target component and the hypothesis, prompting another round of membership queries. If the reply is positive, the algorithm terminates.

In order to build a DFA from the information obtained in membership queries, the L^* algorithm exploits the Nerode equivalence. Given a language and two words u and u', a *distinguishing suffix* for u and u' is a word v such that either uv or $u'v$ is in the language, but not both. Two words are *Nerode equivalent* if there is no distinguishing suffix for them. The Nerode equivalence can be used to build a minimal DFA, where each state corresponds to an equivalence class. The L^* algorithm exploits this fact by maintaining an increasing set U of words, called prefixes, that correspond to states, and an increasing set V of words, called suffixes, that are used to distinguish between states. The sets U and V are gradually increased while performing membership queries for words of form uv, with $u \in U$ and $v \in V$, until U covers all states, and each pair of states is distinguished by some suffix in V.

For languages where symbols carry data from a large or unbounded domain, the solution to our language inference problem is less straightforward, which the following example illustrates.

Example (Water pump). Consider a simple component: a water pump that controls the level of water in a tank. In order to lower or raise the water level, the pump accepts input of the form level(p), where p is a data parameter that can be instantiated with a real number to represent the desired water level. To ensure a minimum amount of water in the tank, the water level can always be raised, but it can never be lowered twice in a row. We define the water pump language as a set of sequences of commands, by stating that a sequence level(p_1) ... level(p_m) is in the language ("valid") if whenever $p_i > p_{i+1}$ for some $1 \leq i \leq m - 2$, then $p_{i+1} \leq p_{i+2}$. For example, the sequences level(1)level(2) and level(3)level(4)level(4) are both in the water pump language.   □

Prefix (of form $\mathsf{level}(x_1)\mathsf{level}(x_2)$)	Suffix (of form $\mathsf{level}(p)$)	Relations between data parameters	Response
$\mathsf{level}(3)\mathsf{level}(1)$	$\mathsf{level}(0)$	$p < x_2$	INVALID
	$\mathsf{level}(1)$	$p = x_2$	VALID
	$\mathsf{level}(2)$	$x_2 < p < x_1$	VALID
	$\mathsf{level}(3)$	$p = x_1$	VALID
	$\mathsf{level}(4)$	$x_1 < p$	VALID
$\mathsf{level}(5)\mathsf{level}(3)$	$\mathsf{level}(2)$	$p < x_2$	INVALID
	$\mathsf{level}(3)$	$p = x_2$	VALID
	$\mathsf{level}(4)$	$x_2 < p < x_1$	VALID
	$\mathsf{level}(5)$	$p = x_1$	VALID
	$\mathsf{level}(6)$	$x_1 < p$	VALID

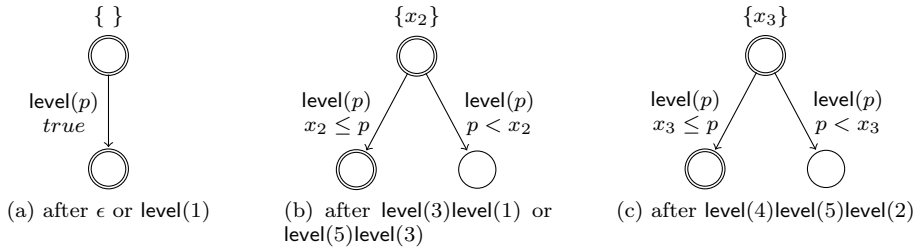Fig. 1. Results of executing test sequences on the water pump example

Fig. 2. Decision trees of depth 1 for the water pump example

The symbols in the water pump language (i.e., its commands) carry data values from an unbounded domain (real numbers). To build an automaton model that recognizes the water pump language, we cannot identify locations by the same standard Nerode equivalence used by L^* to infer regular languages. For example, the prefixes $\mathsf{level}(3)\mathsf{level}(1)$ and $\mathsf{level}(5)\mathsf{level}(3)$ would then not be equivalent. They are distinguished by the suffix $\mathsf{level}(2)$ (since $\mathsf{level}(3)\mathsf{level}(1)\mathsf{level}(2)$ is in the water pump language, but $\mathsf{level}(5)\mathsf{level}(3)\mathsf{level}(2)$ is not). In fact, it can be seen that almost all prefixes would be inequivalent under this approach. Looking at the definition of the water pump language, the natural solution to this problem is to redefine the Nerode equivalence so that it is based not on actual data values in suffixes, but rather on how the *relations* between data parameters in the suffix and the prefix determine whether a word is accepted or rejected.

Symbolic decision trees. We introduce *symbolic decision trees* (SDTs) as a means of symbolically representing relations between data values in a suffix and a prefix. An SDT is generated for a given prefix (with
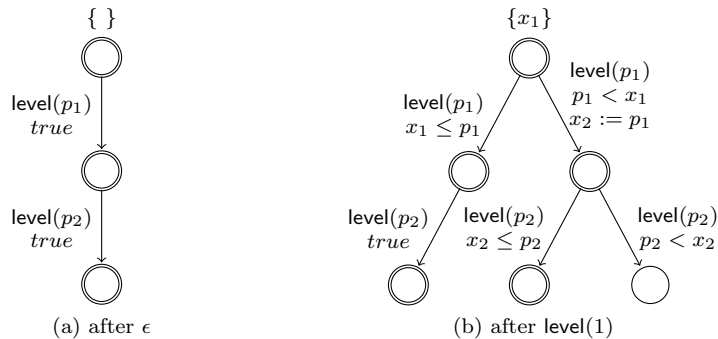

Fig. 3. Decision trees of depth 2 for the water pump example

concrete data values, e.g. level(1)) and a *symbolic suffix*, i.e., a sequence of symbols with uninstantiated data parameters (called *parameterized symbols*)[3]. It represents relations between different instantiations of the data parameters in the suffix, as well as relations between these instantiations and the data values in the prefix. The SDT also shows how these relations determine whether a particular word is in the target language or not. For example, in the water pump example, an SDT after a prefix level(3)level(1) expresses that words of the form level(3)level(1)level(p) are accepted whenever p at least as big as 1 and rejected otherwise. Figure 2(b) shows this particular SDT. We depict trees with the root location at the top and annotate locations with registers. By convention, a register named x_i holds the i-th data value of the corresponding prefix. Locations are either accepting (double circles) or rejecting. Transitions (branches) are labeled with guarded symbols. An SDT accepts a suffix if that suffix can reach an accepting location from the root location, by satisfying the appropriate guards. In Figure 2, Tree (a) accepts the suffix level(p) regardless of the concrete value of p, while Trees (b) and (c) require p to be at least as big as a data parameter in the prefix (the second one in (b) and the third one in (c)).

An SDT for a prefix and a symbolic suffix is generated by instantiating the parameters in the suffix according to a given *theory*, i.e., a set of operations and tests on the data domain. For example, in the water pump, the theory consists of the relations $<$ and $=$ over integers. The SDTs in Figure 2 are generated from the symbolic suffix level(p) after different prefixes. Tree (b) is generated after either of the prefixes level(3)level(1) or level(5)level(3), by checking language membership for the sequences in the corresponding part of the table in Figure 1 and summarizing the results:

- The first sequence in each part of the table (with suffixes level(0) and level(2), respectively) corresponds to the right branch in the tree. It is not in the water pump language ("invalid"), so the branch is rejecting. The relation $p < x_2$ is used as a guard for the branch.
- The remaining four suffixes in each part of the table correspond to the left branch in the tree. They are all in the water pump language ("valid"), so they can be grouped together to form an accepting branch. We use the guard $x_2 \leq p$ to collectively represent the relations in all of the sequences, and to distinguish this branch from the rejecting branch.

We use symbolic decision trees to separate different prefixes, i.e., in place of distinguishing suffixes in a symbolic version of the Nerode equivalence. We define this equivalence in Section 6. The symbolic Nerode equivalence forms the basis of our learning algorithm, SL^*. In the following, we describe the main features of SL^*.

Learning framework. We follow the pattern of classic active learning, i.e., our learning algorithm can be described as a series of interactions between a Teacher and a Learner. In our framework, the Teacher answers equivalence queries and tree queries. Tree queries are used in place of membership queries.

Tree query: The Learner submits a prefix and a symbolic suffix to the Teacher. The Teacher generates an SDT for the prefix and symbolic suffix, and returns it to the Learner. When learning a model of a target component, the SDT is generated by executing test cases on the component and observing its responses (whether each sequence is valid or not). The suffixes and corresponding responses are then organized to form an SDT.

Equivalence query: The Learner submits an automaton to the Teacher and asks whether it correctly models the target component. In theory, the Teacher replies OK if it does, and otherwise supplies a counterexample, i.e., a sequence that is valid but not accepted by the automaton, or vice versa. In practice, however, we do not have access to an implementation of a Teacher with complete information about the component's behavior. In these cases, equivalence queries can be approximated, e.g., by using conformance testing or monitoring of the component in order to find counterexamples.

The Learner infers a *register automaton* (see Section 4) that recognizes the target language, i.e., models the behavior of the target component. First, the Learner makes a set of tree queries to determine locations and transitions in the automaton. When certain stability criteria have been met, the Learner submits the automaton to the Teacher in an equivalence query. If the equivalence query is successful, the algorithm terminates; otherwise, a counterexample is returned. Counterexamples guide the Learner to make tree queries

---

[3] This is a simplified presentation: in actuality, we generate an SDT for a *set* of symbolic suffixes, each of which is a sequence of actions. We describe the generation of SDTs in more detail in Section 5.
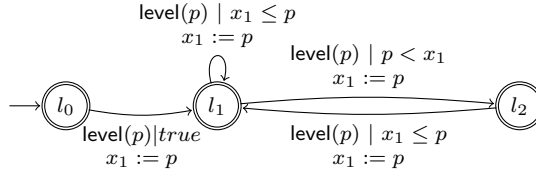
Fig. 4. A very simple water pump component

for more and/or longer suffixes after a given prefix. This will lead to refinements in the automaton: previously unified prefixes may be separated, new registers may be introduced, and transitions may be refined or new ones introduced.

Example (Water pump acceptor). Figure 4 shows an inferred model that accepts the water pump language. The automaton has three locations: l_0 (which is initial), l_1, and l_2, all of which are accepting; there is also a sink location not shown in the figure. There is one register (or variable) x_1 that stores the current water level. Transitions are denoted by arrows; each transition is labeled with a parameterized symbol level(p), a guard that compares p to the register x_1, and possibly an assignment to x_1. The register x_1 is initialized by the transition from l_0 to l_1. Symbols that do not match any transition lead to the sink location and are omitted in the figure. During operation, the pump moves between location l_1 (where the water level is above minimum) and location l_2 (where the water level has just been lowered and cannot immediately be lowered again). □

When SL^* infers a model of a component, such as the water pump, the Learner (at a very abstract level) builds a prefix-closed set of prefixes, i.e., words with concrete data values that reach locations in the inferred register automaton. To determine when prefixes should lead to the same location in the automaton, the learner makes tree queries and compares resulting SDTs to each other:

- Prefixes with isomorphic SDTs can be unified, and lead to the same location. SDTs can be made isomorphic by renaming their registers. For example, the trees in Figure 2 (b) and (c) are isomorphic if we rename x_3 to x_2. This in turn means that the corresponding prefixes level(5)level(3) and level(4)level(5)level(2) lead to the same location (cf. Figure 4).
- Prefixes with SDTs that cannot be made isomorphic (under any renaming) lead to different locations. For example, the SDTs in Figure 2 (a) and (b) cannot be made isomorphic, so the prefix level(5)level(3) reaches a different location than the prefixes ϵ and level(1).

The transitions of SDTs are used to create registers, guards, and assignments in the automaton. For example, in Figure 2, the second data value of level(5)level(3) must be stored in a register, to enable comparison with data parameters in a suffix.

3. Theories and Data Languages

In this and the following section, we introduce the central concepts of our framework: theories, data languages, and register automata.

Theories. Our framework is parameterized by a *theory*, i.e., a pair $\langle \mathcal{D}, \mathcal{R} \rangle$ where $\mathcal{D}$ is an unbounded domain of *data values*, and $\mathcal{R}$ is a set of *relations* on $\mathcal{D}$. The relations in $\mathcal{R}$ can have arbitrary arity. Some notable theories in this paper are:

equality: We consider equality over natural numbers, i.e., the theory $\langle \mathbb{N}, \{=\} \rangle$, but also equality over other infinite domains, such as session identifiers, or password/username strings.

inequality and equality: We consider theories with inequality and equality, over, e.g., real numbers or rational numbers. In Example 2, we use the theory $\langle \mathbb{R}, \{<, =\} \rangle$.

equality and increments: We consider sequence numbers that increase in steps of 1, and introduce a relation INC(m, n) to model the operation $m + 1 = n$ for two sequence numbers m, n. Then, we can use the theory $\langle \mathbb{N}, \{=, \text{INC}\} \rangle$ to model increasing sequence numbers.

equality and inequality with sums: We model the operation $m + n = o$ as a relation $\textsc{sum}(m, n, o)$. Then, we can use the theory $\langle \mathbb{R}^+, \{<, =, \textsc{sum}\} \rangle$ to model inequality, equality, and sums over positive real numbers.

The above theories can all be extended with constants (allowing, e.g., theories of sums with constants). In Section 7, we provide an in-depth discussion of how to implement tree queries for different theories; in Section 9, we show practical examples and benchmarks. In the following, we assume that some theory has been fixed.

Data languages. We assume a set Σ of *actions*, each with an arity that determines how many parameters it takes from the domain $\mathcal{D}$. In this paper, we assume that all actions have arity 1, but it is straightforward to extend our results to the case where actions have arbitrary arity.

A *data symbol* is a term of form $\alpha(d)$, where α is an action and $d \in \mathcal{D}$ is a data value. A *data word* is a sequence of data symbols. The concatenation of two data words w and w' is denoted ww'. In this context, we often refer to w as a *prefix* and w' as a *suffix*. For a data word $w = \alpha_1(d_1) \ldots \alpha_n(d_n)$, let $Acts(w)$ denote its sequence of actions $\alpha_1 \ldots \alpha_n$, and $Vals(w)$ its sequence of data values $d_1 \ldots d_n$. We often refer to a sequence of actions in Σ^* as a *symbolic suffix*. We write V for a set of symbolic suffixes, and $[\![V]\!]$ for the set of words v with $Acts(v) \in V$ Consequently, $u[\![V]\!]$ is the set of words of the form uv with $Acts(v) \in V$. If V is a set of symbolic suffixes, then $\alpha^{-1}V$ is the set of subsequences of suffixes in V such that $v = \alpha_1 \alpha_2 \ldots \alpha_n$ is in V iff $v' = \alpha_2 \ldots \alpha_n$ is in $\alpha^{-1}V$.

Definition 3.1 ($\mathcal{R}$-indistinguishable data words). Two data words $w = \alpha_1(d_1) \ldots \alpha_n(d_n)$ and $w' = \alpha_1(d'_1) \ldots \alpha_n(d'_n)$ are *$\mathcal{R}$-indistinguishable*, denoted $w \approx_{\mathcal{R}} w'$, if

- $Acts(w) = Acts(w')$, and
- $R(d_{i_1}, \ldots, d_{i_j})$ if and only if $R(d'_{i_1}, \ldots, d'_{i_j})$, whenever $R \in \mathcal{R}$ and $i_1, \ldots, i_j$ are indices between 1 and n.

□

Intuitively, w and w' are $\mathcal{R}$-indistinguishable if they have the same sequences of actions and cannot be distinguished by the relations in $\mathcal{R}$. We use $[w]_{\mathcal{R}}$ to denote the equivalence class of $\mathcal{R}$-indistinguishable data words that w belongs to. In the water pump example, where $\mathcal{R} = \{<, =\}$, the data words $\mathsf{level}(2)\mathsf{level}(1)$ and $\mathsf{level}(4)\mathsf{level}(0)$ are $\mathcal{R}$-indistinguishable: 2 is greater than (and not equal to) 1; 4 is greater than (and not equal to) 0.

A *data language* $\mathcal{L}$ is a set of data words that respects $\mathcal{R}$ in the sense that $w \approx_{\mathcal{R}} w'$ implies $w \in \mathcal{L} \leftrightarrow w' \in \mathcal{L}$. A data language can be represented as a mapping from the set of data words to $\{+, -\}$, where $+$ stands for ACCEPT and $-$ for REJECT.

4. Register Automata

We assume a set of *registers* (or variables), $x_1, x_2, \ldots$. A *parameterized symbol* is a term of form $\alpha(p)$, where α is an action and p a formal parameter. A *guard* is a conjunction of negated and unnegated relations (from $\mathcal{R}$) over the formal parameter p and registers. An *assignment* is a simple parallel update of registers with values from registers or the formal parameter p. An assignment which updates the registers $x_{i_1}, \ldots, x_{i_m}$ with values from the registers $x_{j_1}, \ldots, x_{j_n}$ or p can be represented as a mapping π from $\{x_{i_1}, \ldots, x_{i_m}\}$ to $\{x_{j_1}, \ldots, x_{j_n}\} \cup \{p\}$, meaning that the value of the register or formal parameter $\pi(x_{i_k})$ is assigned to the register x_{i_k} for $k = 1, \ldots, m$.

Definition 4.1 (Register automaton). A *register automaton* (RA) is a tuple $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$, where

- L is a finite set of *locations*, with $l_0 \in L$ as the *initial location*,
- λ maps each $l \in L$ to $\{+, -\}$,
- $\mathcal{X}$ maps each location $l \in L$ to a finite set $\mathcal{X}(l)$ of registers, and
- Γ is a finite set of *transitions*, each of form $\langle l, \alpha(p), g, \pi, l' \rangle$, where

  - $l \in L$ is a source location,
  - $l' \in L$ is a target location,
  - $\alpha(p)$ is a parameterized symbol,

– g is a guard over p and $\mathcal{X}(l)$, and

– π (the *assignment*) is a mapping from $\mathcal{X}(l')$ to $\mathcal{X}(l) \cup \{p\}$. $\qquad\qquad\square$

We require register automata to be completely specified in the sense that whenever there is an α-transition from some location $l \in L$, then the disjunction of the guards on the α-transitions from l is *true*.

Let us now describe the semantics of an RA. A *state* of an RA $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ is a pair $\langle l, \nu \rangle$ where $l \in L$ and ν is a valuation over $\mathcal{X}(l)$, i.e., a mapping from $\mathcal{X}(l)$ to $\mathcal{D}$. The state is *initial* if $l = l_0$. A *step* of $\mathcal{A}$, denoted $\langle l, \nu \rangle \xrightarrow{\alpha(d)} \langle l', \nu' \rangle$, transfers $\mathcal{A}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input of the data symbol $\alpha(d)$ if there is a transition $\langle l, \alpha(p), g, \pi, l' \rangle \in \Gamma$ with

- $\nu \models g[d/p]$, i.e., d satisfies the guard g under the valuation ν, and
- ν' is the updated valuation with $\nu'(x_i) = \nu(x_j)$ if $\pi(x_i) = x_j$, otherwise $\nu'(x_i) = d$ if $\pi(x_i) = p$.

A *run* of $\mathcal{A}$ over a data word $w = \alpha(d_1) \ldots \alpha(d_n)$ is a sequence of steps

$$\langle l_0, \nu_0 \rangle \xrightarrow{\alpha_1(d_1)} \langle l_1, \nu_1 \rangle \quad \ldots \quad \langle l_{n-1}, \nu_{n-1} \rangle \xrightarrow{\alpha_n(d_n)} \langle l_n, \nu_n \rangle$$

for some initial valuation ν_0. The run is *accepting* if $\lambda(l_n) = +$ and *rejecting* if $\lambda(l_n) = -$. The word w is *accepted (rejected) by $\mathcal{A}$ under ν_0* if $\mathcal{A}$ has an accepting (rejecting) run over w which starts in $\langle l_0, \nu_0 \rangle$.

Example 4.2. We describe a run of the water pump automaton in Figure 4 over the word $w = \mathsf{level}(2)\mathsf{level}(1)\mathsf{level}(4)$. The run is a sequence of three steps

$$\langle l_0, \nu_0 \rangle \xrightarrow{\mathsf{level}(2)} \langle l_1, \nu_1 \rangle \xrightarrow{\mathsf{level}(1)} \langle l_2, \nu_2 \rangle \xrightarrow{\mathsf{level}(4)} \langle l_1, \nu_3 \rangle$$

starting in the initial location l_0 and ending in location l_1. We show each step below.

Step 1: $\langle l_0, \nu_0 \rangle \xrightarrow{\mathsf{level}(2)} \langle l_1, \nu_1 \rangle$ transfers the automaton from the initial location to l_1 by following the transition $\langle l_0, \mathsf{level}(p), true, \pi, l_1 \rangle$ where $\pi(x_1) = p$. The parameter p represents the value 2, which satisfies the guard *true* under the empty valuation. ν_1 is the updated valuation where $\nu_1(x_1) = 2$.

Step 2: $\langle l_1, \nu_1 \rangle \xrightarrow{\mathsf{level}(1)} \langle l_2, \nu_2 \rangle$ transfers the automaton from location l_1 to l_2 by following the transition $\langle l_1, \mathsf{level}(p), p < x_1, \pi, l_2 \rangle$ where $\pi(x_1) = p$. The parameter p represents the value 1, which satisfies the guard $p < x_1$ under the valuation $\nu_1(x_1) = 2$. ν_2 is the updated valuation with $\nu_2(x_1) = 1$.

Step 3: $\langle l_2, \nu_2 \rangle \xrightarrow{\mathsf{level}(4)} \langle l_1, \nu_3 \rangle$ transfers the automaton from location l_2 to l_1 by following the transition $\langle l_2, \mathsf{level}(p), x_1 \le p, \pi, l_1 \rangle$ where $\pi(x_1) = p$. The parameter p represents the value 4, which satisfies the guard $x_1 \le p$ under the valuation $\nu_2(x_1) = 1$. ν_3 is the updated valuation with $\nu_3(x_1) = 4$.

The run is accepting, since it ends in the accepting location l_1. $\qquad\qquad\square$

An RA is *determinate* if there is no data word over which it has both accepting and rejecting runs. Note that an RA defined as above does not necessarily have runs over all data words, since a location need not have outgoing transitions for all actions in Σ. The RAs in this paper are determinate.

Definition 4.3 (Simple register automaton). A *simple register automaton* (SRA) is a determinate register automaton $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ with $\mathcal{X}(l_0) = \emptyset$, which has runs over all data words. $\qquad\qquad\square$

We use SRAs as acceptors for data languages. The language accepted by $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of data words that it accepts. Figure 4 shows an example of an SRA.

5. Symbolic Decision Trees and Tree Oracles

In this section, we define *symbolic decision trees* (SDTs), which are essentially RAs in the form of trees. An SDT for a particular set of data words describes which words are accepted and which are rejected, based on relations between data parameters. It can thus be said to describe (a fragment of) a data language at a symbolic level. We use SDTs to identify and separate locations in an inferred RA; two locations are separated if their SDTs for some particular set of data words cannot be made isomorphic by remapping registers.

In our learning framework, SDTs are constructed by a *tree oracle*. A special class of tree oracles, called

canonical tree oracles, satisfy properties needed by our SL^* algorithm. We give the properties here, and describe in more detail in Sections 7 how canonical tree oracles can be realized for different theories.

Definition 5.1 (Symbolic decision tree). A *symbolic decision tree* (SDT) is an RA $\mathcal{T} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ where L and Γ form a tree rooted at l_0.   □

In general, an SDT has registers in the initial location; we use $\mathcal{X}(\mathcal{T})$ to denote these registers $\mathcal{X}(l_0)$. Thus, an SDT has well-defined semantics only wrt. a given valuation of $\mathcal{X}(\mathcal{T})$. If l is a location of $\mathcal{T}$, let $\mathcal{T}[l]$ denote the subtree of $\mathcal{T}$ rooted at l. The transitions from a root location of an SDT are its *initial transitions*. The *initial α-transitions* of an SDT are the transitions for action α from the root location. They form a subset of the initial transitions for an SDT. Initial α-transitions are guarded by *initial α-guards*. For example, the SDT in Figure 2 (b) has two initial level-transitions with initial level-guards $x_2 \leq p$ and $p < x_2$.

Definition 5.2 (SDT isomorphism). Let $\mathcal{T} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ and $\mathcal{T}' = (L', l_0', \mathcal{X}', \Gamma', \lambda')$ be SDTs. $\mathcal{T}$ and $\mathcal{T}'$ are *isomorphic*, denoted $\mathcal{T} \simeq \mathcal{T}'$, if there is a bijection $\phi : L \mapsto L'$ between the locations of $\mathcal{T}$ and those of $\mathcal{T}'$ such that

- $\phi(l_0) = l_0'$,
- $\mathcal{X}(l) = \mathcal{X}'(\phi(l))$ for all $l \in L$,
- $\lambda(l) = \lambda'(\phi(l)$ for all $l \in L$, and
- $\langle l_1, \alpha(p), g, \pi, l_2 \rangle \in \Gamma$ precisely when $\langle \phi(l_1), \alpha(p), g', \pi, \phi(l_2) \rangle \in \Gamma'$ for some g' with $g' \leftrightarrow g$.   □

Let $\gamma : \mathcal{X}(\mathcal{T}) \mapsto \mathcal{X}(\mathcal{T}')$ be a bijection from the initial registers of $\mathcal{T}$ to the initial registers of $\mathcal{T}$. We say that $\mathcal{T}$ and $\mathcal{T}'$ are *isomorphic under γ*, denoted $\mathcal{T} \simeq_\gamma \mathcal{T}'$, if γ can be extended to a bijection from all registers of $\mathcal{T}$ to all registers of $\mathcal{T}'$, under which $\mathcal{T} \simeq \mathcal{T}'$.

Example 5.3. Let $\mathcal{T}$ denote the SDT in Figure 2 (b) and let $\mathcal{T}'$ denote the SDT in Figure 2 (c). For $\mathcal{T}, \mathcal{T}'$, let l_0, l_0' denote the root locations, l_1, l_1' the left-hand leaves, and l_2, l_2' the right-hand leaves. Let γ be the bijection from the registers of $\mathcal{T}$ to the registers of $\mathcal{T}'$ defined by $\gamma(x_2) = x_3$. $\mathcal{T}$ and $\mathcal{T}'$ are isomorphic under γ if there is a bijection from the locations of $\mathcal{T}$ to the locations of $\mathcal{T}'$, with the properties in Def. 5.2. In the following, we show that the bijection ϕ, defined as $\phi(l_0) = l_0'$, $\phi(l_l) = l_l'$, and $\phi(l_2) = l_2'$, has these properties:

- $\phi(l_0) = l_0'$.
- Since $\mathcal{X}(l_0) = \{x_2\}$, $\mathcal{X}(\phi(l_0)) = \{x_3\}$, and $\gamma(x_2) = x_3$, we get that $\mathcal{X}(l_0) = \gamma(\mathcal{X}'(\phi(l_0)))$. For all other locations, $\mathcal{X}(l_1)$, $\mathcal{X}(l_2)$, $\gamma(\mathcal{X}'(\phi(l_1)))$, and $\gamma(\mathcal{X}'(\phi(l_2)))$ are empty sets, i.e., there are only registers in the initial locations. Thus, $\mathcal{X}(l) = \gamma(\mathcal{X}'(\phi(l)))$ for all locations in $\mathcal{T}$.
- $\lambda(l) = \lambda'(\phi(l))$ for all locations in $\mathcal{T}$, i.e., l_0, l_0' are both accepting, l_1, l_1' are also both accepting, and l_2, l_2' are both rejecting.
- Let g', h' be guards with $g' = x_3 \leq p$ and $h' = p < x_3$. Let g, h be guards with $g = x_2 \leq p$ and $h = p < x_2$. Since $\gamma(x_2) = x_3$, we get that $g \leftrightarrow g'$ and $h \leftrightarrow h'$ under γ. Then the transition $\langle l_0, \mathsf{level}(p), g, \pi, l_1 \rangle$ is in Γ precisely when $\langle l_0', \mathsf{level}(p), g', \pi, l_1' \rangle$ is in Γ', and the transition $\langle l_0, \mathsf{level}(p), h, \pi, l_2 \rangle$ is in Γ precisely when $\langle l_0', \mathsf{level}(p), h', \pi, l_2' \rangle$ is in Γ'.   □

We now describe how a tree oracle can be used to obtain symbolic decision trees. First we need some preliminary definitions.

Let u be a data word with $Vals(u) = d_1, \ldots, d_k$. The *potential of u*, denoted $pot(u)$, is the set of indices i among $1, \ldots, k$ for which there is no j with $i < j \leq k$ such that $d_j = d_i$. For example, the potential of the data word $\mathsf{level}(4)\mathsf{level}(9)\mathsf{level}(4)$ is $\{2, 3\}$. Intuitively, $pot(u)$ contains the indices i for which the variable x_i can occur in a guard that is constructed by a tree oracle. Potentials provide a consistent way to refer to parameters in a data words, even when some data value occurs more than once. We make the design decision to always use the last occurrence of any duplicate data value, in order to solve the problem of which variable (x_j or x_i) to use in guards whenever $d_j = d_i$. For example, when constructing an SDT for the prefix $u = \mathsf{level}(4)\mathsf{level}(9)\mathsf{level}(4)$, the variables x_2 and x_3 can be used in guards, but not x_1.

Let ν_u be the valuation of the set of registers $\{x_1, \ldots, x_k\}$, defined by $\nu_u(x_i) = d_i$ (recall that $Vals(u) = d_1, \ldots, d_k$). A *$u$-guard* is a guard g over the formal parameter p and the variables $\{x_i \ : \ i \in pot(u)\}$ such that there is some data value d with $\nu_u \models g[d/p]$, i.e., d satisfies g after the sequence u.

We require that for each data word u and each u-guard g, there is a data value d_u^g in $\mathcal{D}$ that we can use as a representative for all data values in $\mathcal{D}$ that satisfy the guard g. We call such a data value *representative*. A representative data value obeys the following rules:

- $\nu_u \models g[\mathtt{d}_u^g/p]$ (i.e., $\mathtt{d}_u^g$ satisfies p after u), and such that
- whenever g' is a stronger u-guard satisfied by $\mathtt{d}_u^g$ (i.e., $\nu_u \models g'[\mathtt{d}_u^g/p]$) then $\mathtt{d}_u^{g'} = \mathtt{d}_u^g$.

The representative data value for a guard g is determined the first time it is needed. Whenever g is replaced by a stronger guard g', we define $\mathtt{d}_u^{g'}$ to be $\mathtt{d}_u^g$ if possible without violating any of the rules for representative data values.

Example 5.4. Consider again the water pump example, and the word $u = \mathsf{level}(4)\mathsf{level}(9)\mathsf{level}(4)$. The valuation ν_u of the set $\{x_1, x_2, x_3\}$ of registers is defined by $\nu_u(x_1) = 4$, $\nu_u(x_2) = 9$ and $\nu_u(x_3) = 4$. The potential of u is $\{2, 3\}$. The guard $x_2 < p < x_3$ is not a u-guard, since it cannot be satisfied by any data value under the valuation ν_u. The guard $g = p < x_2$ is a u-guard, since it can be satisfied by a data value under the valuation ν_u and 2 is in the potential of u. The representative value $\mathtt{d}_u^g$ for g can be any value that is less than 9 (since g is $g = p < x_2$ and ν_u maps x_2 to 9). For example, $\mathtt{d}_u^g$ can be 6. Representative values are used to extend the prefix by some symbol $\mathsf{level}(d)$ where d satisfies the guard g. The last requirement for representative data values implies that we can still use the same symbol when g is refined, if possible. For instance, if g is strengthened to g', defined as $x_3 < p < x_2$, then we keep $\mathtt{d}_u^{g'}$ as 6. $\qquad\square$

Definition 5.5 ((u, V)-tree). For a data word u with $Vals(u) = d_1, \ldots, d_k$, and a set V of symbolic suffixes, a (u, V)-tree is an SDT $\mathcal{T}$, which has runs over all data words $v \in \llbracket V \rrbracket$, and which satisfies the technical restriction that whenever $\langle l, \alpha(p), g, \pi, l' \rangle$ is the jth transition on some path from l_0, then for each $x_i \in \mathcal{X}(l')$ we have either

(i) $i < k + j$ and $\pi(x_i) = x_i$, or
(ii) $i = k + j$ and $\pi(x_i) = p$ (recall that k is the length of u). $\qquad\square$

Intuitively, a (u, V)-tree is an SDT for a set V of symbolic suffixes and where, in any run over a data word $v \in V$, the register x_i may contain only the value of the ith data value in uv. This restriction is introduced to make it easier to compare (u, V)-trees.

Definition 5.6 (Tree oracle). Let $\langle \mathcal{D}, \mathcal{R} \rangle$ be a theory. A *tree oracle* for $\langle \mathcal{D}, \mathcal{R} \rangle$ is a function $\mathcal{O}$, which for a language $\mathcal{L}$, data word u and a set V of symbolic suffixes returns a (u, V)-tree $\mathcal{O}(\mathcal{L}, u, V)$, such that for any data word $uv \in u\llbracket V \rrbracket$ it holds that v is accepted by $\mathcal{O}(\mathcal{L}, u, V)$ under ν_u iff $uv \in \mathcal{L}$, and rejected iff $uv \notin \mathcal{L}$. $\qquad\square$

In the following, we will mostly omit mentioning the theory when talking about tree oracles, since it is mostly given by context. We will mostly use the notation $\mathcal{O}_\mathcal{L}(u, V)$ for $\mathcal{O}(\mathcal{L}, u, V)$.

Definition 5.7 (Canonical tree oracle). Let $\mathcal{L}$ be a data language. The tree oracle $\mathcal{O}$ is *canonical* if it satisfies the following conditions for any language $\mathcal{L}$, prefix u, and set of symbolic suffixes V:

1. for each initial transition $\langle l_0, \alpha(p), g, \pi, l \rangle$ of $\mathcal{O}_\mathcal{L}(u, V)$ we have $\mathcal{O}_\mathcal{L}(u, V)[l] \simeq \mathcal{O}_\mathcal{L}(u\alpha(\mathtt{d}_u^g), \alpha^{-1}V)$,
2. whenever $V \subseteq V'$, then $\mathcal{O}_\mathcal{L}(u, V') \simeq_\gamma \mathcal{O}_\mathcal{L}(u', V')$ implies $\mathcal{O}_\mathcal{L}(u, V) \simeq_\gamma \mathcal{O}_\mathcal{L}(u', V)$ for all u, u' and γ, and
3. whenever $V \subseteq V'$, then

   (i) for each initial α-guard g of $\mathcal{O}_\mathcal{L}(u, V)$ there is an initial α-guard h of $\mathcal{O}_\mathcal{L}(u, V')$ with $\nu_u \models h[\mathtt{d}_u^g/p]$ and $\nu_u \models h \longrightarrow g$, and
   (ii) $\mathcal{X}(\mathcal{O}_\mathcal{L}(u, V)) \subseteq \mathcal{X}(\mathcal{O}_\mathcal{L}(u, V'))$. $\qquad\square$

Intuitively, the first condition states that the trees returned by the tree oracle are constructed recursively, using the definition of representative data values. The second condition states that extending V can only preserve or introduce inequivalence between trees of different prefixes. The third condition states that extending V can only refine the initial transitions and increase the set of registers. Together, these conditions ensure monotonicity when V is extended.

Example 5.8. Figure 2 (a) shows a (u, V)-tree for $u = \mathsf{level}(1)$ and $V = \{\mathsf{level}\}$. The SDT has one initial transition, $\langle l_0, \mathsf{level}(p), true, \pi, l_1 \rangle$. It is refined by either of the SDTs in Figure 3. The SDT in Figure 3 (a) has only one initial transition, guarded by $true$, which refines the initial transition in Figure 2 (a). The SDT in Figure 3 (b) has two initial transitions with the guards $x_1 \le p_1$ and $p_1 < x_1$, respectively. They also refine the initial transition in Figure 2 (a).

The reason for the refinements in Figure 3 (b) is that when we extend the tree to length 2, we test words of

the form level(4)level(d_1)level(d_2), using different data values for d_1 and d_2. Since, e.g. level(4)level(3)level(2) is rejected, but, e.g., level(4)level(5)level(2) is accepted, the initial guard *true* is no longer valid and must be refined into $x_1 \leq p_1$ and $p_1 < x_1$. The assignment $x_2 := p_1$ is induced by the two guards in the lower right-hand side of the tree, where acceptance (the middle leaf) and rejection (the rightmost leaf) depend on the relation between p_2 and p_1.   □

6. Nerode Equivalence and Automata

In this section, we show how a canonical tree oracle can be used to define a generalization of the classical Nerode equivalence to the symbolic setting, and how such an equivalence induces a definition of canonical register automata. In our version of the Nerode equivalence, we require that each equivalence class has at least one representative data word that can emulate the behavior of any other data word in the class. To ensure that such a word exists, we make a technical requirement on theories we consider in this paper.

Definition 6.1 (k-extendable). Assume a theory $\langle \mathcal{D}, \mathcal{R} \rangle$. Let $k \geq 0$. A data word u is *k-extendable* (wrp. to $\langle \mathcal{D}, \mathcal{R} \rangle$) if either

- $k = 0$, or
- for any data word u' with $u' \approx_{\mathcal{R}} u$ and symbol $\alpha(d')$, there is a symbol $\alpha(d)$ such that $u\alpha(d) \approx_{\mathcal{R}} u'\alpha(d')$, and such that $u\alpha(d)$ is $(k-1)$-extendable.

The set of *∞-extendable* data words is the largest set W of data words such that if $u \in W$ is in the set, then for any data word u' with $u' \approx_{\mathcal{R}} u$ and symbol $\alpha(d')$, there is a symbol $\alpha(d)$ such that $u\alpha(d) \approx_{\mathcal{R}} u'\alpha(d')$ and $u\alpha(d) \in W$.   □

If u is k-extendable, this implies that for any u' with $u' \approx_{\mathcal{R}} u$ and any suffix v' of length k, there is a suffix v such that $u'v' \approx_{\mathcal{R}} uv$. As an example, the word $\alpha(1)\alpha(2)$ in the theory $\langle \mathbb{N}, \{<, =\} \rangle$ of equality and inequality over natural numbers is 0-extendable but not 1-extendable, since it cannot be extended by any suffix to form a word that is equivalent to $\alpha(1)\alpha(3)\alpha(2)$. On the other hand, $\alpha(2)\alpha(4)$ is 1-extendable (but not 2-extendable).

A theory is said to be *strongly extendable* if all data words are ∞-extendable. A theory is said to be *weakly extendable* if for all k and all u there is a u' with $u' \approx_{\mathcal{R}} u$ which is k-extendable. It can be seen that a theory is weakly extendable if the empty word is k-extendable for all k. The theory $\langle \mathbb{N}, \{<, =\} \rangle$ of equality and inequality over natural numbers is weakly extendable. For example, if $k = 3$, $\alpha(2)\alpha(6)$ is a k-extendable word that is $\mathcal{R}$-equivalent to $\alpha(2)\alpha(4)$. The theory $\langle \mathbb{R}, \{<, =\} \rangle$ of equality and inequality over real numbers is strongly extendable. In this theory, any word, such as $\alpha(2)\alpha(4)$, is ∞-extendable since there is an infinite number of real numbers between 2 and 4.

Let us now define our generalization of the Nerode equivalence. For this exposition, we assume that the theory is strongly extendable. At the end of this section, we indicate how to modify the results to the case that the theory is weakly extendable.

For data words u and u' with $Vals(u) = d_1, \ldots, d_k$ and $Vals(u') = d'_1, \ldots, d'_{k'}$,

Definition 6.2 (Nerode equivalence). Assume a strongly extendable theory. Let u and u' be data words with $Vals(u) = d_1, \ldots, d_k$ and $Vals(u') = d'_1, \ldots, d'_{k'}$, and let $\mathcal{O}$ be a canonical tree oracle for our theory. Let $u \equiv_{\mathcal{O}_{\mathcal{L}}}^{\gamma} u'$ denote that there is a bijection γ between a subset of $x_1, \ldots, x_k$ and a subset of $x_1, \ldots, x_{k'}$ such that $\mathcal{O}_{\mathcal{L}}(u, V) \simeq_{\gamma} \mathcal{O}_{\mathcal{L}}(u', V)$ for all sets V of symbolic suffixes. The words u and u' are *$\mathcal{L}$-equivalent*, denoted $u \equiv_{\mathcal{O}_{\mathcal{L}}} u'$, if $u \equiv_{\mathcal{O}_{\mathcal{L}}}^{\gamma} u'$ for some bijection γ between a subset of $x_1, \ldots, x_k$ and a subset of $x_1, \ldots, x_{k'}$.   □

In the remainder of this section, we assume that u and u' are data words with $Vals(u) = d_1, \ldots, d_k$ and $Vals(u') = d'_1, \ldots, d'_{k'}$, and that $\mathcal{O}$ is a canonical tree oracle. For any γ between a subset of $x_1, \ldots, x_k$ and a subset of $x_1, \ldots, x_{k'}$, let $\hat{\gamma}$ be the extension of γ obtained by extending the domain of γ with x_{k+1} and defining $\hat{\gamma}(x_{k+1}) = x_{k'+1}$.

Define a data language $\mathcal{L}$ to be *regular* with respect to $\mathcal{O}$ if $\equiv_{\mathcal{O}_{\mathcal{L}}}$ has finite index. Note that the regularity of $\mathcal{L}$ is relative to the particular tree oracle $\mathcal{O}$ that is used. We can now state and prove an analogue of the classical Myhill-Nerode theorem. This theorem provides the basis for convergence of our SL^* algorithm.

Theorem 6.3 (Myhill-Nerode). Assume a strongly extendable theory. Let $\mathcal{L}$ be a data language, and let $\mathcal{O}$ be a canonical tree oracle for the theory. If $\mathcal{L}$ is regular wrp. to $\mathcal{O}$, then there is an SRA that accepts $\mathcal{L}$.

Proof. Let k be big enough so that for any u, u' we have $\mathcal{O}_{\mathcal{L}}(u, \Sigma^k) \simeq_{\gamma} \mathcal{O}_{\mathcal{L}}(u', \Sigma^k)$ if and only if $u \equiv_{\mathcal{O}_{\mathcal{L}}}^{\gamma} u'$ and so that the initial guards of $\mathcal{O}_{\mathcal{L}}(u, \Sigma^k)$ are the same as the initial guards of $\mathcal{O}_{\mathcal{L}}(u, V')$ for any V' with $\Sigma^k \subseteq V'$, and so that $\mathcal{X}(\mathcal{O}_{\mathcal{L}}(u, \Sigma^k))$ includes $\mathcal{X}(\mathcal{O}_{\mathcal{L}}(u, V'))$ for any V' with $\Sigma^k \subseteq V'$. Such a k must exist by the regularity of $\mathcal{L}$, and by the observation that, using Condition 3 of 5.7, initial guards can be refined only finitely many times.

In the proof, we will first construct an SRA $\mathcal{A}$, and thereafter establish that $\mathcal{A}$ accepts $\mathcal{L}$. First, we define the set L of locations with transitions between them, using a spanning tree construction. Then, we define λ for all locations. Locations in L can be either marked or unmarked. Initially, L contains only the single unmarked location l_{ϵ}, which is also the initial location. The set L is then extended and modified as follows: The set L is then extended and modified by the following procedure.

- Repeatedly choose an arbitrary unmarked location $l_u \in L$ and do the following:

  1. for each initial transition $\langle l_0, \alpha(p), g, \pi, l \rangle$ of $\mathcal{O}_{\mathcal{L}}(u, \Sigma^{k+1})$,

     – if there is no $l_{u'}$ in L with $u\alpha(\mathsf{d}_u^g) \equiv_{\mathcal{O}_{\mathcal{L}}} u'$, then add $l_{u\alpha(\mathsf{d}_u^g)}$ (unmarked) to L, and add $\langle l_u, \alpha(p), g, \pi, l_{u\alpha(\mathsf{d}_u^g)} \rangle$ to Γ,

     – if there is already some $l_{u'}$ in L with $u\alpha(\mathsf{d}_u^g) \equiv_{\mathcal{O}_{\mathcal{L}}}^{\gamma} u'$, then add $\langle l_u, \alpha(p), g, (\pi \circ \gamma^{-1}), l_{u'} \rangle$ to Γ,

  2. mark l_u,

  until all locations in L are marked. The set L is now taken as the set of locations of $\mathcal{A}$.

The procedure is guaranteed to terminate since there is a finite number of equivalence classes of $\equiv_{\mathcal{O}_{\mathcal{L}}}$. Note that in general, L may contain fewer locations than there are equivalence classes of $\equiv_{\mathcal{O}_{\mathcal{L}}}$, since not all equivalence classes need to have their own location. This can happen if some equivalence classes are "covered" by other ones. For instance, using the theory of equality, assume that $\mathcal{L}$ accepts only words of form $\alpha(d_1)\alpha(d_2)\alpha(d_3)\alpha(d_4)$ with $d_1 = d_3$ and $d_2 = d_4$. Then the equivalence class $u = \alpha(1)\alpha(2)$ is sufficient to cover the behavior for all prefix of length 2. In particular, u covers the behavior of the prefix $u(1)u(1)$, which is not equivalent to u.

To complete the definition of $\mathcal{A}$, for each $l_u \in L$ define $\mathcal{X}(l_u) = \mathcal{X}(\mathcal{O}_{\mathcal{L}}(u, \Sigma^k))$ and define $\lambda(l_u) = +$ if and only if $u \in \mathcal{L}$.

We must now check that $\mathcal{A}$ indeed accepts $\mathcal{L}$. Consider $\mathcal{O}_{\mathcal{L}}(\epsilon, \Sigma^{(n+k)})$ and an arbitrary word $w = \alpha(d_1) \ldots \alpha(d_n)$. Since $\mathcal{O}$ is a tree oracle, $\mathcal{O}_{\mathcal{L}}(\epsilon, \Sigma^{(n+k)})$ classifies w correctly. We will establish a correspondence between runs over w in $\mathcal{A}$ and runs over w in $\mathcal{O}_{\mathcal{L}}(\epsilon, \Sigma^{(n+k)})$, showing that also $\mathcal{A}$ classifies w correctly. To establish this correspondence, we first generate a representative data word u'_m for each location m of $\mathcal{O}_{\mathcal{L}}(\epsilon, \Sigma^{(n+k)})$, as follows. Let u'_{m_0} be the empty word ϵ, for the initial location m_0. If u'_m has been generated, and $\langle m, \alpha(p), g, \pi, m_1 \rangle$ is a transition, then let u'_{m_1} be $u'_m \alpha(\mathsf{d}_{u'_m}^g)$.

We thereafter establish that there is a run of $\mathcal{A}$ over w of form

$$\langle l_{\epsilon}, \nu_0 \rangle \xrightarrow{\alpha_1(d_1)} \langle l_{u_1}, \nu_1 \rangle \quad \ldots \quad \langle l_{u_{n-1}}, \nu_{n-1} \rangle \xrightarrow{\alpha_n(d_n)} \langle l_{u_n}, \nu_n \rangle$$

precisely if there is a run of $\mathcal{O}_{\mathcal{L}}(\epsilon, \Sigma^{(n+k)})$ over w of form

$$\langle m_0, \mu_0 \rangle \xrightarrow{\alpha_1(d_1)} \langle m_1, \mu_1 \rangle \quad \ldots \quad \langle m_{n-1}, \mu_{n-1} \rangle \xrightarrow{\alpha_n(d_n)} \langle m_n, \mu_n \rangle$$

where for each $i = 0, \ldots, n$ there is a bijection γ_i from the registers of l_i to the registers of m_i, such that $u_i \equiv_{\mathcal{O}_{\mathcal{L}}}^{\gamma_i} u'_{m_i}$ and $\nu_i = (\mu_i \circ \gamma_i)$.

We establish the correspondence between the above two runs by induction over the position $i = 0, \ldots, n$ in the run. For the base case $i = 0$, we trivially establish $u_0 \equiv_{\mathcal{O}_{\mathcal{L}}}^{\gamma_i} u'_{m_0}$ since $u_0 = u'_{m_0} = \epsilon$, and $\nu_0 = (\mu_0 \circ \gamma_0)$ since there are no registers in the initial locations. For the inductive step, assume that the correspondence holds for some i, i.e., $u_i \equiv_{\mathcal{O}_{\mathcal{L}}}^{\gamma_i} u'_{m_i}$ and $\nu_i = (\mu_i \circ \gamma_i)$. By $u_i \equiv_{\mathcal{O}_{\mathcal{L}}}^{\gamma_i} u'_{m_i}$ and the construction of $\mathcal{A}$, there is a transition $\langle l_{u_i}, \alpha(p), g, \pi, l_{u_{i+1}} \rangle$ iff there is a transition $\langle m_i, \alpha(p), \gamma_i(g), \pi', m_{i+1} \rangle$ with $\pi' = \gamma_i \circ \pi \circ \hat{\gamma_i}^{-1}$, i.e., π' maps registers in $m_i + 1$ to registers in m_i and the parameter p. Using the valuation $\nu_i = (\mu_i \circ \gamma_i)$, this implies that $\nu_i \models g[d/p]$ iff $\mu_i \models \gamma_i(g)[d/p]$.

We must also show that the correspondence holds for the updated valuations ν_{i+1} and μ_{i+1}. By the construction of $\mathcal{A}$, there are two cases: either u_{i+1} is the representative data word $u_i \alpha(\mathsf{d}_{u_i}^g)$ or u_{i+1} is equivalent to $u_i \alpha(\mathsf{d}_{u_i}^g)$ under some remapping ρ between registers.

- If $u_{i+1} = u_i \alpha(\mathtt{d}_{u_i}^g)$, we have $u_{i+1} \equiv_{\mathcal{O}_\mathcal{L}}^{\hat{\gamma}_i} u'_{m_{i+1}}$. The valuation ν_{i+1} is $\nu_{i+1} = \mu_{i+1} \circ \hat{\gamma}_i$. Since $\hat{\gamma}_i$ maps registers in $l_{u_{i+1}}$ to registers in m_{i+1}, we see that $\mu_{i+1} \circ \hat{\gamma}_i$ maps registers in $l_{u_{i+1}}$ to data values, i.e., we can take $\hat{\gamma}_i$ as γ_{i+1}.

- Otherwise, $u_i \alpha(\mathtt{d}_{u_i}^g) \equiv_{\mathcal{O}_\mathcal{L}}^{\rho} u_{i+1}$ for some bijection ρ between registers. This implies that $u_{i+1} \equiv_{\mathcal{O}_\mathcal{L}}^{\hat{\gamma}_i \circ \rho^{-1}} u'_{m_{i+1}}$. The valuation ν_{i+1} is $\nu_{i+1} = \mu_{i+1} \circ (\hat{\gamma}_i \circ \rho^{-1})$. Since $\hat{\gamma}_i \circ \rho^{-1}$ maps registers in $l_{u_{i+1}}$ to registers in m_{i+1}, we see that $\mu_{i+1} \circ (\hat{\gamma}_i \circ \rho^{-1})$ maps registers in $l_{u_{i+1}}$ to data values, i.e., we can take $\hat{\gamma}_i \circ \rho^{-1}$ as γ_{i+1}.

From the above correspondence between runs, we use $u_n \equiv_{\mathcal{O}_\mathcal{L}}^{\gamma_n} u'_{m_n}$ to conclude that $\mathcal{A}$ accepts w iff $\mathcal{O}_\mathcal{L}(\epsilon, \Sigma^{(n+k)})$ does. We have thus established that $\mathcal{A}$ accepts $\mathcal{L}$, and the proof is concluded. $\qquad\square$

Extension to weakly extendable theories. We here briefly outline how the definition of Nerode equivalence and Theorem 6.3 can be adapted to the setting of a weakly extendable theory. In a weakly extendable theory, we let $u \equiv_{\mathcal{O}_\mathcal{L}}^{\gamma} u'$ denote that there is a bijection γ between a subset of $x_1, \ldots, x_k$ and a subset of $x_1, \ldots, x_{k'}$ such that $\mathcal{O}_\mathcal{L}(u, V) \simeq_\gamma \mathcal{O}_\mathcal{L}(u', V)$ for all sets V of symbolic suffixes of length $\leq k$, where k is the largest integer such that u and u' are both k-extendable. The consecutive definitions are as before. In the proof of Theorem 6.3, care must be taken when constructing the locations of $\mathcal{A}$ from data words: these must be k-extendable for a sufficiently big k.

7. Canonical Tree Oracles for Some Common Theories

In this section, we describe how canonical tree oracles can be realized for some commonly occurring theories: the theory of equalities, the theory of equality and inequality over rational (or real) numbers, and the theory of equality and inequality over integers.

Throughout this section, we let u be a data word with $Vals(u) = d_1, \ldots, d_k$, and V be a set of symbolic suffixes. For an action α, we will use V_α for $\alpha^{-1}V$ (i.e., the set of sequenes $\alpha_2 \ldots \alpha_n$ with $\alpha_1 \alpha_2 \ldots \alpha_n \in V$).

In order to simplify the notation involved in defining (u, V)-trees, we will in this section only define (u, V)-trees for which some elements, as defined in Definition 5.5, are redundant and can be inferred from the context. We will therefore omit these elements in definitions of (u, V)-trees. The redundant elements are the following.

- The set $\mathcal{X}(l)$ of registers in a location l can be directly obtained as the set of registers that occur in $\mathcal{T}[l]$, i.e., the subtree of $\mathcal{T}$ rooted at l.
- The assignment π in a transition $\langle l, \alpha(p), g, \pi, l' \rangle$ of a (u, V)-tree maps each $x_i \in \mathcal{X}(l')$ to an element which is uniquely defined by i and l' (namely, by Definition 5.5, if $\langle l, \alpha(p), g, \pi, l' \rangle$ is the jth transition on some path from l_0, then $\pi(x_i) = x_i$ whenever $i < k + j$, and $\pi(x_i) = p$ whenever $i = k + j$).

Therefore, we will in what follows define (u, V)-trees as tuples of form $\mathcal{T} = (L, l_0, \Gamma, \lambda)$, in which each transition has the form $\langle l, \alpha(p), g, l' \rangle$ with the assignment removed.

7.1. Canonical Tree Oracle for the Theory of Equalities

In this section, we present a realization of a canonical tree oracle for the theory of equalities over an infinite domain, such as the set of natural numbers. Our tree oracle $\mathcal{O}$ will generate (u, V)-trees in which each guard is either an equality of form $p = x_i$ or a conjunction of negated equalities of form $\bigwedge_{j \in I} p \neq x_j$ for some set I of indexes. For guards of the above form, we choose representative data values as follows:

- when g is of form $p = x_i$, we let $\mathtt{d}_u^g$ be d_i, and
- when g is of form $\bigwedge_{j \in I} p \neq x_j$, we let $\mathtt{d}_u^g$ be some data value not in $Vals(u)$ which is uniquely determined by u (this ensures that we can use the same data value as $\mathtt{d}_u^{g'}$ for any u-guard g' of form $\bigwedge_{j \in I'} p \neq x_j$). In the following, we let $\mathtt{d}_u^0$ denote this chosen data value.

Let us more precisely describe the trees constructed by our tree oracle.

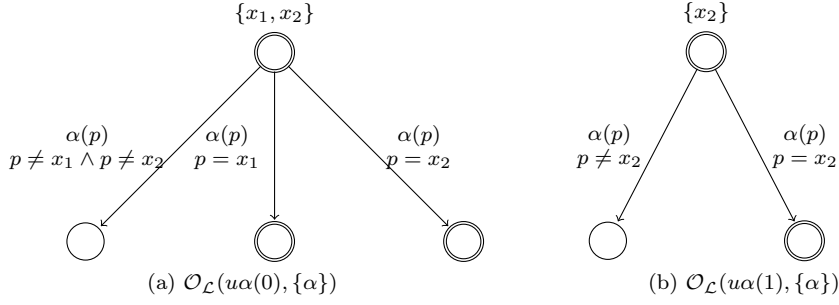**Definition 7.1 (Equality tree).** An *equality tree* for (u, V) is a (u, V)-tree $\mathcal{T}$ such that

Fig. 5. Two canonical subtrees

- for each action α, there is a set $I \subseteq pot(u)$ of indices, such that the initial α-guards consist of the equalities of form $p = x_i$ for $i \in I$, and one conjunction of inequalities of form $\bigwedge_{j \in I} p \neq x_j$,

- for each initial transition $\langle l_0, \alpha(p), g, l \rangle$ of $\mathcal{T}$, the tree $\mathcal{T}[l]$ is an equality tree for $(u\alpha(\mathtt{d}_u^g), \alpha^{-1}V)$.   □

Recall that $pot(u)$ is the set of indices i among $1, \ldots, k$ for which there is no $j > i$ with $d_j = d_i$.

Definition 7.1 suggests that the oracle $\mathcal{O}$ should construct $\mathcal{O}_{\mathcal{L}}(u, V)$ recursively from its subtrees, constructed as trees of form $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^g), \alpha^{-1}V)$, once the set I has been determined. Thus, the key step in the construction is to determine the set I. We will let I contain an index $i \in pot(u)$ precisely when this is necessary, i.e., if some suffix of form $\alpha(d_i)v$ with $v \in [\![\alpha^{-1}V]\!]$ would be incorrectly classified when $i \notin I$. Looking more closely at this, we see that when $i \notin I$, then $\alpha(d_i)v$ is classified by the branch reached via the guard g defined as $\bigwedge_{j \in I} p \neq x_j$, i.e., v is classified by $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), \alpha^{-1}V)$. Since, by the recursive nature of the construction, we can assume that $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), \alpha^{-1}V)$ correctly classifies suffixes $v \in [\![\alpha^{-1}V]\!]$ when they occur after the prefix $u\alpha(d_i)$, it seems that one could check whether $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), \alpha^{-1}V)$ classifies suffixes after $u\alpha(d_i)$ in the same way as $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), \alpha^{-1}V)$ by checking whether these two trees are isomorphic. However, one must then first take into consideration that $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), \alpha^{-1}V)$ is "specialized" to the case where the ith and $k + 1$st data values in the prefix $u\alpha(d_i)$ are equal (implying that it never contains an occurrence of x_i since $i \notin pot(u\alpha(d_i))$), which is not the case for $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), \alpha^{-1}V)$. We illustrate this observation by the following example.

Let $\mathcal{L}$ contain all words of form $\alpha(d_1)\alpha(d_2)$, and let $\mathcal{L}$ contain a word of form $\alpha(d_1)\alpha(d_2)\alpha(d_3)$ precisely when $d_1 = d_3$ or $d_2 = d_3$. Let u be $\alpha(1)$, let V be $\{\alpha\alpha\}$, and choose $\mathtt{d}_u^0$ as 0. The subtree corresponding to $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), \alpha^{-1}V)$ is then $\mathcal{O}_{\mathcal{L}}(u\alpha(0), \{\alpha\})$, and the subtree corresponding to $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), \alpha^{-1}V)$ is $\mathcal{O}_{\mathcal{L}}(u\alpha(1), \{\alpha\})$. These trees are shown in Figure 5. We see that before comparing them, we must first specialize $\mathcal{O}_{\mathcal{L}}(u\alpha(0), \{\alpha\})$ to the case when x_1 and x_2 are equal, and that such a specialization will replace x_1 by x_2, thereby making the two right subtrees isomorphic, one of which can be removed, thereby transforming $\mathcal{O}_{\mathcal{L}}(u\alpha(0), \{\alpha\})$ into a tree which is isomorphic to $\mathcal{O}_{\mathcal{L}}(u\alpha(1), \{\alpha\})$.

Thus, before defining our oracle, we must first define how to specialize a subtree $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), \alpha^{-1}V)$ to classify suffixes after the prefix $u\alpha(d_i)$ (in which the last symbol in the prefix is equal to the ith). We will define such a specialization, which specializes an equality tree for (u, V) to the situation where some subset $\{d_j : j \in J\}$ of values in u are equal. For a set J of indices, let $max(J)$ be the largest index in J. For two indices i, j, let $J[j/i]$ denote $(J \setminus \{i\}) \cup \{j\}$.

Definition 7.2 (Specialization of Equality Tree). Let $\mathcal{T}$ be an equality tree for (u, V), and let $J \subseteq pot(u)$ be a set of indices. Then $\mathcal{T}\langle J \rangle$ denotes the equality tree for (u, V) obtained from $\mathcal{T}$ by performing the following transformation for each α:

- whenever $\mathcal{T}$ has several initial α-transitions of form $\langle l_0, \alpha(p), (p = x_j), l_j \rangle$ with $j \in J$, then all subtrees of form $(\mathcal{T}[l_j])\langle J[k + 1/j] \rangle$ for $j \in J$ must be defined and isomorphic, otherwise $\mathcal{T}\langle J \rangle$ is undefined. If all such subtrees are defined and isomorphic, then $\mathcal{T}\langle J \rangle$ is obtained from $\mathcal{T}$ by

  1. replacing all initial α-transitions of form $\langle l_0, \alpha(p), (p = x_j), l_j \rangle$ for $j \in J$ by the single transition $\langle l_0, \alpha(p), (p = x_m), l_m \rangle$, where m is $max(J)$,
  2. replacing $\mathcal{T}[l_m]$ by $(\mathcal{T}[l_m])\langle J[k + 1/m] \rangle$, and
  3. replacing all other subtrees $\mathcal{T}[l']$ reached by initial α-transitions (which have not been replaced in the Step 1) by $(\mathcal{T}[l'])\langle J \rangle$.

If, for some α, any of the subtrees generated in Step 2 or 3 are undefined, then $\mathcal{T}\langle J\rangle$ is also undefined, otherwise $\mathcal{T}\langle J\rangle$ is as obtained after performing Steps 1 - 3 for each α. □

We can now describe how our tree oracle $\mathcal{O}$ generates an equality tree $\mathcal{O}_\mathcal{L}(u, V)$ for any prefix u and set of symbolic suffixes V.

Definition 7.3 (Tree oracle for equality). For a language $\mathcal{L}$, a prefix u and set of symbolic suffixes V, the equality tree $\mathcal{O}_\mathcal{L}(u, V)$ is constructed as follows.

- If $V = \{\epsilon\}$, then $\mathcal{O}_\mathcal{L}(u, \{\epsilon\})$ is the trivial tree with one location l_0 and no registers. It accepts (i.e., $\lambda(l_0) = +$) if $u \in \mathcal{L}$, otherwise $\lambda(l_0) = -$. To find out whether $u \in \mathcal{L}$, the tree oracle performs a membership query for u.

- If $V \neq \{\epsilon\}$, then for each α such that $V_\alpha = \alpha^{-1}V$ is nonempty,

  – let I be the subset of indices i in $pot(u)$ such that $\mathcal{O}_\mathcal{L}(u\alpha(\mathrm{d}_u^0), V_\alpha)\langle\{i, k+1\}\rangle$ is undefined or $\mathcal{O}_\mathcal{L}(u\alpha(\mathrm{d}_u^0), V_\alpha)\langle\{i, k+1\}\rangle \not\simeq \mathcal{O}_\mathcal{L}(u\alpha(d_i), V_\alpha)$ (i.e., I is the set of indices such that $\mathcal{O}_\mathcal{L}(u\alpha(\mathrm{d}_u^0), V_\alpha)$ classifies suffixes after $u\alpha(d_i)$ incorrectly),

  – $\mathcal{O}_\mathcal{L}(u, V)$ is constructed as $\mathcal{O}_\mathcal{L}(u, V) = (L, l_0, \Gamma, \lambda)$, where, letting $\mathcal{O}_\mathcal{L}(u\alpha(d_i), V_\alpha)$ be the tuple $(L_i^\alpha, l_{0i}^\alpha, \Gamma_i^\alpha, \lambda_i^\alpha)$ for $i \in (I \cup \{0\})$,

    · L is the disjoint union of all L_i^α plus an additional initial location l_0,

    · Γ is the union of all Γ_i^α for $i \in (I \cup \{0\})$, and in addition the transitions of form $\langle l_0, \alpha(p), g_i, l_{0i}^\alpha\rangle$ with $i \in (I \cup \{0\})$, where g_i is $\bigwedge_{j \in I} p \neq x_j$ for $i = 0$, and g_i is $p = x_i$ for $i \neq 0$, and

    · λ agrees with each λ_i^α on L_i^α. Moreoever, if $\epsilon \in V$, then $\lambda(l_0) = +$ if $u \in \mathcal{L}$, and $\lambda(l_0) = -$ if $u \notin \mathcal{L}$. Again, to find out whether $u \in \mathcal{L}$, the tree oracle performs a membership query for u.

  Intuitively, $\mathcal{O}_\mathcal{L}(u, V)$ is constructed by joining the trees $\mathcal{O}_\mathcal{L}(u\alpha(d_i), V_\alpha)$ with guard $p = x_i$ for $i \in I$, and the tree $\mathcal{O}_\mathcal{L}(u\alpha(\mathrm{d}_u^0), V_\alpha)$ with guard $\bigwedge_{j \in I} p \neq x_j$ as children of a new root. □

Definition 7.3 suggests that the tree oracle $\mathcal{O}$ can construct the equality tree $\mathcal{O}_\mathcal{L}(u, V)$ recursively as follows. For each equivalence class induced by $\approx_\mathcal{R}$ on the set $u[\![V]\!]$, choose a representative data word uv in that class, determine whether $uv \in \mathcal{L}$ by performing a membership query, and construct the trivial equality tree $\mathcal{O}_\mathcal{L}(uv, \{\epsilon\})$ Each such $\mathcal{O}_\mathcal{L}(uv, \{\epsilon\})$ is a trivial tree with a single location labeled by $+$ (accept) or $-$ (reject). Thereafter, recursively construct $\mathcal{O}_\mathcal{L}(uv', V')$ for increasingly shorter uv' and V's with increasingly longer symbolic suffixes following the rules in Definition 7.3.

Having defined the tree oracle $\mathcal{O}$, we must now prove that it is indeed canonical, i.e., that it satisfies the conditions in Definition 5.7. Before we do that, we establish some auxiliary properties. For a word u with $Vals(u) = d_1, \ldots, d_k$ and a set $J \subseteq pot(u)$, let σ_u^J denote a substitution operation on data words, which replaces all d_i with $i \in J$ by d_m, where $m = max(J)$. Thus, $\sigma_u^J(v)$ is the data word obtained from v by replacing all d_i with $i \in J$ by d_m, where $m = max(J)$.

Proposition 7.4. Let $\mathcal{L}$ be a language, let u be a prefix, and V, V' be sets of symbolic suffixes with $V \subseteq V'$. If $\mathcal{O}_\mathcal{L}(u, V')\langle J\rangle$ is defined, then there is a data language $\mathcal{K}$ such that

1. $\mathcal{O}_\mathcal{L}(u, V')\langle J\rangle \simeq \mathcal{O}_\mathcal{K}(\sigma_u^J(u), V')$, and
2. $\mathcal{O}_\mathcal{L}(u, V)\langle J\rangle$ is defined and $\mathcal{O}_\mathcal{L}(u, V)\langle J\rangle \simeq \mathcal{O}_\mathcal{K}(\sigma_u^J(u), V)$.

Proof. We first observe that for any $\mathcal{L}$, u, V', and J we have that $\mathcal{O}_\mathcal{L}(u, V')\langle J\rangle$ is defined if and only if $uv \in \mathcal{L} \leftrightarrow uv' \in \mathcal{L}$ whenever $v, v' \in [\![V']\!]$ are such that $\sigma_u^J(v) = \sigma_u^J(v')$; intuitively, this states that $\mathcal{L}$ does not distinguish between data values in $\{d_j : j \in J\}$ when they occur in a suffix $v \in [\![V']\!]$ after u. The property can be established by induction over the depth of V', following the construction in Definition 7.3.

We then let the language $\mathcal{K}$ be defined in such a way that $\sigma_u^J(u)\sigma_u^J(v) \in \mathcal{K} \leftrightarrow uv \in \mathcal{L}$ for any $v \in [\![V']\!]$. Intuitively, $\mathcal{K}$ is classifies words obtained by the replacement σ_u^J in the same way as $\mathcal{L}$ classifies the original words. Note that for suffixes v' which cannot be obtained as $\sigma_u^J(v)$ for any v, one may have to determine whether $\sigma_u^J(u)v' \in \mathcal{K}$ from information about whether $\sigma_u^J(u)v'' \in \mathcal{K}$ for some suffix v'' such that $\sigma_u^J(u)v''$ is equivalent to $\sigma_u^J(u)v'$. For instance, if $\mathcal{L}$ is the languague used for the example in Figure 5, and u is $\alpha(1)\alpha(0)$ and J is $\{1, 2\}$, then we infer that $\alpha(0)\alpha(0)\alpha(1) \notin \mathcal{K}$ by observing that the equivalent word $\alpha(0)\alpha(0)\alpha(42)$ is not in $\mathcal{K}$.

We can then establish Property 1, i.e., that $\mathcal{O}_{\mathcal{L}}(u, V')\langle J\rangle \simeq \mathcal{O}_{\mathcal{K}}(\sigma_u^J(u), V')$, again by induction over the depth of V'.

To establish Property 2, we infer that $\mathcal{O}_{\mathcal{L}}(u, V)\langle J\rangle$ is defined by noting that it is equivalent to the property that $uv \in \mathcal{L} \leftrightarrow uv' \in \mathcal{L}$ whenever $v, v' \in [\![V]\!]$ are such that $\sigma_u^J(v) = \sigma_u^J(v')$, which follows from the just established same property for V' and $V \subseteq V'$. We similarly infer $\mathcal{O}_{\mathcal{L}}(u, V)\langle J\rangle \simeq \mathcal{O}_{\mathcal{K}}(\sigma_u^J(u), V)$, from the same property for V' and $V \subseteq V'$. $\qquad\square$

In order to prove that the tree oracle $\mathcal{O}$ defined by Definition 7.3 is canonical (as Theorem 7.6), we also need the following lemma.

Lemma 7.5. Let $V \subseteq V'$. Then for any prefix $u, \tilde{u}$, data languages $\mathcal{L}, \mathcal{K}$, and any bijection $\gamma : \mathcal{X}(\mathcal{O}_{\mathcal{L}}(u, V')) \mapsto \mathcal{X}(\mathcal{O}_{\mathcal{K}}(\tilde{u}, V'))$ we have that $\mathcal{O}_{\mathcal{L}}(u, V') \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$ implies $\mathcal{O}_{\mathcal{L}}(u, V) \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$.

Intuitively, Lemma 7.5 states that any equality tree for (u, V), generated by our canonical tree oracle is uniquely determined by the corresponding (u, V')-tree. This will essentially imply Condition 2 of Definition 5.7. Condition 3 also follows implicitly by the constructions in the following proof.

Proof. We prove the lemma by induction on the depth of V. The base case, $V = \{\epsilon\}$, follows immediately. In the inductive step, we assume that the property is true for any V'', V' u, $\tilde{u}$, $\mathcal{L}$, and $\mathcal{K}$, where where V'' replaces V in the property to be proven, and has strictly smaller depth than V. We can assume that V contains some non-empty word. For each action α such that $V_\alpha = \alpha^{-1}V$ is nonempty, let $V'_\alpha = \alpha^{-1}V'$, implying $V_\alpha \subseteq V'_\alpha$. Let $Vals(u)$ be $d_1, \ldots, d_k$ and let $Vals(\tilde{u})$ be $\tilde{d}_1, \ldots, \tilde{d}_{\tilde{k}}$. For g being $\bigwedge_{j \in pot(\tilde{u})} p \neq x_j$, let $\mathsf{d}_{\tilde{u}}^g$ be $\tilde{d}_0$. For $i \in pot(u)$ such that $x_i \in \mathcal{X}(\mathcal{O}_{\mathcal{L}}(u, V'))$, let $\gamma(i)$ be defined by $x_{\gamma(i)} = \gamma(x_i)$. For a bijection γ between a subset of $x_1, \ldots, x_k$ and a subset of $x_1, \ldots, x_{\tilde{k}}$, let $\hat{\gamma}$ be the extension of γ obtained by extending the domain of γ with x_{k+1}, and defining $\hat{\gamma}(x_{k+1}) = x_{\tilde{k}+1}$.

In this inductive step, we assume $\mathcal{O}_{\mathcal{L}}(u, V') \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$ and must prove $\mathcal{O}_{\mathcal{L}}(u, V) \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$. The assumption $\mathcal{O}_{\mathcal{L}}(u, V') \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$ means that for each α such that $V_\alpha = \alpha^{-1}V$ is nonempty,

(i') $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^0), V'_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_0), V'_\alpha)$

(ii') $\mathcal{O}_{\mathcal{L}}(u, V')$ has an initial α-guard $p = x_i$ iff $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$ has an initial α-guard $p = \gamma(x_i)$, and $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V'_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_{\gamma(i)}), V'_\alpha)$.

We must establish that $\mathcal{O}_{\mathcal{L}}(u, V) \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$, i.e., that the corresponding properties hold also when V' is replaced by V. These corresponding properties are:

(i) $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^0), V_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_0), V_\alpha)$

(ii) $\mathcal{O}_{\mathcal{L}}(u, V)$ has an initial α-guard $p = x_i$ iff $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$ has an initial α-guard $p = \gamma(x_i)$, and $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_{\gamma(i)}), V_\alpha)$.

For this, we may as inductive hypothesis use that $\mathcal{O}_{\mathcal{L}}(u\alpha(d), V'_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}), V'_\alpha)$ implies $\mathcal{O}_{\mathcal{L}}(u\alpha(d), V_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}), V_\alpha)$ for any d, $\mathcal{L}$, and $\mathcal{K}$ (since V_α has strictly smaller depth than V). Let us prove properties (i) and (ii) separately.

(i) The property $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^0), V_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_0), V_\alpha)$ trivially follows by the inductive hypothesis.

(ii) Assume first that $\mathcal{O}_{\mathcal{L}}(u, V)$ has an initial α-guard $p = x_i$. We first establish that $\mathcal{O}_{\mathcal{L}}(u, V')$ has an initial α-guard $p = x_i$. For this, we note that the assumption that $\mathcal{O}_{\mathcal{L}}(u, V)$ has an initial α-guard $p = x_i$ implies that either

- $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^0), V_\alpha)\langle\{i, k+1\}\rangle$ is undefined, in which case $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^0), V'_\alpha)\langle\{i, k+1\}\rangle$ is undefined by Proposition 7.4, implying that $\mathcal{O}_{\mathcal{L}}(u, V')$ has an initial α-guard $p = x_i$, or

- $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^0), V_\alpha)\langle\{i, k+1\}\rangle \not\simeq \mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha)$, in which case we infer by the inductive hypothesis and Proposition 7.4 that either

  – $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^0), V'_\alpha)\langle\{i, k+1\}\rangle$ is undefined, again implying that $\mathcal{O}_{\mathcal{L}}(u, V')$ has an initial α-guard $p = x_i$, or

  – there is a data language $\hat{\mathcal{L}}$ such that both $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^0), V'_\alpha)\langle\{i, k+1\}\rangle \simeq \mathcal{O}_{\hat{\mathcal{L}}}(u\alpha(\mathsf{d}_u^0), [)\{i, k+1\}]V'_\alpha$

and $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), V_\alpha)\langle\{i, k+1\}\rangle \simeq \mathcal{O}_{\hat{\mathcal{L}}}(u\alpha(\mathtt{d}_u^0), [)\{i, k+1\}]V_\alpha$. In this case, we can instantiate the inductive hypothesis as

$$\mathcal{O}_{\hat{\mathcal{L}}}(u\alpha(\mathtt{d}_u^0), V_\alpha') \simeq \mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha') \text{ implies } \mathcal{O}_{\hat{\mathcal{L}}}(u\alpha(\mathtt{d}_u^0), V_\alpha) \simeq \mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha) \quad ,$$

which by Proposition 7.4 means that

$$\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), V_\alpha')\langle\{i, k+1\}\rangle \simeq \mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha') \text{ implies } \mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), V_\alpha)\langle\{i, k+1\}\rangle \simeq \mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha)$$

from which we conclude that $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), V_\alpha')\langle\{i, k+1\}\rangle \not\simeq \mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha')$.

Having established that $\mathcal{O}_{\mathcal{L}}(u, V')$ has an initial α-guard $p = x_i$, we infer from the assumption $\mathcal{O}_{\mathcal{L}}(u, V') \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$ that $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$ has an initial α-guard $p = x_{\gamma(i)}$ and that $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha') \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_{\gamma(i)}), V_\alpha')$, implying by the inductive hypothesis that $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_{\gamma(i)}), V_\alpha)$. From the previously established condition (i), we obtain that either

- $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), V_\alpha)\langle\{i, k+1\}\rangle$ and $\mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_0), V_\alpha)\langle\{\gamma(i), \tilde{k}+1\}\rangle$ are both undefined, or
- $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), V_\alpha)\langle\{i, k+1\}\rangle \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_0), V_\alpha)\langle\{\gamma(i), \tilde{k}+1\}\rangle$.

From this, the initial assumption about $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathtt{d}_u^0), V_\alpha)\langle\{i, k+1\}\rangle$ and $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha)$, and the just established $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_{\gamma(i)}), V_\alpha)$, we infer that either $\mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_0), V_\alpha)\langle\{\gamma(i), \tilde{k}+1\}\rangle$ is undefined or that $\mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_{\gamma(i)}), V_\alpha) \not\simeq \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_0), V_\alpha)\langle\{\gamma(i), \tilde{k}+1\}\rangle$, implying that $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$ has an initial α-guard $p = x_{\gamma(i)}$. From the inductive hypothesis, we immediately get that $\mathcal{O}_{\mathcal{L}}(u\alpha(d_i), V_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}_{\gamma(i)}), V_\alpha)$.

The proof in the other direction, where we assume $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$ has an initial α-guard $p = x_{\gamma(i)}$, follows by a symmetric argument. $\qquad\square$

Theorem 7.6. *The tree oracle $\mathcal{O}$ defined by Definition 7.3 is canonical.*

Proof. The property that $\mathcal{O}$ is indeed a tree oracle for $\mathcal{L}$ follows from the construction in Definition 7.3, using Proposition 7.4. To prove that $\mathcal{O}$ is canonical, we must prove that it satisfies the three conditions in Definition 5.7. Condition 1 follows directly from the construction in Definition 7.3. Condition 2 follows as a special case of Lemma 7.5, by taking $\mathcal{K}$ to be $\mathcal{L}$. Finally, Condition 3 follows by noting that if $p = x_i$ is an initial guard of $\mathcal{O}_{\mathcal{L}}(u, V)$, then $p = \gamma(x_i)$ is an initial guard of $\mathcal{O}_{\mathcal{L}}(u, V')$, implying that the "inequality guard" of $\mathcal{O}_{\mathcal{L}}(u, V)$ includes fewer conjuncts than that of $\mathcal{O}_{\mathcal{L}}(u, V')$, and hence is weaker. This observation also implies that $\mathcal{X}(\mathcal{O}_{\mathcal{L}}(u, V)) \subseteq \mathcal{X}(\mathcal{O}_{\mathcal{L}}(u, V'))$. $\qquad\square$

7.2. Tree Construction for the Theory of Inequalities

In this section, we present a realization of a tree oracle for the theory of equalities and inequalities over rational or real numbers. Our tree oracle generates (u, V)-trees of a special form, called *interval trees*, in which guards are open, closed, or half-open intervals. We will consider a *u-guard* to be of form $x_i \sim p \sim' x_j$, of form $p \sim' x_j$, or of form $x_i \sim p$, such that $\sim, \sim' \in \{<, \leq\}$ and $i, j \in pot(u)$, which is satisfiable under ν_u. We write $p = x_i$ for $x_i \leq p \leq x_i$. As representative value for $p = x_i$, we take d_i. As representative value for an interval of form $x_i \sim p \sim' x_j$ with $i \neq j$, we take a data value $d \notin Vals(u)$ with $d_i < d < d_j$. Let us now define interval trees.

Definition 7.7 (Interval tree). Let u be a data word with $Vals(u) = d_1, \ldots, d_k$, and let V be a set of symbolic suffixes. An *interval tree* (for (u, V) is a (u, V)-tree $\mathcal{T}$ such that

- For each action α, the set of initial α-guards is a set of intervals of form

$$p \sim x_{i_1} \qquad x_{i_1} \sim p \sim x_{i_2} \qquad \ldots \qquad x_{i_{m-1}} \sim p \sim x_{i_m} \qquad x_{i_m} \sim p \quad ,$$

  where each $\sim$ is in $\{<, \leq\}$ and $i_1, \ldots, i_m$ are indices in $pot(u)$ with $d_{i_1} \leq \ldots \leq d_{i_m}$, such that for each $j = 1, \ldots, m$ the guards contain at least one non-strict comparison with x_{i_j},

- for each initial transition $\langle l_0, \alpha(p), g, l \rangle$ of $\mathcal{T}$, the tree $\mathcal{T}[l]$ is an interval tree for $(u\alpha(\mathtt{d}_u^g), \alpha^{-1}V)$. $\qquad\square$

   Our tree oracle will construct interval trees by a procedure which is similar in structure to that used for constructing equality trees, with some natural differences. In analogy with the case for the theory of equalities, we need to specialize interval trees to the case when several data values in u are made equal. We therefore include a definition analogous to Definition 7.2. In the setting of intervals, such a definition makes sense only when the values made equal are adjacent to each other in $Vals(u)$.

   Let u be a prefix with $Vals(u) = d_1, \ldots, d_k$. A subset J of $pot(u)$ is *adjacent in* u if $d_i < d_n < d_j$ for $n \in pot(u)$ and $i, j \in J$ implies $n \in J$.

Definition 7.8 (Specialization of interval tree). Let $\mathcal{T}$ be an interval tree for $\langle u, V \rangle$, and let J be a subset of $pot(u)$ which is adjacent in u. Then $\mathcal{T}\langle J \rangle$ denotes the equality tree for (u, V) obtained from $\mathcal{T}$ by performing the following transformation for each α:

- whenever $\mathcal{T}$ has several initial α-transitions of form $\langle l_0, \alpha(p), (x_i \sim p \sim' x_{i'}), l_i \rangle$ with $\sim, \sim'$ in $\{<, \leq\}$ and $i, i' \in J$, then all subtrees of form $(\mathcal{T}[l_i])\langle J \cup \{k+1\} \rangle$ must be defined and isomorphic, otherwise $\mathcal{T}\langle J \rangle$ is undefined. If all such subtrees are defined and isomorphic, then $\mathcal{T}\langle J \rangle$ is obtained from $\mathcal{T}$ by

   1. replacing all initial α-transitions of form $\langle l_0, \alpha(p), (x_i \sim p \sim' x_{i'}), l_i \rangle$ of $\mathcal{T}$, where $\sim, \sim'$ in $\{<, \leq\}$ and $i, i' \in J$, by the single transition $\langle l_0, \alpha(p), (p = x_m), l_i \rangle$ where m is $max(J)$, and i is some index in J for which $\mathcal{T}$ has an initial α-transition of form $\langle l_0, \alpha(p), (x_i \sim p \sim' x_{i'}), l_i \rangle$,
   2. replacing $\mathcal{T}[l_i]$ by $(\mathcal{T}[l_i])\langle J \cup \{k+1\} \rangle$, and
   3. replacing all other initial subtrees $\mathcal{T}[l']$ by $(\mathcal{T}[l'])\langle J \rangle$.

   If, for some α, any of the subtrees generated in Step 2 or 3 are undefined, then $\mathcal{T}\langle J \rangle$ is also undefined, otherwise $\mathcal{T}\langle J \rangle$ is as obtained after Steps 1 - 3 for each α.   □

   We can now describe how our tree oracle $\mathcal{O}$ generates an interval tree $\mathcal{O}_{\mathcal{L}}(u, V)$ for any prefix u and set of symbolic suffixes V.

Definition 7.9 (Tree oracle for inequalities). For a language $\mathcal{L}$, a prefix u and set of symbolic suffixes V, the interval tree $\mathcal{O}_{\mathcal{L}}(u, V)$ is constructed as follows.

- If $V = \{\epsilon\}$, then $\mathcal{O}_{\mathcal{L}}(u, \{\epsilon\})$ is the trivial tree with one location l_0 and no registers. It accepts (i.e., $\lambda(l_0) = +$) if $u \in \mathcal{L}$, otherwise $\lambda(l_0) = -$. To find out whether $u \in \mathcal{L}$, the tree oracle performs a membership query for u.
- If V contains some non-empty word, then for each α such that $V_\alpha = \alpha^{-1}V$ is nonempty, let $g_1^\alpha, \ldots, g_n^\alpha$ be the maximal u-guards such that either

   - g_i^α is of form $p = x_i$, or
   - g_i^α is a non-trivial interval, and for any $d' \in D$ with $\nu_u \models g_i^\alpha[d'/p]$, which is a representative data value some u-guard, we have, letting $\mathcal{T}_i^\alpha$ denote $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^{g_i^\alpha}), V_\alpha)$,

      · $\mathcal{O}_{\mathcal{L}}(u\alpha(d'), V_\alpha) \simeq \mathcal{T}_i^\alpha$, whenever $d' \notin Vals(u)$, and

      · $\mathcal{T}_i^\alpha \langle \{x_j, x_{k+1}\} \rangle$ is defined and $\mathcal{O}_{\mathcal{L}}(u\alpha(d'), V_\alpha) \simeq \mathcal{T}_i^\alpha \langle \{x_j, x_{k+1}\} \rangle$ whenever $d' = d_j$ for some $j \in pot(u)$.

   Now $\mathcal{O}_{\mathcal{L}}(u, V)$ is constructed by joining the trees $\mathcal{T}_1^\alpha, \ldots, \mathcal{T}_n^\alpha$ as children of a new root, using the guards $g_1^\alpha, \ldots, g_n^\alpha$ in the natural way. More precisely, if we let $\mathcal{T}_i^\alpha$ be the tuple $(L_i^\alpha, l_{0i}^\alpha, \Gamma_i^\alpha, \lambda_i^\alpha)$, then $\mathcal{O}_{\mathcal{L}}(u, V) = (L, l_0, \Gamma, \lambda)$, where

      · L is the disjoint union of all L_i^α plus an additional initial location l_0,

      · Γ is the union of all Γ_i^α and for each $i = 1, \ldots, n$ the transition $\langle l_0, \alpha(p), g_i^\alpha, l_{0i}^\alpha \rangle$,

      · λ agrees with each λ_i^α on L_i^α, and if $\epsilon \in V$, then $\lambda(l_0) = +$ if $u \in \mathcal{L}$, and $\lambda(l_0) = -$ if $u \notin \mathcal{L}$. Again, to find out whether $u \in \mathcal{L}$, the tree oracle performs a membership query for u.   □

Definition 7.9 suggests that the tree oracle $\mathcal{O}$ can be realized as follows. Given u and V, we first construct a trivial interval tree $\mathcal{O}_{\mathcal{L}}(uv, \{\epsilon\})$ for a representative data word uv in each equivalence class of words in $u[\![V]\!]$. Each such $\mathcal{O}_{\mathcal{L}}(uv, \{\epsilon\})$ is a trivial tree with a single location labeled by $+$ (accept) or $-$ (reject). We then recursively construct $\mathcal{O}_{\mathcal{L}}(uv', V')$ for increasingly shorter uv' and appropriate V', following the construction in the definition.

We will next prove that the tree oracle for inequalities is canonical, i.e., that it satisfies the conditions in Definition 5.7. Before we can do that, we need to establish some properties.

Proposition 7.10. Let $\mathcal{L}$ be a language, let u be a prefix, and V, V' be sets of symbolic suffixes with $V \subseteq V'$. Let $J \subseteq pot(u)$ be adjacent in u. If $\mathcal{O}_{\mathcal{L}}(u, V')\langle J \rangle$ is defined, then there is a data language $\mathcal{K}$ such that

1. $\mathcal{O}_{\mathcal{L}}(u, V')\langle J \rangle \simeq \mathcal{O}_{\mathcal{K}}(\sigma_u^J(u), V')$, and
2. $\mathcal{O}_{\mathcal{L}}(u, V)\langle J \rangle$ is defined, and $\mathcal{O}_{\mathcal{L}}(u, V)\langle J \rangle \simeq \mathcal{O}_{\mathcal{K}}(\sigma_u^J(u), V)$.

Proof. The proof follows the same lines as the proof of Proposition 7.4. The language $\mathcal{K}$ is defined in such a way that $\sigma_u^J(u)\sigma_u^J(v) \in \mathcal{K} \leftrightarrow uv \in \mathcal{L}$ for any $v \in [\![V']\!]$.   □

In order to prove that the tree oracle $\mathcal{O}$ defined by Definition 7.9 is canonical (as Theorem 7.12), we also need the following lemma.

Lemma 7.11. Let $V \subseteq V'$. Then for any prefixes $u, \tilde{u}$, data languages $\mathcal{L}, \mathcal{K}$, and any bijection $\gamma : \mathcal{X}(\mathcal{O}_{\mathcal{L}}(u, V')) \mapsto \mathcal{X}(\mathcal{O}_{\mathcal{K}}(\tilde{u}, V'))$ we have $\mathcal{O}_{\mathcal{L}}(u, V') \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$ implies $\mathcal{O}_{\mathcal{L}}(u, V) \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$.

Proof. We prove the lemma by induction on the depth of V. The base case, $V = \{\epsilon\}$, follows immediately. For the inductive step, we assume that the property is true for any $V'', V' u, \tilde{u}, \mathcal{L}$, and $\mathcal{K}$, where where V'' replaces V in the property to be proven, and has strictly smaller depth than V. We can assume that V contains some non-empty word. For each action α such that $V_\alpha = \alpha^{-1}V$ is nonempty, let $V'_\alpha = \alpha^{-1}V'$, implying $V_\alpha \subseteq V'_\alpha$. Let $Vals(u)$ be $d_1, \ldots, d_k$ and let $Vals(\tilde{u})$ be $\tilde{d}_1, \ldots, \tilde{d}_{\tilde{k}}$. For $i \in pot(u)$ such that $x_i \in \mathcal{X}(\mathcal{O}_{\mathcal{L}}(u, V'))$, let $\gamma(i)$ be defined by $x_{\gamma(i)} = \gamma(x_i)$. We extend γ to guards and trees in the natural way. For a bijection γ between a subset of $x_1, \ldots, x_k$ and a subset of $x_1, \ldots, x_{\tilde{k}}$, let $\hat{\gamma}$ be the extension of γ obtained by extending the domain of γ with x_{k+1}, and defining $\hat{\gamma}(x_{k+1}) = x_{\tilde{k}+1}$.

For the inductive step, we assume $\mathcal{O}_{\mathcal{L}}(u, V') \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$, which means that for each α such that $V_\alpha = \alpha^{-1}V$ is nonempty,

(i') $\mathcal{O}_{\mathcal{L}}(u, V')$ has an initial α-guard g iff $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$ has an initial α-guard $\gamma(g)$,
   and then $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^g), V'_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\mathsf{d}_{\tilde{u}}^{\gamma(g)}), V'_\alpha)$.

We shall establish that this property holds also when V' is replaced by V, i.e.,

(i) $\mathcal{O}_{\mathcal{L}}(u, V)$ has an initial α-guard g iff $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$ has an initial α-guard $\gamma(g)$,
   and then $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^g), V_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\mathsf{d}_{\tilde{u}}^{\gamma(g)}), V_\alpha)$.

For this, we may as inductive hypothesis use that for any $d, \tilde{d}, \mathcal{L}$, and $\mathcal{K}$, we have that $\mathcal{O}_{\mathcal{L}}(u\alpha(d), V'_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}), V'_\alpha)$ implies $\mathcal{O}_{\mathcal{L}}(u\alpha(d), V_\alpha) \simeq_{\hat{\gamma}} \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\tilde{d}), V_\alpha)$. Let us move to the proof of (i). From the inductive hypothesis and the construction of guards in Definition 7.9, it follows that each initial α-guard of $\mathcal{O}_{\mathcal{L}}(u, V)$ includes some initial α-guard of $\mathcal{O}_{\mathcal{L}}(u, V')$, and that each initial α-guard of $\mathcal{O}_{\mathcal{L}}(u, V')$ is included in some initial α-guard of $\mathcal{O}_{\mathcal{L}}(u, V)$. The analogous property holds for $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$.

Consider some initial α-guard g of $\mathcal{O}_{\mathcal{L}}(u, V)$. By properties of representative data values, d_u^g is the same as d_u^h for some initial α-guard h of $\mathcal{O}_{\mathcal{L}}(u, V')$, with $\nu_u \models h \to g$. The corresponding property holds for representative data values of form $\mathsf{d}_{\tilde{u}}^g$. From this observation, and the property that for all initial α-guards h of $\mathcal{O}_{\mathcal{L}}(u, V')$ we have $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^h), V_\alpha) \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\mathsf{d}_{\tilde{u}}^{\gamma(h)}), V_\alpha)$, which follows from the inductive hypothesis, we conclude $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^g), V_\alpha) \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\mathsf{d}_{\tilde{u}}^{\gamma(g)}), V_\alpha)$. It remains to prove that the initial α-guards of $\mathcal{O}_{\mathcal{L}}(u, V)$ and $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$ are the same (modulo γ). This follows from an argument analogous to that in the proof of Lemma 7.5, by establishing the properties that

1. for all initial α-guards g of $\mathcal{O}_{\mathcal{L}}(u, V)$, and all initial act-guards h of $\mathcal{O}_{\mathcal{L}}(u, V')$ we have $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^g), V_\alpha) \simeq \mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^h), V_\alpha)$ if and only if $\mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\mathsf{d}_{\tilde{u}}^{\gamma(g)}), V_\alpha) \simeq \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\mathsf{d}_{\tilde{u}}^{\gamma(h)}), V_\alpha)$, and that
2. $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^g), V_\alpha)\langle\{i, k+1\}\rangle \simeq \mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^h), V_\alpha)$
   if and only if $\mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\mathsf{d}_{\tilde{u}}^{\gamma(g)}), V_\alpha)\langle\{\gamma(i), \tilde{k}+1\}\rangle \simeq \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\mathsf{d}_{\tilde{u}}^{\gamma(h)}), V_\alpha)$.

These properties imply that there is a bijection between the initial α-guards g of $\mathcal{O}_{\mathcal{L}}(u, V)$ and those of $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$ such that $\mathcal{O}_{\mathcal{L}}(u\alpha(\mathsf{d}_u^g), V_\alpha) \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}\alpha(\mathsf{d}_{\tilde{u}}^{\gamma(g)}), V_\alpha)$. Moreover, an initial α-guard g of $\mathcal{O}_{\mathcal{L}}(u, V)$

includes an initial α-guard h of $\mathcal{O}_{\mathcal{K}}(u, V')$ if and only if the initial α-guard $\gamma(g)$ of $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$ includes the initial α-guard $\gamma(h)$ of $\mathcal{O}_{\mathcal{K}}(\tilde{u}, V')$. (There is also an additional case, where g includes an additional guard of form $p = x_i$, which can be handled analogously). This establishes property (i), and hence proves that $\mathcal{O}_{\mathcal{L}}(u, V) \simeq_\gamma \mathcal{O}_{\mathcal{K}}(\tilde{u}, V)$. □

Theorem 7.12. *The tree oracle $\mathcal{O}$ defined by Definition 7.9 is canonical.*

Proof. The property that $\mathcal{O}$ is indeed a tree oracle for $\mathcal{L}$ follows from the construction in Definition 7.9, using Proposition 7.10. To prove that $\mathcal{O}$ is canonical, we must prove that it satisfies the three conditions in Definition 5.7. Condition 1 follows directly from the construction in Definition 7.9. Condition 2 follows as a special case of Lemma 7.11, by taking $\mathcal{K}$ to be $\mathcal{L}$. Finally, Condition 3 follows as a by-product of the proof of Lemma 7.11. □

7.3. Extensions of Tree Oracles

We now discuss two extensions of the tree oracles presented in this section: *constants*, and *increments* over integers. As a consequence of the extensions, tree construction and the arguments for canonicity provided in Section 7.1 can be adapted.

Constants. A constant c is a particular value from the data domain. In our framework, constants can be represented by a unary relation EQC $\subseteq \mathcal{D}$ containing only c, i.e. with EQC $= \{c\}$. In a (u, V)-tree with a constant c, guards are equalities or conjunctions of inequalities, as in the theory of equality, but we also enable comparisons with c, e.g., of the form $p = c$, or $p < c$, . To avoid unnecessary storing of constants in registers, the oracle will prefer the guard $p = c$ over any comparison to a register, whenever the value in some register is equal to c. The extension also does not affect tree construction and canonicity: We can imagine constants during tree construction as extending the potential of any data word to contain the constants.

An example for the theory of inequalities with constants is the model of the prepaid card shown in the bottom right of Figure 8. On several transitions the updated balance p is compared to constants (e.g., $200 < p$). We could also, for example, modify the model of the pump shown in Fig. 4 to test p against some constant threshold level.

Increments. An increment can be described as a binary relation: INC(m, n) iff $m + 1 = n$. The tree oracle adds some restrictions in order to be able to construct canonical (u, V)-trees when theories are extended with increments. These restrictions are necessary since theories with increments are not, in general, weakly extendable. First, the guards in a (u, V)-tree for the theory of increments over integers will either be of the form $x_i + 1 = p$ or $p = x_i$ (a guard of the form $p + 1 = x_i$ is thus not allowed). Second, sometimes there will be a choice between two guards ($x_i + 1 = p$ and $p = x_j$ are both true at the same time). To avoid this, the tree oracle first tries to replace the prefix u with another $\mathcal{R}$-equivalent prefix in which the two guards are *not* both true at the same time. If there is such a prefix, the problem is solved. If there is no such prefix, the tree oracle uses the equality guard $p = x_j$ as the default guard (but in general, either guard would work since, in this case, there is no data word for which only one of the guards is satisfied). An example for the theory of increments is the sequence number counter shown in the top middle of Figure 8.

8. The SL* Algorithm

This section presents our active automata learning algorithm SL^*. The algorithm constructs an automaton (specifically, an SRA) for an unknown data language $\mathcal{L}$ by inferring locations, registers, and transitions.

8.1. Preliminaries

A central data structure in the algorithm is the *observation table*. It stores data words and corresponding SDTs, organizing this information so as to be able to distinguish locations from each other.

Definition 8.1 (Observation table). An observation table is a tuple $\langle U, U^+, V, Z \rangle$, where

- U is a prefix-closed set of data words, called *short prefixes*,

Prefixes $U \cup U^+$	Symbolic suffixes $V = \{\epsilon, \text{ level, level level}\}$
(l_0) ϵ	(tree: level(p)/true → level(p)/true)
(l_1) level(1)	(tree with branches: level(p) $x_1 \leq p$; $\{x_1\}$ level(p) $p < x_1$, $x_2 := p$; level(p) true; level(p) $p < x_2$; level(p) $x_2 \leq p$)
(l_2) level(2)level(1)	(tree: level(p) $x_2 \leq p$; $\{x_2\}$ level(p) $p < x_2$; level(p) true; level(p) true)
(l_{sink}) level(2)level(1)level(0)	(tree: level(p)/true → level(p)/true)
level(1)level(2)	same as l_1 (x_2 renamed x_1)
level(2)level(3)level(1)	same as l_2 (x_3 renamed x_2)
level(2)level(1)level(3)	same as l_1 (x_3 renamed x_1)
level(3)level(1)level(2)	same as l_1 (x_3 renamed x_1)
level(3)level(2)level(1)level(0)	same as l_{sink}
level(1)level(2)level(4)level(3)	same as l_{sink}
. . .	. . .

Fig. 6. Part of a closed and register-consistent observation table for the water pump example.

- U^+ is a set of *extended prefixes*, each of the form $u\alpha(d)$ for some $u \in U$ (and, in general, not disjoint from U),
- V is a set of symbolic suffixes, and
- Z maps each element u in $U \cup U^+$ to a (u, V)-tree.   □

The set U^+ of extended prefixes contains exactly those data words of the form $u\alpha(d)$ where d is the representative data value $\mathtt{d}_u^g$ of some initial α-guard of $Z(u)$.

An observation table $\langle U, U^+, V, Z \rangle$ is *closed*, if for every $u\alpha(d) \in U^+$ there is a short prefix $u' \in U$ and a γ such that $Z(u\alpha(d)) \simeq_\gamma Z(u')$. It is *register-consistent* if, for each $u\alpha(d) \in U^+$, whenever $x_i \in \mathcal{X}(Z(u\alpha(d)))$ for $i \leq |Vals(u)|$, then there is a register $x_i \in \mathcal{X}(Z(u))$. Closedness ensures that all transitions in the automaton have a target location. Register-consistency ensures that whenever a data value in u is needed to construct the SDT after $u\alpha(d)$, then that data value also can be stored in a register in the SDT after u.

Definition 8.2 (Hypothesis automaton). A closed and register-consistent observation table $\langle U, U^+, V, Z \rangle$ can be used to construct a hypothesis automaton $Hyp(\langle U, U^+, V, Z \rangle) = (L, l_0, \mathcal{X}, \Gamma, \lambda)$, where

- $L = U$ and $l_0 = \epsilon$,
- $\mathcal{X}$ maps each location $u \in U$ to $\mathcal{X}(Z(u))$ (note that $\mathcal{X}(l_0)$ is the empty set),
- $\lambda(u) = +$ if $u \in \mathcal{L}$, otherwise $\lambda(u) = -$, and
- each initial α-transition $\langle l_0, \alpha(p), g, \pi, l' \rangle$ of $Z(u)$ generates a transition $\langle u, \alpha(p), g, \pi', u' \rangle$ in Γ, where

  - u' is the (unique) short prefix in U with $Z(u\alpha(\mathtt{d}_u^g)) \simeq_\gamma Z(u')$,
  - π' is an assignment $\mathcal{X}(Z(u')) \mapsto (\mathcal{X}(Z(u)) \cup \{p\})$. For $x_i \in \mathcal{X}(Z(u'))$, we define $\pi'(x_i) = \gamma^{-1}(x_i)$ if $\gamma^{-1}(x_i)$ stores a data value of u in $Z(u\alpha(\mathtt{d}_u^g))$, and $\pi'(x_i) = p$ otherwise.   □

Example 8.3. Fig. 6 shows a closed and register-consistent observation table for the water pump example

in Section 2. A set of symbolic suffixes V labels the column; rows are labeled with short prefixes from U (marked with locations) and with extended prefixes from U^+. Each table cell (referred to by row label u and column label V) stores the SDT $Z(u)$. The observation table can be used to generate the automaton in Fig. 4, with locations obtained from the short prefixes. The assignment $x_1 := p$ on the transition from l_1 to l_2 is derived from the SDT for the prefix $\mathsf{level}(2)\mathsf{level}(1)$. All other assignments (on transitions to l_1) are derived from the SDT for the prefix $\mathsf{level}(1)$.   □

Algorithm 1 SL^*

Require: A set Σ of actions, a data language $\mathcal{L}$, a tree oracle $\mathcal{O}$ for the theory.
Ensure: An SRA $\mathcal{H}$ with $\mathcal{L}(\mathcal{H}) = \mathcal{L}$ ▷ **Initialization**
1: $U \leftarrow \{\epsilon\}$, $V \leftarrow (\{\epsilon\} \cup \Sigma)$, $Z(\epsilon) \leftarrow \mathcal{O}_{\mathcal{L}}(\epsilon, V)$
2: **loop**
3:    **repeat**   ▷ **Hypothesis construction**
4:      $U^+ \leftarrow \{u\alpha(\mathsf{d}_u^g) \ : \ u \in U, \ \alpha \in \Sigma, \text{ and } g \text{ initial } \alpha\text{-guard of } Z(u)\}$
5:      **for** each $u \in (U \cup U^+)$ **do** $Z(u) \leftarrow \mathcal{O}_{\mathcal{L}}(u, V)$
6:        **if** $\exists u\alpha(d) \in U^+$ s.t. $Z(u\alpha(d)) \not\simeq_\gamma Z(u')$ for any γ and $u' \in U$ **then**   Check for closedness
7:          $U \leftarrow U \cup \{u\alpha(d)\}$
8:        **end if**
9:        **if** $\exists u\alpha(d) \in U^+$, $\exists d_i \in Vals(u)$, and $\exists x_i \in \mathcal{X}(Z(u\alpha(d)))$ Check for register-consistency
            s.t. $\nu_u(x_i) = d_i$, but $x_i \notin \mathcal{X}(Z(u))$ **then**
10:         $V \leftarrow V \cup \{\alpha(d) \cdot v\}$ for $v \in V$ with $x_i \in \mathcal{X}(\mathcal{O}_{\mathcal{L}}(u\alpha(d), \{v\}))$
11:       **end if**
12:     **end for**
13:   **until** $\langle U, U^+, V, Z \rangle$ **is** closed **and** register-consistent

14:   $\mathcal{H} \leftarrow Hyp(\langle U, U^+, V, Z \rangle)$ ▷ **Hypothesis validation**
15:   **if** $eq(\mathcal{H})$ **then**
16:     **return** $\mathcal{H}$

17:   **else**   ▷ **Counterexample processing**
18:     let $\sigma = \alpha_1(d_1) \dots \alpha_n(d_n)$ be the returned counterexample
19:     **for** $\langle u_{i-1}, \alpha_i(p), g_i, \pi_i, u_i \rangle$ in run of $\mathcal{H}$ over σ **do**
20:       **if** g_i does not refine an initial transition of $\mathcal{O}_{\mathcal{L}}(u_{i-1}, V_{i-1})$ **then**   *Case 1:* New transition
21:         $V \leftarrow V \ \cup \ V_{i-1}$
22:       **end if**
23:       **if** $\mathcal{O}_{\mathcal{L}}(u_{i-1}\alpha_i(d_i), V_i) \not\simeq_\gamma \mathcal{O}_{\mathcal{L}}(u_i, V_i)$ for γ used to construct $\mathcal{H}$ **then**   *Case 2:* New location / wrong remapping
24:         $V \leftarrow V \ \cup \ V_i$
25:       **end if**
26:     **end for**
27:   **end if**
28: **end loop**

8.2. Algorithm description

Algorithm 1 shows a pseudocode description of SL^*. The algorithm is initialized (line 1) with U containing the empty word, the set of symbolic suffixes V being the empty sequence together with the set of all actions, and $Z(\epsilon)$ being the SDT $\mathcal{O}_{\mathcal{L}}(\epsilon, V)$. The algorithm then iterates three phases: *hypothesis construction, hypothesis validation*, and *counterexample processing* until no more counterexamples are found, monotonically adding locations and transitions to hypothesis automata. We detail these phases below, referring to the corresponding lines in Algorithm 1.

Hypothesis construction (lines 3-13). The algorithm constructs a hypothesis automaton by making tree queries. The result of a tree query for a prefix u and a set V of symbolic suffixes is the (u, V)-tree $\mathcal{O}_{\mathcal{L}}(u, V)$, which is stored in the observation table as as $Z(u)$. The algorithm continues to make tree queries as long as the observation table is not closed or not register-consistent:

- If the table is not closed, then there is an extended prefix $u\alpha(d) \in U^+$ that leads to a location that is not identified by any of the short prefixes in U, i.e., $Z(u\alpha(d))$ is not equivalent to $Z(u')$ for any short prefix u'. This is remedied by making $u\alpha(d)$ a short prefix, i.e., adding it to U.

- If the table is not register-consistent, then $Z(u\alpha(d))$ has a register x_i that expects a value from u but $Z(u)$ does not have a register for storing this value. This is remedied by extending V: for the particular symbolic suffix $v \in V$ that uses the register x_i in $Z(u\alpha(d))$, we add the symbolic suffix αv to V. Subsequent tree queries will then ensure that x_i is added to $Z(u)$.

Hypothesis validation (lines 14-16). The hypothesis automaton $\mathcal{H}$ is submitted for an equivalence query. The teacher either replies "yes", or returns a counterexample (a word that is accepted by $\mathcal{H}$ but rejected by the target system, or vice versa). If it replies "yes", the algorithm terminates and returns $\mathcal{H}$. Otherwise, the counterexample has to be analyzed.

In a black-box scenario, of course, equivalence queries cannot be implemented: since the teacher does not have access to the internals of system under learning, it cannot compare the system to the inferred model directly. Thus, in practice, equivalence queries are usually approximated by some form of search for a counterexample. Search strategies can help establishing additional properties on conjectures and counterexamples: We use exhaustive exploration of the the set of data words up to some fix time limit in the experiments presented in Section 9. While this is a very expensive search strategy, it provides two guarantees: (1) we only find shortest counterexamples, and (2) we know that at every time the current hypothesis is definitely correct for words up to some specific length.

Random walks, on the other hand, have proven to be a simple yet very effective strategy for finding counterexamples fast [AHKV14, CHJ15]. While finding counterexamples very efficiently, they do not yield obvious guarantees on the inferred models and can produce fairly long counterexamples. This is not a problem in principle, but it can lead to adding long suffixes to the observation table. Long suffixes can have a dramatic influence on the efficiency (number of tree queries and cost of individual tree queries). Removing loops from counterexamples [AHKV14, CHJ15] has proven to be a very effective heuristic for controlling the effect of long counterexamples on the performance of the learning algorithm.

Counterexample analysis (lines 17-27). A counterexample indicates either that a location is missing, (i.e., that U has to be extended), or that a transition is missing, (i.e., that the initial transitions of SDTs need to be refined), or that we used an incorrect renaming γ between some SDTs when constructing the hypothesis. All three cases can be remedied by adding an abstract suffix of the counterexample to the set V of symbolic suffixes of the observation table. The challenge here is determining which abstract suffix to add to V. For a counterexample σ of length m, we denote by σ_i its prefix of length i, and by v_i its suffix of length $m - i$. Moreover, let V_i be the singleton set $\{Acts(v_i)\}$.

In a run of $\mathcal{H}$ over σ, the i-th step $\langle u_{i-1}, \nu_{i-1} \rangle \xrightarrow{\alpha_i(d_i)} \langle u_i, \nu_i \rangle$ traverses transition $\langle u_{i-1}, \alpha_i(p), g_i, \pi_i, u_i \rangle$, i.e., prefix σ_i leads to the location corresponding to short prefix u_i from U. Since σ is a counterexample, we know that at some point, the run of $\mathcal{H}$ over σ diverges from the behavior of the component we are trying to learn. In order to determine where this happens, we analyze the sequence $u_0 = \epsilon, \ldots, u_m$ of short prefixes and the corresponding (u_i, V_i)-trees for $0 \le i \le m$ computed by $\mathcal{O}_{\mathcal{L}}(u_i, V_i)$, using an argument similar to the one presented in [RS93].

For the first tree $\mathcal{O}_{\mathcal{L}}(u_0, V_0)$, the prefix u_0 is the empty word ϵ, and V_0 contains all actions of σ. The tree's run over σ conflicts with the run of $\mathcal{H}$ (one is accepting while the other is not). The last tree in the sequence, $\mathcal{O}_{\mathcal{L}}(u_m, V_m)$, on the other hand, is *not* in conflict with $\mathcal{H}$. Since $V_m = \{\epsilon\}$, the tree $\mathcal{O}_{\mathcal{L}}(u_m, V_m)$ contains only one node, stating whether u_m is accepted or not. This tree was originally supplied as the result of a tree query for the word u_m, so it must be correct. Hence, there is at least one index j of the counterexample for which the following holds:

- The short prefix u_{j-1} and $\mathcal{O}_{\mathcal{L}}(u_{j-1}, V_{j-1})$ contain a counterexample to $\mathcal{H}$ in the subtree(s) that are within the scope of g_j, the corresponding guard in $\mathcal{H}$.
- The short prefix u_j and $\mathcal{O}_{\mathcal{L}}(u_j, V_j)$ do not contain a counterexample.

The index j can be found in a binary search. We can then distinguish two cases.

Case 1: New transition. The guard g_j in the step of $\mathcal{H}$ from u_{j-1} to u_j does not refine an initial transition of $\mathcal{O}_{\mathcal{L}}(u_{j-1}, V_{j-1})$, i.e., g_i intersects with two guards of $\mathcal{O}_{\mathcal{L}}(u_{j-1}, V_{j-1})$. In this case the SDT distinguishes cases that $\mathcal{H}$ does not distinguish. Adding the one element in V_{j-1} to V will result in new and refined transitions from u_{j-1} in the hypothesis. This is guaranteed by the monotonicity requirement on tree oracles in Def. 5.7.

Case 2: New location / wrong remapping. The tree $\mathcal{O}_{\mathcal{L}}(u_j, V_j)$ is not isomorphic to the corresponding subtree

after $\alpha(\mathsf{d}_{u_{j-1}}^{g_j})$ of $\mathcal{O}_{\mathcal{L}}(u_{j-1}, V_{j-1})$ under the renaming of registers γ that was used in the hypothesis: the subtree of $\mathcal{O}_{\mathcal{L}}(u_{j-1}, V_{j-1})$ contains a counterexample to $\mathcal{H}$, but $\mathcal{O}_{\mathcal{L}}(u_j, V_j)$ does not. By adding the one element in V_j to V, either γ will be refined, or $\mathcal{O}_{\mathcal{L}}(u_j, V)$ and $\mathcal{O}_{\mathcal{L}}(u_{j-1}\alpha(\mathsf{d}_{u_{j-1}}^{g_j}), V)$ will be found inequivalent, making $u_{j-1}\alpha(\mathsf{d}_{u_{j-1}}^{g_j})$ a separate location.

Example 8.4. Suppose we are learning the water pump component, shown in Fig. 4. The first hypothesis has only one state and accepts all data words. A counterexample for this hypothesis is, e.g., $\mathsf{level}(3)\mathsf{level}(2)\mathsf{level}(1)$, which is accepted by the hypothesis but not valid on the target component. The counterexample is processed using binary search between index 0 and index 3 to find the index where its behavior diverges from that of the hypothesis, and thus, what symbolic suffix to add to the observation table. Instead of tracing the binary search, we discuss the results for all indices of the counterexample.

- At index 3, we have $\sigma = \mathsf{level}(3)\mathsf{level}(2)\mathsf{level}(1)$ and, in the hypothesis, σ corresponds to the short prefix $u_3 = \epsilon$. We are at the end of the counterexample, so – by construction – there is no counterexample in the SDT $\mathcal{O}_{\mathcal{L}}(u_3, \{\epsilon\})$: this SDT has been used to determine the acceptance of the location for u_3 in the hypothesis.
- At index 2, we have $\sigma_2 = \mathsf{level}(3)\mathsf{level}(2)$ and $V_2 = \{\mathsf{level}(p)\}$. In the hypothesis, σ_2 corresponds to the short prefix $u_2 = \epsilon$. The hypothesis and the SDT $\mathcal{O}_{\mathcal{L}}(u_2, V_2) = \mathcal{O}_{\mathcal{L}}(\epsilon, \{\mathsf{level}(p)\})$ have the same behavior (everything is accepted).
- At index 1, we have $\sigma_1 = \mathsf{level}(3)$ and $V_1 = \{\mathsf{level}(p_1)\mathsf{level}(p_2)\}$. In the hypothesis, σ_1 corresponds to the short prefix $u_1 = \epsilon$. The hypothesis and the SDT $\mathcal{O}_{\mathcal{L}}(u_1, V_1)$ also have the same behavior (everything is accepted).
- At index 0, we have $\sigma_0 = \epsilon$ and $V_0 = \{\mathsf{level}(p_1)\mathsf{level}(p_2)\mathsf{level}(p_3)\}$ In the hypothesis, σ_0 corresponds to the short prefix $u_0 = \epsilon$. The behavior of the SDT $\mathcal{O}_{\mathcal{L}}(u_0, V_0)$ diverges from that of the hypothesis within the scope of g_1 (which is *true*), e.g., in the case of σ.

A binary search could, e.g., start at index 2, move to the left since there is no counterexample in the SDT, continue at index 1 which still has no counterexample, and end between index 0 and index 1 with $j = 1$. Then, either Case 1 or Case 2 applies.

- Case 1 does not apply: In both the SDT for index 0 and the hypothesis, there is an initial $\mathsf{level}(p)$-transition guarded by *true*, so the guard in the hypothesis can be said to refine the guard in the SDT.
- Case 2 applies: Let $\alpha(\mathsf{d}_{u_0}^{g_1}) = \mathsf{level}(1)$. The SDT $\mathcal{O}_{\mathcal{L}}(u_1, V_1)$ (see Fig. 3 (a)) is not equivalent to the subtree of $\mathcal{O}_{\mathcal{L}}(u_0, V_0)$ after $\mathsf{level}(1)$ (see Fig. 3 (b)). The symbolic suffix $V_0 = \mathsf{level}(p_1)\mathsf{level}(p_2)$ is added to the observation table. $\square$

8.3. Correctness and termination

That SL^* returns a correct SRA upon termination follows by the properties of equivalence queries. For regular data languages, termination follows from the properties of SDTs in Section 5, from Theorem 6.3, and from the algorithm itself: SDTs will only be refined when adding symbolic suffixes, and this can happen only finitely often. Each added symbolic suffix will either lead to a new transition, a refined transition, a new register assignment or a new location. By adapting arguments from other contexts [Ang87, HSJC12], Theorem 6.3 can be used to show that SL^* converges to a minimal (in terms of locations and registers) SRA for $\mathcal{L}$. Note that this minimal number of locations and transitions also depends on the particular tree oracle that is used.

Complexity. We estimate the worst case number of counterexamples and show how they lead to a correct model with n locations, t transitions, and at most r registers per location. Since each location has one access sequence, $n \leq t$, and thus we estimate the costs in t and r only. Each counterexample results in one additional suffix in the observation table, leading to a new transition or to discarding a bijection between two prefixes in U. The former can happen t times before all transitions are identified. The latter can happen at most tr times, since it corresponds to breaking a symmetry between two of at most r registers at one of $n \leq t$ locations (cf. [How12]). The algorithm terminates after $O(tr)$ equivalence queries. The number of tree queries depends on the length m of the longest counterexample and on the size of the observation table. The algorithm uses a maximum of m calls per counterexample, and the size of $U \cup U^+$ in the final observation table is $t + 1$. This leads to $O(t^2 r + trm)$ tree queries and yields the following theorem.

Theorem 8.5. The algorithm SL^* infers a data language $\mathcal{L}$ with $O(tr)$ equivalence queries and $O(t^2r + trm)$ tree queries.   □

9. Evaluation

In this section, we evaluate our learning framework on a number of example components. The example components are small, but represent a set of different theories. The purpose of this evaluation is demonstrating the feasibility of the conceptual framework i.e., using different theories in a generic automata learning algorithm). To this end, we have implemented theories from the different classes discussed in Section 5 that generate SDTs from tests. The SL^* algorithm has been implemented prototypically using LearnLib[4] [IHS15], an automata learning framework developed by some of the authors. We conduct experiments in a simulation environment. The behavior of black-box components is emulated using register automata as "ground truth".

The development and evaluation of a tool for inferring register automaton models of software components is an ongoing effort. We are working on a proper extension to LearnLib (namely RaLib). It is based on the conceptual architecture presented in this paper and on the prototypical implementation used in this evaluation. We evaluate performance and practicality of RaLib in [CHJ15] where we also compare its performance against other tools that infer register automata models with the theory allows only equalities over positive integers — namely Tomte [AHK+12] and the implementation of our own algorithm from [HSJC12]. Experiments show that RaLib is competitive: It outperforms the other tools on most of the benchmarks. Tomte has an edge on benchmarks with many registers (e.g., big data structures with a capacity of more than 10 elements). Detailed comparisons of all three tools can be found in [AHKV14] and [CHJ15] but are not presented here since these tools use a number of optimizations that are beyond the scope of this work.

In the remainder of the section, we first discuss the theories and technical details of the implementation, before describing each of the example components and our experimental results.

9.1. Implementation

We have implemented the SL^* algorithm together with a teacher for a select set of theories, and an approximate equivalence oracle for a black-box scenario. Technically, the theories are implemented in a modular fashion, such that compatible relations (i.e., sums, equalities, or inequalities) can be combined into theories. For the scope of this paper, we used four dedicated basic theories (augmented by constants for some examples).

- $\langle \mathbb{N}, \{=\} \rangle$: The most basic theory allows only equalities over positive integers. This is essentially a new implementation of the theory of equality used in [HSJC12] (but allowing the use of constants).
- $\langle \mathbb{N}, \{=, \text{INC}\} \rangle$: The second theory for equalities additionally supports increments on positive integers (cf. Section 7.3). We use this theory for modeling and inferring the behavior of sequence numbers.
- $\langle \mathbb{R}^+, \{<, =\} \rangle$: The theory of equalities and inequalities over positive real numbers makes it possible to model systems such as, e.g., a priority queue.
- $\langle \mathbb{R}^+, \{<, =, \text{SUM}\} \rangle$: We extended the theory of equalities and inequalities of positive reals to also support binary sums of data values (i.e., guards of form $x_i + x_j = p$), where d_i and d_j are values in a data word. We also implemented a restricted version (denoted by SUM_c) that only supports guards of form $x_i + c = p$ where c is a constant. The implementation uses the same ideas that have been discussed in Section 7.3: we augment the potential of a data word to contain also sums of data.

The implementation of tree queries $\mathcal{O}_\mathcal{L}(u, V)$ is based on the ideas presented in Section 7: We create SDTs from the tests using the relations from $\mathcal{R}$ as guards and merge equivalent subtrees from the leaves to the root, joining guards as necessary.

The teacher uses a constraint solver for two operations: (1) to generate concrete (i.e., representative) data values for test cases from symbolic constraints on these values, and (2) to find differences between multiple symbolic decision trees (i.e., acceptance vs. rejection for certain data values). These differences are computed when processing counterexamples (cf. Section 8). The input for the constraint solver is provided by the
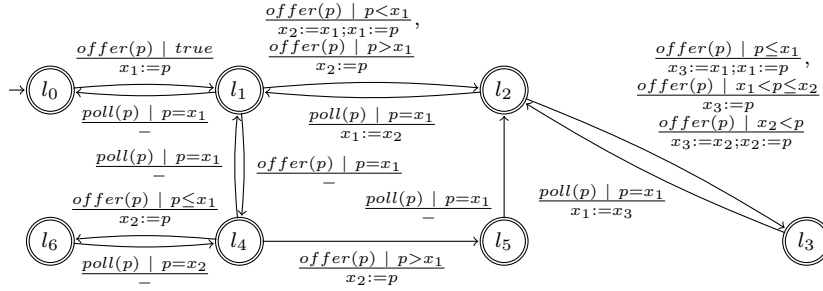
---

[4] http://www.learnlib.de

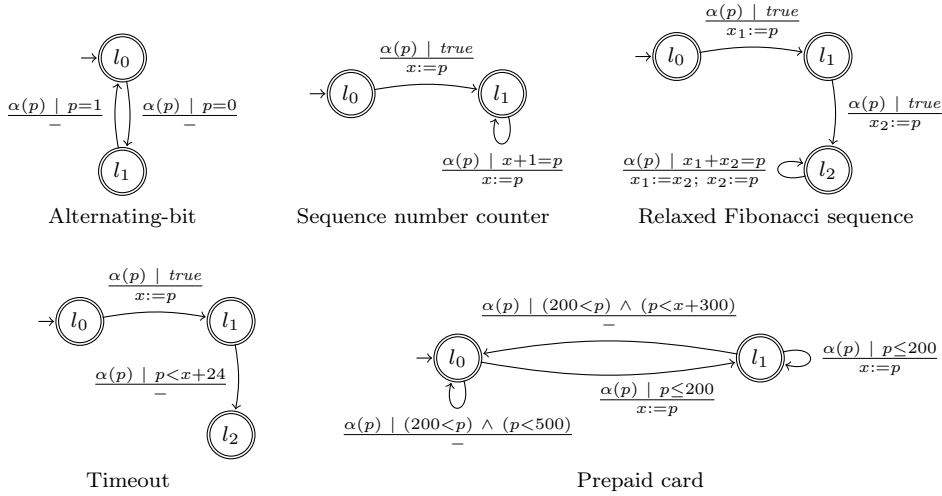Fig. 7. Priority queue from the Java Class Library.

Fig. 8. Small models combining control flow and data.

learning algorithm, without requiring white-box access to the system under learning. We use Z3[5] [dMB08] as a constraint solver.

Equivalence queries have been implemented to search for counterexamples by exhaustive exploration up to some fix time limit. We use tree queries (similar to the approach in [GRR12]) and generate $\mathcal{O}_{\mathcal{L}}(\epsilon, w)$ for all $w \in \Sigma^k$ up to some depth k and compare the SDTs to the hypothesis. We start with $k = 3$ and increase k until the fixed time limit of 10 minutes is reached or until a counterexample is found. This ensures that upon termination we can guarantee the correctness of the inferred model up to some depth k. In other works [AHKV14, CHJ15] randomized generation of data words and random walks have proved successful and efficient, especially when investigating counterexamples for loops before submitting the counterexamples to the learning algorithm.

9.2. Benchmarks

We have inferred a simplified version of the connection establishment phase of TCP, a bounded priority queue from the Java Class Library, and a set of five smaller benchmark systems (Alternating-bit protocol, Sequence number, Timeout, a prepaid card, and a Fibonacci counter). Models for the smaller examples are shown in Fig. 8. Rejecting states are omitted in the figures, and we assume a single action $\alpha(p)$.

Alternating-bit. This example is an abstract version of the alternating bit protocol (shown in the top left

---

$$\xrightarrow{} l_0 \quad \xrightarrow[x:=p]{syn-ack(p) \mid p=x+1} \quad l_3 \quad \xrightarrow[-]{fin-ack(p) \mid p=x} \quad l_4 \quad \xrightarrow[-]{\substack{syn-ack(p) \mid p=x+1 \\ x:=p \\ fin-ack(p) \mid p=x}}$$

$$\xrightarrow[x:=p]{init(p) \mid p=x} \qquad \xrightarrow[-]{ack(p) \mid p=x} \qquad \xrightarrow[-]{fin-ack(p) \mid p=x} \qquad \xrightarrow[-]{ack(p) \mid p=x}$$

$$l_1 \quad \xrightarrow[x:=p]{syn(p) \mid p=x+1} \quad l_2 \qquad\qquad l_5$$

$$\substack{syn-ack(p) \mid p=x+1 \\ x:=p}, \quad \substack{fin-ack(p) \mid p=x \\ -}, \quad \substack{ack(p) \mid p=x \\ -} \qquad \substack{syn-ack(p) \mid p=x+1 \\ x:=p} \qquad \substack{syn(p) \mid p=x+1 \\ x:=p}, \quad \substack{fin-ack(p) \mid p=x \\ -}, \quad \substack{syn-ack(p) \mid p=x+1 \\ x:=p}, \quad \substack{ack(p) \mid p=x \\ -}$$
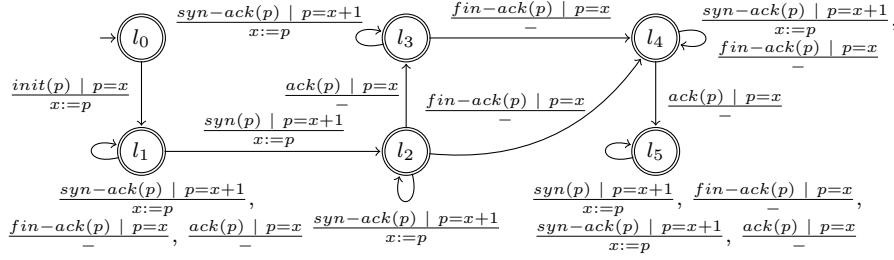
Fig. 9. Connection establishment of TCP.

of Fig. 8). It alternates two states; on each transition, a data parameter is tested for equality against two constants 0 and 1. This example uses the theory of integers with equalities, i..e, $\langle \mathbb{N}, \{=\} \rangle$, with constants. We write $p = 0$ and $p = 1$ in the figure instead of using unary relations $\text{EQ}0(p)$ and $\text{EQ}1(p)$.

Sequence number counter. The example in the center of the top row in Fig. 8) implements the behavior of a sequence number. The parameter value of the initial $\alpha(p)$ is stored in register x. All numbers in subsequent transitions then have to equal the previous sequence number plus one. This example uses the $\langle \mathbb{N}, \{=, \text{INC}\} \rangle$ theory of integers with equalities and increments. In the figure, we write $x + 1 = p$ instead of $\text{INC}(x, p)$.

Relaxed Fibonacci sequence. The system in the top right of Fig. 8 accepts a relaxed version of the Fibonacci sequence, i.e., with no constraint on the first two numbers in the sequence. We have implemented it using the theory of equalities, inequalities and binary sums of reals, i.e., $\langle \mathbb{R}^+, \{<, =, \text{SUM}\} \rangle$.

Timeout. The example (bottom left in Fig. 8) specifies a timeout. We use the parameters in actions to encode time stamps. In the first transition of this model, a data parameter (time stamp) is stored. A second transition is only available if performed within less than 24 time units (e.g., hours) after the first step. The model uses the theory of reals, equalities, inequalities, and binary sums of parameters and constants, i.e., $\langle \mathbb{R}^+, \{<, =, \text{SUM}_c\} \rangle$. The only used constant is 24. This example system would also accept a sequence of actions where the second time stamp is earlier (smaller) than the first time stamp. This could be avoided by introducing an additional constraint on the set of valid data words, similar to, e.g., timed languages.

Prepaid Card. The example in the bottom right of Fig. 8 models a prepaid card. The card's balance is limited to USD 500, and no more than USD 300 can be topped up in a single transaction. This example uses the $\langle \mathbb{R}^+, \{<, =, \text{SUM}_c\} \rangle$ theory of equalities, inequalities and binary sums of reals (of parameters and constants). The example needs three constants: 300, 500, and 200. The locations l_0 and l_1 encode whether the account balance is above USD 200 (l_0) or below USD 200 (l_1). The example shows that we do not infer a model of the prepaid card as such, but only of its interface behavior: the model does store the current balance only in case it is less than USD 200, when depositing more than USD 300 does not violate the general limit on the balance.

TCP. Fig. 9 shows the connection establishment phase of TCP. The example uses a set of five actions: $init$, syn, $syn-ack$, ack, and $fin-ack$. The transition $init(p)$ was added in order to generate an initial sequence number. After the initialization, each synchronizing message increases the set sequence number; all other messages use the current sequence number. Location l_5 represents a successfully established connection. The model uses the theory of equalities and increments of Integers $\langle \mathbb{N}, \{=, \text{INC}\} \rangle$. Studying the model carefully, we see that it obeys the restriction presented in Section 7.3: There is only one guarded special case for every action from every location.

Priority Queue. The inferred model of the Java class `java.util.PriorityQueue` is shown in Fig. 7. The capacity of the queue was limited manually to three elements in the test driver. The output of *poll* operations is modeled as extra input parameter, describing constraints on data values in traces of the queue. The learned model distinguishes between states with two different elements (l_2) and two identical elements (l_4) in the queue, since the learning algorithm does not consider subsumption between locations. This example illustrates the theory of reals with equalities and inequalities, i.e., $\langle \mathbb{R}^+, \{<, =\} \rangle$.

	No. of model parameters				Theory	No. of queries		Depth of EQs		Time [s]	
	locs	vars	trans	const	(domain, relations)	TQs	EQs	k_1	k_2	TQs	EQs
ABP	3	0	5	2	$\langle \mathbb{N}, \{=\}\rangle$	9	1	-	11	0.1	599.9
Sequence Number	3	1	4	0	$\langle \mathbb{N}, \{=, \text{INC}\}\rangle$	8	1	-	10	0.1	599.9
TCP	7	1	51	0	$\langle \mathbb{N}, \{=, \text{INC}\}\rangle$	187	2	6	7	0.6	599.4
PriorityQueue	8	2	33	0	$\langle \mathbb{R}^+, \{<, =\}\rangle$	113	5	6	7	4.3	595.7
Timeout	4	1	5	1	$\langle \mathbb{R}^+, \{<, =, \text{SUM}_c\}\rangle$	9	1	-	∞	0.2	0.1
Prepaid Card	3	1	7	3	$\langle \mathbb{R}^+, \{<, =, \text{SUM}_c\}\rangle$	16	2	3	4	1.3	598.7
Fibonacci counter	4	2	6	0	$\langle \mathbb{R}^+, \{<, =, \text{SUM}\}\rangle$	19	2	3	5	0.2	599.8

Table 1. Experimental results obtained on a 2GHz Intel Core i7 with 8GB of memory running Linux kernel 3.8.0.

9.3. Experiments

We have tested our implementation of SL^* on the set of examples presented above. Common optimizations for saving tests were used: a cache and a prefix-closure filter. Table 1 shows the results.

All experiments were conducted in a simulation environment provided by LearnLib for the evaluation of learning algorithms. In this simulation environment the behavior of components is specified using register automaton models. As described above, we implemented teachers for four theories. The teachers offer tree queries and equivalence queries to the SL^* algorithm. The teachers we implemented were faithful to the assumed black-box scenario and could only execute test cases on the target components. Tree queries were broken down into tests. Equivalence queries were approximated (as discussed above) by exhaustive exploration with a fixed limit on the overall runtime of each experiment.

We report the number of locations, variables, and transitions for all inferred models. For each case, we state the theory (number of constants, domain and relations). We also report the number of tree queries (TQs) and equivalence queries (EQs) made. For equivalence queries, we state the depth k_1 at which the last counterexample was found and the greatest explored depth k_2 (up to which inferred models are guaranteed to be correct). Finally, we show execution times.

Time consumption for learning is below one second for most of the examples; the only "real" Java class, the priority queue, takes a little more time (4.3 seconds). The difference between k_1 and k_2 gives an idea of how likely to be correct the final hypothesis is: If k_2 is bigger than k_1, then the depth was increased by $k_2 - k_1$ without finding a new counterexample. A large difference suggests that the learning algorithm has converged to the correct RA. For some examples no counterexamples where found and for the Timeout example $k_2 = \infty$, i.e., the equivalence query terminated successfully. This was possible because all sequences of length greater than two are not in the language of this example. For the theories with multiple relations ($\langle \mathbb{R}^+, \{<, =\}\rangle$, and $\langle \mathbb{R}^+, \{<, =, \text{SUM}_c\}\rangle$ or $\langle \mathbb{R}^+, \{<, =, \text{SUM}\}\rangle$) the reached depth k_2 is smaller, regardless of the number of locations and transitions in the final model. This is due to the exploding number of $\mathcal{R}$-distinguishable classes of data words in such cases. One way of addressing this challenge in practice is introducing typed parameters and using multiple simpler disjoint domains.

Another optimization that we have used in previous work [HIS+12] (for the restricted case with only equality tests) is extending the models with component output. This drastically reduces the effort needed for inference and allows to infer significantly larger models. Another rather straight-forward optimization is to allow a more selective refinement of decision trees, so that they grow only along paths where this is necessary.

We are implementing these optimizations in RaLib [CHJ15]. Documenting and evaluating these optimizations is beyond the scope of this work. The purpose of this evaluation was to demonstrate that interesting theories can be implemented and combined in a modular fashion with a generic automata learning algorithm. This was done successfully and the resulting software architecture and generic learning algorithm now actually serve as the basis of RaLib.

10. Conclusions

We have presented a symbolic active learning algorithm for generating EFSM models of black-box components using dynamic analysis. Our algorithm generalizes the classical L^* algorithm to the symbolic setting, so that it can generate register automata. The algorithm is parameterized on a particular theory (i.e., a set of operations and tests on the data domain that can be used in guards). It is the first fully automated technique that can, in principle, be combined with any theory and generate full RA models with variables, guards, and operations. with, e.g., sequence numbers, time stamps, or other variables that are manipulated using modestly complex arithmetic operations and relations. Our preliminary implementation demonstrates that the approach can infer protocols comprising sequence numbers, time stamps, and variables that are manipulated using simple arithmetic operations or compared for inequality even in a black-box scenario.

A particularly promising direction for future research will be the combination with white-box methods like symbolic execution, both for searching counterexamples as well as for supporting construction of decision trees. We also plan to investigate decidability of tree queries and equivalence queries in our learning model for different data domains.

References

[ABL02]   Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proc. POPL 2002*, pages 4–16. ACM, 2002.

[ACMN05]   Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Proc. POPL 2005*, pages 98–109. ACM, 2005.

[AdRP13]   Fides Aarts, Joeri de Ruiter, and Erik Poll. Formal models of bank cards for free. In *Proc. ICSTW 2013*, pages 461–468. IEEE, 2013.

[AHK+12]   Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits W. Vaandrager. Automata learning through counterexample guided abstraction refinement. In *Proc. FM 2012*, volume 7436 of *LNCS*, pages 10–27. Springer, 2012.

[AHKV14]   Fides Aarts, Falk Howar, Harco Kuppens, and Frits W. Vaandrager. Algorithms for inferring register automata - a comparison of existing approaches. In *Proc. ISoLA 2014, Part I*, volume 8802 of *LNCS*, pages 202–219. Springer, 2014.

[AJUV14]   Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, pages 1–41, 2014.

[AKT+12]   Fides Aarts, Harco Kuppens, Jan Tretmans, Frits W. Vaandrager, and Sicco Verwer. Learning and testing the bounded retransmission protocol. In *Proc. ICGI 2012*, volume 21 of *JMLR Proceedings*, pages 4–18. JMLR.org, 2012.

[Ang87]   Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[Arb04]   Farhad Arbab. Reo: a channel-based coordination model for component composition. *Math. Struct. in Comput. Sci.*, 14(3):329–366, 2004.

[ASV10]   Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and abstraction of the biometric passport. In *Proc. ISoLA 2010, Part I*, volume 6415 of *LNCS*, pages 673–686. Springer, 2010.

[BB13]   Matko Botinčan and Domagoj Babić. Sigma*: symbolic learning of input-output specifications. In *Proc. POPL 2013*, pages 443–456. ACM, 2013.

[BBC+06]   Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proc. 2006 EuroSys Conf.*, pages 73–85. ACM, 2006.

[BHLM13]   Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. A fresh approach to learning register automata. In *Proc. DLT 2013*, volume 7907 of *LNCS*, pages 118–130. Springer, 2013.

[BJK+04]   Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2004.

[BJR08]   Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In *Proc. FASE*, volume 4961 of *LNCS*, pages 317–331, 2008.

[BPT10]   Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of web services via planning in asynchronous domains. *Artif. Intell.*, 174(3-4):316–361, 2010.

[CGP01]   Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[CHJ15]   Sofia Cassel, Falk Howar, and Bengt Jonsson. RALib: A LearnLib extension for inferring efsms. In *DIFTS 2015*, 2015. Available online: www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf.

[CHJS14]   Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Learning extended finite state machines. In *Proc. SEFM 2014*, volume 8702 of *LNCS*, pages 250–264. Springer, 2014.

[dMB08]   Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[EPG+07]   Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[GHP02]   Eran Gery, David Harel, and Eldad Palachi. Rhapsody: A complete life-cycle model-based development system. In *Proc. IFM 2002*, volume 2335 of *LNCS*, pages 1–10. Springer, 2002.

[GIO12]   Roland Groz, Muhammad-Naeem Irfan, and Catherine Oriat. Algorithmic improvements on regular inference of software models and perspectives for security testing. In *Proc. ISoLA 2012, Part I*, volume 7609 of *LNCS*, pages 444–457. Springer, 2012.

[GRR12]   Dimitra Giannakopoulou, Zvonimir Rakamarić, and Vishwanath Raman. Symbolic learning of component interfaces. In *Proc. SAS 2012*, volume 7460 of *LNCS*, pages 248–264. Springer Berlin Heidelberg, 2012.

[HHNS02]   Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *Proc. FASE 2002*, volume 2306 of *LNCS*, pages 80–95. Springer, 2002.

[HIS+12]   Falk Howar, Malte Isberner, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. Inferring semantic interfaces of data structures. In *Proc. ISoLA 2012, Part I*, volume 7609 of *LNCS*, pages 554–571. Springer, 2012.

[HJM05]   Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *Proc. ESEC/FSE 2005*, pages 31–40. ACM, 2005.

[HNS03]   Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *Proc. CAV 2003*, volume 2725 of *LNCS*, pages 315–327. Springer, 2003.

[How12]   Falk Howar. *Active learning of interface programs*. PhD thesis, Technical University of Dortmund, Germany, 2012.

[HSJC12]   Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *Proc. VMCAI 2012*, volume 7148 of *LNCS*, pages 251–266. Springer, 2012.

[HSM11]   Falk Howar, Bernhard Steffen, and Maik Merten. Automata learning with automated alphabet abstraction refinement. In *Proc. VMCAI 2011*, volume 6538 of *LNCS*, pages 263–277. Springer, 2011.

[Hui07]   Antti Huima. Implementing Conformiq Qtronic. In *Proc. TestCom/FATES 2007*, volume 4581 of *LNCS*, pages 1–12. Springer, 2007.

[IHS14]   M. Isberner, F. Howar, and B. Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.

[IHS15]   M. Isberner, F. Howar, and B. Steffen. The open-source learnlib - A framework for active automata learning. In Daniel Kroening and Corina S. Pasareanu, editors, *Proc. CAV 2015*, volume 9206 of *LNCS*, pages 487–495. Springer, 2015.

[JM09]   Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, 2009.

[LMP08]   Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proc. ICSE 2008*, pages 501–510. ACM, 2008.

[MM14]   Oded Maler and Irini-Eleftheria Mens. Learning regular languages over large alphabets. In *Proc. TACAS 2014*, volume 8413 of *LNCS*, pages 485–499. Springer, 2014.

[RS93]   Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.

[SL07]   Guoqiang Shu and David Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS 2007*, page 25. IEEE, 2007.

[UL07]   Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007.

[WBDP10]   Neil Walkinshaw, Kirill Bogdanov, John Derrick, and Javier Paris. Increasing functional coverage by inductive testing: A case study. In *Proc. ICTSS 2010*, volume 6435 of *LNCS*, pages 126–141. Springer, 2010.

[XSL+13]   Hao Xiao, Jun Sun, Yang Liu, Shang-Wei Lin, and Chengnian Sun. Tzuyu: Learning stateful typestates. In *Proc. ASE 2013*, pages 432–442. IEEE, 2013.