DEQ: Equivalence Checker for Deterministic Register Automata

A. S. Murawski[1], S. J. Ramsay[2], and N. Tzevelekos[3]

[1] University of Oxford, UK
[2] University of Bristol, UK
[3] Queen Mary University of London, UK

Abstract. Register automata are one of the most studied automata models over infinite alphabets with applications in learning, systems modelling and program verification. We present an equivalence checker for deterministic register automata, called DEQ, based on a recent polynomial-time algorithm that employs group-theoretic techniques to achieve succinct representations of the search space. We compare the performance of our tool to other available implementations, notably the learning library RALib and nominal frameworks LOIS and NLambda.

Keywords: automata over infinite alphabets, computational group theory, nominal sets, register automata

1 Introduction

Register automata [10,18] are one of the simplest models of computation over infinite alphabets. They operate on an infinite domain of data by storing data values in a finite number of registers, where the values are available for future comparisons or updates. The automata can also recognise when a data value does not appear in any of the registers and, moreover, when a data value has not been seen so far at all [23].

Recent years have seen a surge of interest in models over infinite alphabets due to their ability to account for numerous computational scenarios with unbounded data. For instance, XML query languages [21] need to compare attribute values that originate from infinite domains. In programming verification, in turn, there is need for abstractions of computations over infinite datatypes [10] as well as scenarios involving unbounded resources, such as Java objects or ML references. The expressivity of register automata in this context led to a number of applications, such as run-time verification [8] and equivalence checking for fragments of ML [17] and Java [15]. More broadly, they have been advocated as a convenient formalism for systems modelling, which fuelled interest in extending learning algorithms to the setting [20,4,1,6,14].

This paper presents DEQ, a tool for verifying language equivalence of register automata in the deterministic case (the nondeterministic case is undecidable [18]). As many of the above-mentioned applications rely on equivalence checking, several implementations are available online for comparison. We compare the performance

of our tool to the equivalence routines built on RALib [5], which is a library for active learning algorithms for register automata, and two other implementations programmed in LOIS [12] and NLambda [11]. The latter two are frameworks for manipulating infinite structures in an imperative and functional style respectively, which make it possible to define register automata as first-class data and, in particular, naturally adapt the standard Hopcroft-Karp routine [9] to this setting.

Our experiments show that DEQ compares favourably to its competitors. At the theoretical level, this is thanks to being based on a recently proposed polynomial-time algorithm [16], which was the first polynomial-time algorithm for the problem. The algorithm improves upon the "naive" algorithm that would expand a register automaton to a finite-state automaton over a sufficiently large finite alphabet, often incurring an exponential blow-up. The implementation from RALib is similar in spirit to the naive approach but it incorporates a number of tricks used to handle symmetry. On the other hand, LOIS and NLambda represent infinite entities using first-order formulas and employ SMT solvers to evaluate the resultant programs. In particular, the equivalence-testing routine coded in NLambda has recently been used as part of an automata learning framework [14]. Ours is also the only tool that can handle global freshness, i.e. the recognition/generation of values that have not been seen thus far during the course of computation. This is an important feature in the context of programming languages that makes it possible to model object creation faithfully.

Register automata Throughout the paper, we let $\mathcal{D}$ be an infinite set (alphabet). Its elements will be called *data values*. Register automata are a simple model for modelling languages and behaviours over such an alphabet. They operate over finitely many states and, in addition, are equipped with a finite number of *registers*, where they can store elements of $\mathcal{D}$. Each automaton transition can refer to the registers by requiring e.g. that the data value from a specific register be part of the transition's label or, alternatively, that the label feature a *fresh* data value (either not *currently* in the registers, or globally fresh — never seen before), which could then be stored in one of the registers. Formally, transition labels are pairs (t, d) where t is a tag drawn from a finite alphabet and $d \in \mathcal{D}$. We write $q \xrightarrow{t,i} q'$ to indicate that the transition is labelled with (t, d) where t is a tag and d is the data value currently stored in register i. Similarly, $q \xrightarrow{t,i^\bullet} q'$ describes transitions labelled with (t, d), where d can be any element of $\mathcal{D}$ that is currently *not* stored in any registers. Once the transition fires, d will be stored in register i. In contrast, $q \xrightarrow{t,i^\circledast} q'$ captures transitions with labels (t, d), where d ranges over all elements of $\mathcal{D}$ that have not yet been encountered by the automaton (i.e. d is "globally fresh").

As an example, consider the automaton in Figure 5 (left), which simulates a bounded "fresh" stack of size 2. By the latter we mean that the simulated stack can store up to two 2 distinct data values. The automaton starts from state p_1, with all its registers empty (erased). It can make a transition labelled with $(push, d_1)$, for any data value d_1, store it in register 1, and go to state p_2. From there, it can either pop the data value already stored in register 1 and go back to

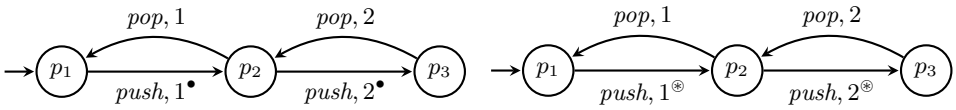

Fig. 1. Two implementations of a size-2 "fresh" stack.

p_1 (also erasing the register), or push another data value by making a transition $(push, d_2)$, for any data value $d_2 \neq d_1$, and go to state p_2. From there, it can pop the data value already stored in register 2 (and erase that register) and go to state p_1, and so on.

The automaton in Figure 1 (right) also simulates a 2-bounded fresh stack, but it does so using globally fresh transitions. That is, each $(push, d)$ transition made by the automaton is going to have a data value d that is different from all data values used before. We can thus see that bisimilarity of the two automata will fail after one pop: the LHS automaton will erase a data value from its registers and, consequently, will be able to push the same data value again later. The automaton on the RHS, though, will always be pushing globally fresh data values. In other words, the following trace:

$$(push, d_1)\,(pop, d_1)\,(push, d_1)$$

is permitted by the automaton on the LHS, but not by the RHS one.

2 Implementation

We have developed command-line tool for deciding language equivalence of this class of automata that is implemented in Haskell[4]. The two input automata (DRA) are specified using an XML file format which is parsed using the *xml-conduit* library [22] and internalised as a pair of Haskell records. At the same time, a dictionary of symbols is created so that user-specified strings (e.g. the names of states) can be mapped to integers for efficiency. The algorithm presented in [16] is, strictly speaking, given for computing bisimilarity of two states within the same automaton. Hence, our implementation first transforms the input, which is an instance of the language equivilance problem for two DRA, into an instance of the Bisimilarity problem, by constructing the disjoint union of the two automata.

Our algorithm exploits the observation that in the deterministic setting, language equivalence and bisimilarity are closely related, and it attempts to build a bisimulation relation incrementally. To avoid potential exponential blow-ups, we rely on symbolic representations based on (partial) permutations, which capture matches between register content in various configurations. The outline of the algorithm is presented below.

[4] The tool and its source are available at http://github.com/stersay/deq

```
1 R = R_init; Δ = {u_0};
2 while (Δ is not empty) do
3     u = Δ.get();
4     if u ∉ Gen(R)
5         if u fails single-step testing return NO;
6         Δ.add(succ-set(u));
7         Δ = Δ.add({u});
8         R = R updated with u;
9 return YES
```

The algorithm is similar in flavour to the classic Hopcroft-Karp algorithm for DFA [9], which maintains *sets of pairs of states*. In contrast, we work with four-tuples (q_1, σ, q_2, h), where q_1, q_2 are states, σ is a partial permutation and h is a parameter related to the number of registers. DEQ represents partial permutations using an implementation of immutable integer maps that is based on big endian patricia trees [13]. Most operations complete in amortized time $O(\mathsf{min}(n, W))$, where W is the number of bits of an integer. This is important because manipulating partial permutations through insertion and deletion is at the core of the innermost loop (line 8).

Starting from a four-tuple u_0, which represents the input equivalence problem, our implementation uses a finite sequence data type (based on 2-3 finger trees, for constant time access to either end) to represent a queue Δ (initialised to $\{u_0\}$) [19]. This sequence is used to store the four-tuples that still need to be scrutinised to establish the original equivalence. Since the total number of possible four-tuples is exponential in the number of registers (because one component is a partial permutation over the register indexes), the algorithm prescribes a sophisticated compact representation called a generating system. The generating system $\mathcal{R}$ represents the set of four-tuples that have already been analysed (its initial value $\mathcal{R}_{init}$ contains four-tuples with identical states and identity permutations).

Each iteration of the loop considers a four-tuple u taken from the queue (line 3): first we check if it is already generated by the generating system accumulated so far (line 4). Querying the generating system for membership requires deciding if a permutation belongs to a permutation group generated by $\mathcal{R}$. For this purpose, DEQ uses an implementation of the celebrated Schreier-Sims polynomial time group membership algorithm provided by the HaskellForMaths library [2].

If the four-tuple is already generated we move on to the next iteration. Otherwise we check if the configurations represented by u can withstand a one-step attack in the corresponding bisimulation game (line 5). If u fails single-step testing, the algorithm can immediately terminate and return NO. If u passes the tests, the algorithm generates a set $\mathsf{succ\text{-}set}(u)$ consisting of "successor four-tuples" that are added to Δ for future verification (lines 6-7). In this case, the generating system is extended to represent u as well (line 8). For efficiency reasons, DEQ fuses these two parts of the algorithm together. A collection of successors is computed by looping over all outgoing transitions relevant to u, and failing if any cannot be constructed. In what follows we refer to this as the inner loop of the algorithm. The successor four-tuples are then added to the queue and the generating system $\mathcal{R}$ is extended so that it generates u.

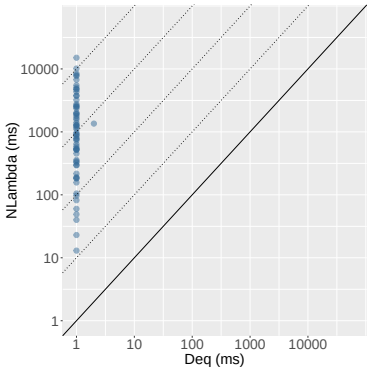

Fig. 2. DEQ and NLambda.

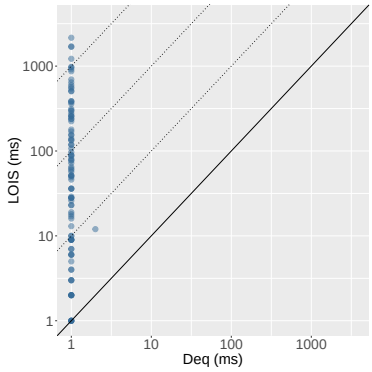

Fig. 3. DEQ and LOIS.

The polynomial-time complexity of the algorithm stems from several results in group theory and computational group theory. For instance, in order to check $u \notin \mathsf{Gen}(\mathcal{R})$, we need to perform group membership testing [7] and termination of the loop follows from the fact that subgroup chains of symmetric groups can only have linear length [3]. All the theoretical details related to the Algorithm can be found in [16].

3 Case Studies

In theory the worst-case performance of the algorithm is dominated by the complexity of permutation group membership testing, which is $O(n^5)$ for a straightforward implementation of the Schreier-Sims algorithm. In the following series of case studies we examine the performance of our implementation for particular families of inputs. In this way we can highlight certain interesting examples and also try to gauge the expected performance outside of the worst case, in particular where the group structure is quite simple.

All the experiments were carried out on the CAV'19 AE virtual machine, running on a Windows 10 host equipped with an Intel Core i7-8650U CPU running at 1.9GHz and 8GB of RAM. For the purposes of running the tools LOIS and RALib-EqCheck (see below), Z3 4.4.1 and OpenJDK 1.8.0_191 were installed on the virtual machine.

3.1 Stack data structure

There are natural situations in which the symbolic bisimulations are quite simple, such as when the number of tuples depends only on $|Q|$. This is the case for automata simulating a finite stack, which have been considered within the nominal automata learning framework of [14]. In this case study, we describe two families of automata simulating finite stacks, indexed by the stack size. Here and

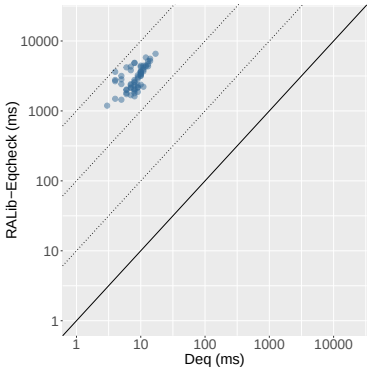

Fig. 4. DEQ and RALib-Eqcheck.

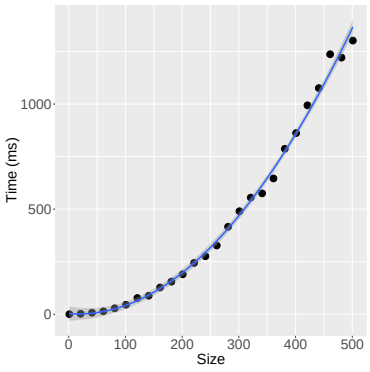

Fig. 5. Stacks scaling.

in the subsequent examples, all states will be assumed final.

LR-stack of size n
- Registers: $1, \ldots, n$
- States: $p_1, \ldots, p_{n+1}$ with p_1 initial
- Available registers at p_i: $[1, i-1]$
- Transitions: for $i \in [1, n]$:
  $p_i \xrightarrow{push, i^\bullet} p_{i+1}$,
  $p_{i+1} \xrightarrow{pop, i} p_i$.

RL-stack of size n
- Registers: $1, \ldots, n$
- States: $q_1, \ldots, q_{n+1}$ with q_1 initial
- Available registers at q_i: $[n-i+2, n]$
- Transitions: for $i \in [1, n]$:
  $q_i \xrightarrow{push, (n-i+1)^\bullet} q_{i+1}$,
  $q_{i+1} \xrightarrow{pop, n-i+1} q_i$

In both families of machines, the registers under their natural order are used in order to store the elements of a stack. Although LR-stacks push data into the registers from left to right and RL-stacks push data into the registers from right to left, this is an internal difference and LR-stacks of size n and RL-stacks of size n are language equivalent. In particular, the equivalence can be witnessed by a symbolic bisimulation in which there is exactly one tuple for each state of the two machines. Our algorithm will compute this bisimulation after $O(n)$ iterations, making $O(n)$ group membership queries.

Hence, we can use this example to examine whether there is any significant overhead to using a blackbox group membership algorithm in practice, when the groups are easy to describe. By considering the plot of running time against stack size in Figure 5, we conclude that any overhead is insignificant since growth remains roughly quadratic in n (a second degree polynomial has been fitted to the data using R's *lm* algorithm).

Tool comparison We can encode this family of examples using the frameworks of LOIS [12] and NLambda [11], and directly as a set of inputs to RALib-EqCheck[5]

[5] We used an unreleased implementation of the equivalence checking algorithm that was kindly communicated to us by F. Howar.

[5]. We use a vector in the former framework and a list in the latter framework in order to represent the sequence of registers. However, these other tools can only handle instances of relatively small size. This is not surprising, since they support classes of automata that are much more general. In contrast, the polynomial time algorithm implemented by DEQ is highly specialised to a specific subclass. Moreover, in the two frameworks, the automata cannot be specified directly. Instead, they need to be generated in the internal language, which in our case generates many inequality constraints. Hence, we have restricted our comparison to stacks of size at most 15. The scatter plots in Figures 2, 3 and 4 show the running times of the three implementations compared on all possible pairs of stack sizes[6]. The results show quite clearly that DEQ can determine (in)equivalence several orders of magnitude faster than the other tools.

3.2 Global simulating local freshness

In this case study we maintain a relatively simple group structure in the generating system, but introduce globally fresh transitions. Due to the use of global freshness, no other tools will process this example. When the history of a run is small, locally fresh transitions can be simulated by globally fresh transitions. This behaviour is captured by the following automata. Since tags are not needed in this example, we shall omit them from the definition.

GloLo* of size n

- Registers: $1, \ldots, n$
- States: $0, \ldots, n+1$ with 0 initial
- Available registers at state $i \in [1, n]$: $[1, i]$
- Available registers at state $n+1$: $[1, n]$
- Transitions: $i \xrightarrow{i^{\circledast}} i+1$, for $i \in [0, n-1]$, and $n \xrightarrow{n^\star} n+1$.

GloLo$^\bullet$ is an automaton that reads r globally fresh names and then a single locally fresh name. GloLo$^{\circledast}$ behaves similarly, except that the final name is required to be globally fresh. Let us distinguish the two sets of states of these two machines by labelling the former p_i and the latter q_i, for each state i.

The reason that p_0 is bisimilar to q_0 is that the machines will read exactly n globally fresh names before reaching state n, and all will be stored in the registers of both machines. Next, one machine reads a locally fresh name, whilst the other reads a globally fresh name, but because all of the history of the computation is stored in registers, local and global freshness coincide, and the configurations are bisimilar. Associated run times are plotted in Figure 6. It can be seen that the management of the history that is required by supporting global freshness adds a linear factor: the curve displayed is a cubic polynomial, fit using R's *lm* algorithm.

[6] The encoding used for the comparison with RALib-EqCheck was slightly modified to reflect certain structural constraints imposed by that tool. This alternative encoding is larger and hence runtimes are not comparable with the other two experiments. Note that the timing data for RALib-EqCheck contains JVM start-up time.

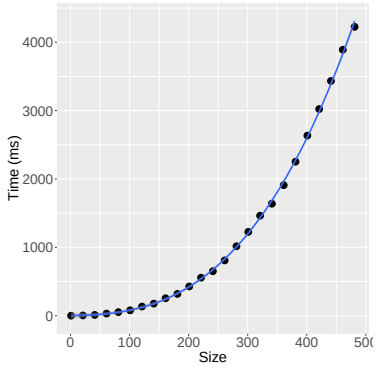

Fig. 6. GloLo scaling.

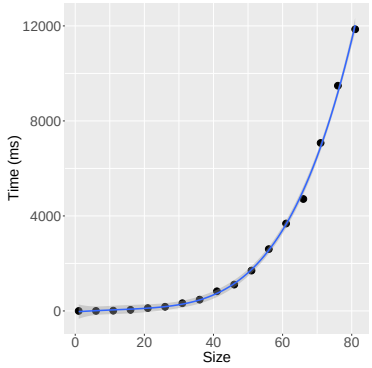

Fig. 7. Cpt scaling.

3.3 Complex partial permutations represented compactly

In the previous case, there was at most one partial permutation associated with every pair of states in the symbolic bisimulation. Here we consider a family of instances in which the collection of partial permutations occurring in constructed bisimulations can be quite complicated.

Cpt of size n

- Registers: $1, \ldots, n$
- States: $q_1, \ldots, q_{n+1}$
- Available registers at state i: $[1, i-1]$
- Transitions: $q_i \xrightarrow{t_1, i^\bullet} q_{i+1}$ and $q_i \xrightarrow{t_1, j} q_{i+1}$ for $i \in [1, n]$ and $j \in \mu(q_i)$.
  $q_{n+1} \xrightarrow{t_i, i^\bullet} q_{n+1}$ and $q_{n+1} \xrightarrow{t_i, j} q_{n+1}$, for $i, j \in [1, n]$.

where $\mu(q)$ is the set of available registers at state q. In this family, for size n we assume that the automaton runs over an alphabet with n distinct tags $\{t_1, \ldots, t_n\}$. We compare the language of such an automaton with the language of the corresponding automaton from a family *CptR*, which have essentially the same structure, but utilise the registers in the opposite order.

In this automaton, the registers are initially populated over the course of the first n states, with the Cpt automaton and the CptR automaton populating registers in the opposite order. Then, when both automata reach state q_n, there is the possibility to store a fresh letter in any register or read from any register, and return to state q_n. Consequently, the correspondence between these states in the Cpt and CptR automata can become complex. However, since all transitions are available to both automata, they will nevertheless be bisimilar, and hence accept the same language. Running times are plotted in Figure 7; the curve displayed is a fourth-degree polynomial.

References

1. F. Aarts, P. Fiterau-Brostean, H. Kuppens, and F. W. Vaandrager. Learning register automata with fresh value generation. In *Proceedings of ICTAC*, volume 9399 of *Lecture Notes in Computer Science*, pages 165–183. Springer, 2015.
2. David Amos. The HaskellForMaths package. http://hackage.haskell.org/package/HaskellForMaths.
3. L. Babai. On the length of subgroup chains in the symmetric group. *Communications in Algebra*, 14(9):1729–1736, 1986.
4. B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A robust class of data languages and an application to learning. *Logical Methods in Computer Science*, 10(4), 2014.
5. S. Cassel, F. Howar, and B. Jonsson. Ralib: A learnlib extension for inferring efsms. In *Proceedings of DIFTS*, 2015.
6. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.
7. M. L. Furst, J. E. Hopcroft, and E. M. Luks. Polynomial-time algorithms for permutation groups. In *Proceedings of FOCS*, pages 36–41. IEEE Computer Society, 1980.
8. R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In *Proceedings of TACAS*, LNCS. Springer, 2013.
9. J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell University, 1971.
10. M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.
11. B. Klin and M. Szynwelski. SMT solving for functional programming over infinite structures. In *Proceedings of MSFP*, volume 207 of *EPTCS*, pages 57–75, 2016.
12. E. Kopczyński and S. Toruńczyk. LOIS: syntax and semantics. In *Proceedings of POPL*, pages 586–598. ACM, 2017.
13. Daan Leijen and Andriy Palmarchuk. Data.intmap library. http://hackage.haskell.org/package/containers.
14. J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In *Proceedings of POPL*, pages 613–625. ACM, 2017.
15. A. S. Murawski, S. J. Ramsay, and N. Tzevelekos. A contextual equivalence checker for IMJ*. In *Proceedings of ATVA*, LNCS, pages 234–240. Springer, 2015.
16. A. S. Murawski, S. J. Ramsay, and N. Tzevelekos. Polynomial-time equivalence testing for deterministic fresh-register automata. In *43rd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 117 of *LIPIcs*, pages 72:1–72:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
17. A. S. Murawski and N. Tzevelekos. Algorithmic nominal game semantics. In *Proceedings of ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 419–438. Springer-Verlag, 2011.
18. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.
19. Ross Paterson, Louis Wasserman, Bertram Felgenhaur, David Feuer, and Milan Straka. Data.sequence library. http://hackage.haskell.org/package/containers.
20. H. Sakamoto. *Studies on the Learnability of Formal Languages via Queries*. PhD thesis, Kyushu University, 1998.
21. T. Schwentick. Automata for XML - A survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.

22. Michael Snoyman and Aristid Breitkreuz. The xml-conduit package. http://hackage.haskell.org/package/xml-conduit.
23. N. Tzevelekos. Fresh-register automata. In *Proceedings of POPL*, pages 295–306. ACM Press, 2011.