BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2024
http://excel.fit.vutbr.cz

Formal Models for Data Languages

Jan Vašák*

Abstract
Data languages are commonly used when working with infinite alphabets formally. This work explores theoretical properties of some automata models over data words, and also shows a practical implementation of one of the models as a regex matcher.

*xvasak01@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Finite automata are a staple formal model in computer science. They, however, are designed to work on finite alphabets and are often not sufficient when one wants a model to work with infinite sets (e.g. integers) or very large finite sets (e.g. Unicode symbols). Data words are commonly used with extensions of finite automata that *are* designed to work with infinite sets. A data word is a sequence consisting of pairs of an alphabet symbol from a finite alphabet, and a data value from a countably infinite data domain.

2. Automata Models

Register Automata. A *register automaton* [1] (RA) extends a finite automaton (FA) by adding a finite set of *registers*, each of which can store a up to one data value. On transitions, registers can be tested for equality or non-equality. Each register can then be updated by a new data value (of another register, or current input data value *in*). See an example of an RA in Figure 1, accepting the language of words whose last data value appeared previously in the word.

Register Set Automata A *register set automaton* [2] (RsA) is the same as an RA, except its registers, sometimes called *set-registers*, hold a set of data values instead of just one. On transitions, registers can be tested for (non-)membership, and register updates are denoted by a set that can contain registers and *in*. They also have an extension RsA^{rm} that also allows the removal of *in* on transitions. See an example of a deterministic RsA, accepting the same language as a the non-deterministic RA from Figure 1, in Figure 2.

History Register Automata A *history register automaton* [3] (HRA) also has a set of set-registers. On a transition, two sets of registers are specified, R_g, R_{up}. The transition checks that *in* is stored *exactly* in the registers in R_g (i.e., is stored in all the registers in R_g, and in none of the other registers) to be enabled. After the transition is taken, it updates the register contents such that *in* is stored exactly in the registers of R_{up}. HRAs can also empty their registers on special reset transitions. See an example of a deterministic HRA in Figure 2. It also accepts the same language as the RA in Figure 1.

Streaming Data-String Tranducers A *streaming data-string transducer* [4] (SDST) is a tranducer model over data words. It has a set of *data variables*, which are essentially RA-style registers that can also be checked for inequality (for an ordered data domain). To generate an output, they have a set of *data-string variables*, each storing a data word. These can be updated by a word constructed from data and data string variables. The SDST has an output function that outputs a data word after reading the entire input data word. See Figure 4 for an example SDST that reverses the input data word.

3. Extending SDSTs

Regular SDSTs can represent (imperative and functional) single-pass list-processing programs [4], and can thus be used in formal analysis and verification of these programs. E.g., SDSTs can represent a program that reverses the input list (see Figure 4).

SDSTs with set-registers. We extend SDSTs by equipping them with a finite set of set-registers (SDST^{set}).

The set-registers can be tested for (non-)membership of *in* on transitions, and can be updated by adding or removing *in*. Another difference is that an SDST^{set} does not allow inequality tests of data variables, only (non-)equality tests.

Theorem 1. *The functional equivalence problem for SDST^{set} is decidable.*

Practically, Theorem 1 gives us a decidable way to determine whether two SDST^{set}s are semantically equivalent, which is important in terms of formal analysis and verification. SDST^{set}s can represent a class of single-pass list-processing programs with a set type data-structure available for them to use. An example program that can be represented by an SDST^{set}, but not by an SDST is a program that removes duplicates from a list (see Figure 5).

4. Relating RsAs and HRAs

Because HRAs can remove the input symbol from registers, we will be comparing them with RsA^{rm} in terms of their expressive powers. The first result is that RsAs generalize HRAs (and it holds for their deterministic variants as well).

Proposition 2. *HRA $\subseteq$ RsA^{rm}*

Corollary 3. *DHRA $\subseteq$ DRsA^{rm}*

The other direction of Proposition 2 is left as an open problem. However, we have found a language differentiating their deterministic variants.

Proposition 4. *DHRA $\subsetneq$ DRsA^{rm}.*

5. RsA Emptines Parametrization

The emptiness problem is known to be Ackermann-complete for both RsA, and RsA^{rm} when the number of registers is part of the input [2]. We show some complexity results of both RsA variants with a fixed number of registers.

Proposition 5. *The emptiness problem for RsA$_1$ is NL-complete.*

Proposition 6. *The emptiness problem for RsA$_1^{rm}$ is NL-complete.*

Proposition 7. *The emptiness problems for RsA$_n^{rm}$ is in $\mathbf{F}_{2^n+1}$ for $n \geq 2$.*

Corollary 8. *The emptiness problems for RsA$_n$ is in $\mathbf{F}_{2^n+1}$ for $n \geq 2$.*

6. RsA-based Regex Matching

RAs can be represent a class of regular expressions with back-references. However, RAs are not determinisable in general, and thus are not useful for effective matching of regexes. There does, however, exist an algorithm that can determinise a class (which was extended in this work) of RA to DRsA [2].

The implemented matcher uses the regex parser of the Python module re [5]. It then constructs an RA from the created syntax tree. With the above-mentioned algorithm, the RA is determinised into a DRsA, which is then used for matching (if the RA and DRsA constructions were successful.

6.1 Experiments

Using the ReDOS (regex denial of service) attack generators rxxr2 [6] and rescue [7], vulnerable regexes with back-references were extracted from a set of regexes used in practice. The generated combinations of regex and attack string were then run on the RsA matcher, re, the pcre2 library [8], and GNU grep [9]. Table 1 shows the number of timeouts for each tool out of the regexes that were successfully determinised (note that the RsA matcher timeouts were caused by RA determinisation, not actual matching). Figures 7 and 8 show scatterplots of matching times of pcre2 and re respectively against the RsA matcher. Though grep was not defeated by the generators, we show that it is possible with a difficult hand-crafted regex and input in Figure 6.

7. Conclusion and Future Work

The complexity of RsA emptiness was closer specified based on the number of registers. The expressive powers of RsAs and HRAs were compared to one another. An extension of SDST that allows them to use set-registers was presented, and it was shown that its functional equivalence is decidable. An RsA-based regex matcher was implemented and compared to existing matchers, with the results showing that an RsA-based matcher could be used for safe matching of regexes with back-references.

In the future, we hope to further explore SDST^{set} and see if we can increase its expressiveness (by, e.g., keeping the order on the data domain or allowing register resets), while keeping functional equivalence decidable. We would also like to extend the class of regexes with back-references that can be represented by DRsA, and start work on a ReDOS generator focusing on back-references.

Acknowledgements

I would like to thank my supervisor, Ing. Ondřej Lengál Ph.D, for providing help and guidance during my bachelor's thesis that this submission is based on.

References

[1] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[2] Sabína Gulčíková and Ondřej Lengál. Register set automata (technical report). 2022.

[3] Radu Grigore and Nikos Tzevelekos. History-register automata. *Logical Methods in Computer Science*, Volume 12, Issue 1, mar 2016.

[4] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 599–610, New York, NY, USA, 2011. Association for Computing Machinery.

[5] Python Software Foundation. *Python Standard Library - re Module*, 2023. Available at: https://docs.python.org/3/library/re.html.

[6] Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[7] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. Rescue: crafting regular expression dos attacks. In *ASE'18*, pages 225–235. ACM, 2018.

[8] P. Hazel. *Perl-compatible Regular Expressions*, 2022. Available at: https://www.pcre.org.

[9] Inc. Free Software Foundation. *GNU grep 3.6*. Online, 2021. Available at: https://git.savannah.gnu.org/cgit/grep.git.