Software Impacts 6 (2020) 100027

Contents lists available at ScienceDirect

Software Impacts

journal homepage: www.journals.elsevier.com/software-impacts

Original software publication

RE2C: A lexer generator based on lookahead-TDFA

Ulya Trofimovich

Not affiliated to an organization

ARTICLE INFO

Keywords:
Lexical analysis
Regular expressions
Finite automata

ABSTRACT

RE2C is a regular expression compiler: it transforms regular expressions into finite state machines and encodes them as programs in the target language. At the core of RE2C is the lookahead-TDFA algorithm that allows it to perform fast and lightweight submatch extraction. This article describes the algorithm used in RE2C and gives an example of TDFA construction.

Code metadata

Current code version	2.0
Permanent link to code/repository used for this code version	https://github.com/SoftwareImpacts/SIMPAC-2020-29
Permanent link to reproducible capsule	https://codeocean.com/capsule/6014695/tree/v1
Legal Code License	Public domain
Code versioning system used	Git
Software code languages, tools, and services used	C++, Bison, RE2C (self-hosting)
Compilation requirements, operating environments & dependencies	OS: Linux, BSD, Nix/Guix, GNU Hurd, OS X, Windows, etc. Dependencies: C++ compiler, Bash, CMake or Autotools. Optional Bison and Docutils.
Link to developer documentation/manual	https://re2c.org
Support email for questions	re2c-general@lists.sourceforge.net

1. Introduction

Regular expression engines can be divided in two categories: run-time libraries and lexer generators. Run-time libraries perform interpretation or just-in-time compilation of regular expressions. They use a variety of algorithms ranging from recursive backtracking to automata, string searching, or some combination of the above. Lexer generators, on the other hand, perform ahead-of-time compilation. They use algorithms based on deterministic finite automata (DFA) and spend considerable time on compilation and optimization in order to emit better code. Consequently, lexer generators usually do not support features that cannot be implemented on vanilla DFA. One such feature is submatch extraction — the ability to find the correspondence between parts of the regular expression and parts of the input string. Submatch extraction is a special case of the parsing problem: in addition to solving the recognition problem it has to find the derivation of the input string in the grammar defined by the regular expression. Unlike full parsing, submatch extraction needs only a partial derivation. Therefore it would be wasteful to perform full parsing, and a more specialized algorithm is needed that has overhead proportional to submatch detalization. For an optimizing lexer generator like RE2C [1,2] it is important that the generated code is at least as fast and memory-efficient as hand-written code, and there is zero overhead if submatch extraction is not used.

2. Tagged DFA

The above requirements place tight constraints on the submatch extraction algorithm: it should be a one-pass DFA-based algorithm that works in constant memory independent of the input length. Such an algorithm was invented by Ville Laurikari [3]. It works as follows. First, the regular expression is converted to a nondeterministic finite automaton with tagged transitions (TNFA). Tags are submatch markers that can be placed anywhere in the regular expression; for example, a

The code (and data) in this article has been certified as Reproducible by Code Ocean: (https://codeocean.com/). More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

E-mail address: skvadrik@gmail.com.

https://doi.org/10.1016/j.simpa.2020.100027
Received 28 July 2020; Received in revised form 29 July 2020; Accepted 4 August 2020

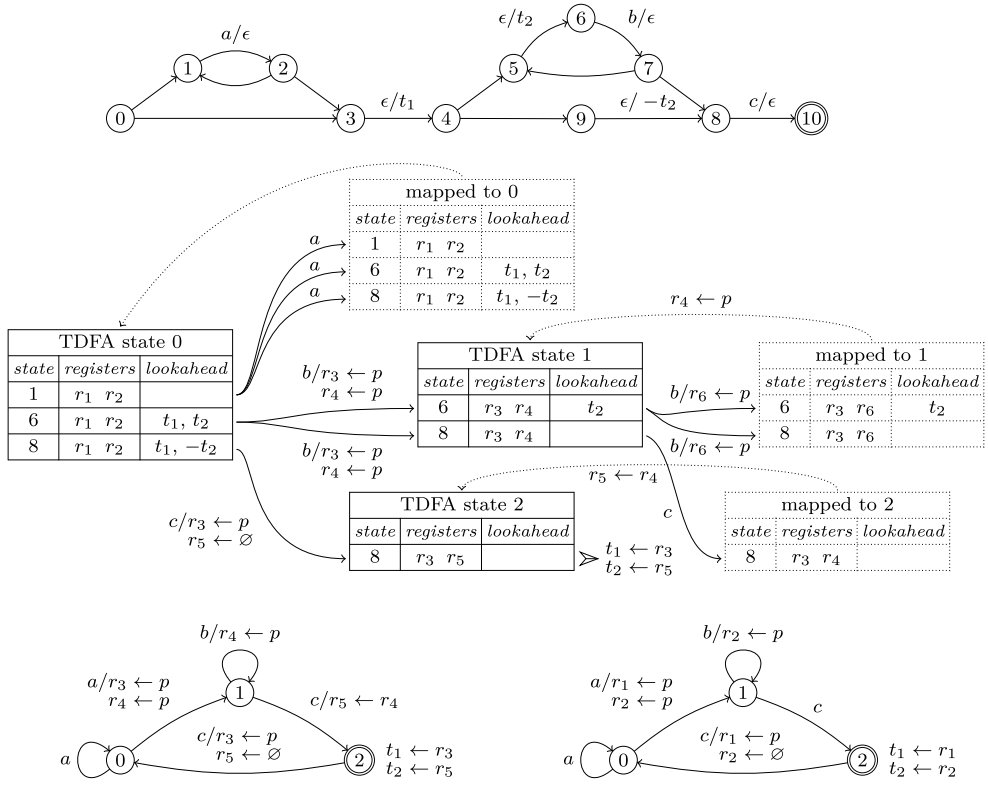

Fig. 1. Lookahead-TDFA construction for regular expression $a^* t_1(t_2\, b)^* c$ with tags t_1 and t_2 that matches strings $a \dots ab \dots bc$ (tag t_1 marks the boundary between as and bs, and tag t_2 marks the last b). Top: TNFA (ϵ/ϵ labels are omitted, and negative tags indicate no-match). Middle: lookahead-TDFA under construction with state-sets, register matrices and lookahead tags (p is the current position, and $\varnothing$ is no-match). Bottom left: lookahead-TDFA after construction. Bottom right: lookahead-TDFA after optimization (the number of registers is minimized and some register operations are removed).

capturing group can be represented with a pair of tags for the opening and closing parentheses. TNFA is in essence a nondeterministic finite state transducer that rewrites symbolic strings into tagged strings. The most important part of Laurikari algorithm is TNFA determinization. Its basic principle is the same as in the powerset construction that converts NFA to DFA: the NFA is simulated on all possible input strings, maintaining a set of states at each step. If the current state-set is new, it becomes a new DFA state, otherwise it is mapped to an existing DFA state. In the Laurikari algorithm state-sets are augmented with tag information: each TNFA substate has an associated vector of registers that store tag values, so that the whole state-set is a matrix indexed by TNFA states and tags. Registers are needed because different substates are reached by different TNFA paths, and the tags along one path may disagree with tags along another path. If a tag is updated on a TNFA path, its value is stored into a new register. The augmented state-sets cannot be mapped in the same way as ordinary DFA states, because a pair of state-sets may have identical TNFA substates, but different registers. The key insight is that mapping of such state-sets is still possible if there is a bijection between their registers: the bijective transformation can be encoded in the form of register reordering operations on TDFA transitions. After determinization all the information in TDFA state-sets is erased, and the resulting TDFA is like ordinary DFA extended with a fixed number of registers and operations on transitions that update and reorder tag values stored in registers. Algorithms like minimization are applicable to TDFA, but they need to differentiate between transitions with different register operations.

3. Lookahead TDFA

One improvement to Laurikari algorithm that allows to greatly reduce the number of register operations is the use of the lookahead symbol [4]. The idea is to delay the application of register operations until the lookahead symbol is known, and attach the operations to the outgoing transition on that symbol instead of the preceding incoming transition. The insight is that some of TNFA paths that have reached the current TDFA state are canceled after the application of lookahead symbol, and all register operations associated with these paths can be canceled as well. In other words, delaying for one step allows one to split register operations on the lookahead symbol, and instead of doing all of them, do only the relevant part. This requires a modification to the underlying TDFA state-sets: in addition to registers, each substate needs to have an associated list of lookahead tags. The additional information is erased after determinization, and the resulting lookahead-TDFA is faster than ordinary TDFA and better suited for further transformations like minimization and register allocation. On the whole, this idea is similar to the improvement of LALR parsers over LR parsers: the use of lookahead information in states greatly reduces the number of conflicts (for TDFA a conflict is not an error, but it means extra registers and register operations). An example of lookahead-TDFA construction can be seen on Fig. 1.

4. Ambiguity resolution

One of the challenges that makes parsing harder than recognition is ambiguity: there may be several ways to parse the input string. Which way to prefer is usually defined by a disambiguation policy.

TDFA algorithm can be parameterized over different policies; for example, RE2C supports both the Perl leftmost-greedy policy and the POSIX longest-match policy [5]. Disambiguation happens at the time of determinization (in ϵ-closure construction), and the resulting TDFA does not perform any disambiguation at run-time: ambiguity-resolving decisions are embedded into its structure.

5. Impact

RE2C is a lexer generator of choice for projects that need *fast* lexical analyzers. It has a flexible user interface that allows one to customize the generated code for a particular environment and input model. Fast and lightweight submatch extraction is yet another feature that makes RE2C a good alternative to other lexer generators. Notable projects that use RE2C are the following:

- Ninja, a build system with a focus on speed. [6] Ninja is used in a great number of open-source projects, and it is a necessary building block in many operating systems and platforms.
- PHP, a popular general-purpose scripting language. [7]
- BRL-CAD, a constructive solid geometry solid modeling computer-aided design system. [8]
- STEPCode, an implementation of ISO 10303 standard. [9]
- Apache SpamAssassin, a program for e-mail spam filtering. [10]
- Yasm, the Modular Assembler Project. [11]
- Wake, the SiFive build tool. [12]

RE2C has a permissive license, which allows it to be used in proprietary software as well. The project has been ported to many operating systems; according to Repology [13], it is packaged in more than 60 distributions. Besides being a useful practical tool, RE2C is a research playground for the development of new algorithms in the field of automata and formal languages.

6. Conclusion and future work

Submatch extraction algorithm implemented in RE2C is both an important theoretical development and a useful practical improvement. It allows one to program lexical analyzers capable of submatch extraction without resorting to manual post-processing or the use of string-searching functions. The overhead on submatch extraction is proportional to submatch detalization: there is zero overhead if no submatch information is needed, and even for large submatch-heavy regular expressions like RFC-compliant URI and HTTP parsers the overhead is modest (about 1.25x compared to simple recognition on DFA [4]). Future work may relate TDFA to other parsing automata, such as DSST [14] and sta-DFA [15], and investigate the possibility of using tagged counter automata for counted repetition [16].

CRediT authorship contribution statement

Ulya Trofimovich: Conceptualization, Software, Validation, Formal analysis, Writing - original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

I want to thank my parents Vladimir and Elina, my friend and fellow researcher Angelo Borsotti, my school math teachers Tatyana Leonidovna and Demian Vladimirovich, and, most of all, Sergei.

References

[1] RE2C, a lexer generator for C, C++ and Go. Website: https://re2c.org, source code: https://github.com/skvadrik/re2c.
[2] Peter Bumbulis, Donald D. Cowan, RE2C: A more versatile scanner generator, ACM Lett. Programm. Lang. Syst. (LOPLAS) 2 (1-4) (1993) 70–84.
[3] Ville Laurikari, NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions, in: Proceedings Seventh International Symposium on String Processing and Information Retrieval, 2000. SPIRE 2000, pp. 181–187, URL: http://laurikari.net/ville/spire2000-tnfa.pdf.
[4] Ulya Trofimovich, Tagged Deterministic Finite Automata with Lookahead, 2017, cs.FL.
[5] Angelo Borsotti, Ulya Trofimovich, Efficient POSIX submatch extraction on NFA, preprint, 2019, URL: https://re2c.org/2019_borsotti_trofimovich_efficient_posix_submatch_extraction_on_nfa.pdf.
[6] Ninja build system, URL: https://ninja-build.org, build files that use RE2C: https://ninja-build.org/build.ninja.html.
[7] PHP Internals Book, chapter *Building PHP*, URL: http://www.phpinternalsbook.com/php7/build_system/building_php.html.
[8] BRL-CAD: Tools, URL: http://sourceforge.net/p/brlcad/code/HEAD/tree/brlcad/trunk/misc/tools/re2c.
[9] STEPCode: Build Process, URL: https://stepcode.github.io/docs/build_process.
[10] SpamAssassin (sa-compile), URL: https://spamassassin.apache.org/full/3.2.x/doc/sa-compile.html.
[11] Yasm, URL: https://yasm.tortall.net.
[12] Wake, URL: https://github.com/sifive/wake.
[13] Repology: RE2C. URL: https://repology.org/project/re2c/information.
[14] Niels Bjørn Bugge Grathwohl, Parsing with Regular Expressions & Extensions to Kleene Algebra, DIKU, University of Copenhagen, 2015.
[15] Mohammad Imran Chowdhury, StaDFA: An Efficient Subexpression Matching Method (Master thesis), Florida State University, 2018.
[16] Michela Becchi, Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation, Washington University In St. Louis, School of Engineering and Applied Science, 2009.