UNIWERSYTET
IM. ADAMA MICKIEWICZA
W POZNANIU

Wydział Matematyki i Informatyki

Bohdan Bondar, Marcin Jałoński, Aleksander Mendoza-Drosik
Numer albumu:

Solomonoff - kompilator transduktorów skończenie stanowych

Solomonoff - Finite state tranducer compiler

Praca inżynierska na kierunku **informatyka**
napisana pod opieką
dr Bartłomieja Przybylskiego

Poznań, luty 2021

Poznań, 9 listopada 2020 r.

Oświadczenie

Ja, niżej podpisana **Bohdan Bondar, Marcin Jałoński, Aleksander Mendoza-Drosik**, studentka Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt. *Solomonoff - kompilator transduktorów skończenie stanowych* napisałam samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałam z pomocy innych osób, a w szczególności nie zlecałam opracowania rozprawy lub jej części innym osobom, ani nie odpisywałam tej rozprawy lub jej części od innych osób. Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej. Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[TAK/TAK/TAK] wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[TAK/TAK/TAK] wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

Streszczenie

This project focuses on research in the field of automata theory and inductive inference. While many existing libraries already provide support for general purpose automata and implement various related algorithms, this project takes a slightly different approach. The primary tool for working with the library, is through domain specific language. Most of the things can be done without writing even a single line of Java code.

Compilation of regular expressions is very efficient thanks to Gloshkov's construction. Hence all operations of concatenation, union, Kleene closure (including $*$, $+$, ?) are constant-time operations. Moreover, the automata will have only as many states as there are symbols in regular expression.

All automata are nondeterministic functional subsequential weighted transducers. The primary semiring of weights is arctic lexicographic semiring (more options will come in the future). Compiler always enforces functionality (that is, at most one output can be printed for each input) through an efficient transducer squaring algorithm (time complexity is quadratic). Moreover, all transitions are ranged - that is they don't accept just a single symbol but instead they span entire range of symbols. Therefore expressions like . translate to just one single transition. Glushkov's construction gives us guarantee that there are only as many transitions as there are symbols in regular expression (each range $[a - z]$ counts as one).

All regular expressions are strongly typed. The type system is polymorphic and (unlike in most Turing-complete languages), the typechecking is not done through unification algorithm, but rather the language inclusion. All types are regular expressions themselves as well, although it is required that they are deterministic (that is, the automaton produced with Glushkov's construction is deterministic). This way, language inclusion can be checked by performing product of automata (quadratic time-complexity). In some places (explained below) also nondeterministic language inclusion is checked, but unfortunately it can only be done with subset construction (exponential time complexity). Hence,

user is advised to use nondeterministic typechecking only when absolutely necessary.

All automata are very small thanks to nondeterministic pseudo-minimisation. Unlike most other libraries that implement minimisation through construction of minimal DFA, here we actually perform pseudo-minimisation on nondeterministic transducers. The algorithm uses heuristics inspired by Brzozowski's construction and Kameda-Weiner's NFA minimisation. Unfortunately performing full minimisation on NFA is a hard problem, which requires exponential complexity. Our algorithm is $O(nlogn)$ on average (and $O(n^2logn)$ in a very unlikely pessimistic case) and attempts to reduce size of automaton as much as possible, without actually searching for the smallest one. Note that not always finding the smallest nondeterministic transducer, should not be a problem, because the NFA are often much smaller than even the smallest DFA. Moreover, thanks to glushkov's construction, all compiled automata are often in practice smaller than minimal DFA even without pseudo-minimisating them.

Evaluation is very efficient thanks to guarantees of lexicographic weights. In pessimistic case evaluation is quadratic, but after performing pseudo-minimisation, it is in practice often close to being linear (optimistic case being deterministic automata, whose evaluation has linear time complexity).

Transducer composition is lazy because otherwise it would pose the danger of exponentially exploding size of automata (composition of two transducers is quadratic, but if composition is used multiple times in a regular expression then, that would lead to exponential number of states in worst case). However, making composition lazy has some advantages - this compiler allows for invoking external functions written in Java. Hence you can ad-hoc mix regular expressions with custom Java functions. If really necessary, there is also a function for explicitly performing composition in non-lazy manner, but it should be used with caution and only applied when automata are reasonably small.

External functions can be called to add even more features. For instance, instead of making import some.other.module a keyword (like in most other languages), here import!('some/other/module.mealy') is an external function. It's possible to read automata in various formats such as ATT, DOT and compressed binary.

Inductive inference/machine learning can be extensively used with ease. At the moment all inference functions are provided by LearnLib (Solomonoff is compatible with LearnLib and AutomataLib). More algorithms, specific to transducers will be added soon.

Solomonoff implementation is small and generic. It takes up roughly 10-15 classes and many algorithms are written in a clean reusable manner. Because

the main way of interaction with the library is via its domain specific language, most of the underlying implementation is free to be changed and reworked in drastic ways at any time. The Java API accessible to end-users is very minimalist just like Java's Pattern.compile (you don't see things like Pattern.union or Pattern.kleeneClosure). This means that our library has a lot more room for refactoring, simplifying and optimising the implementation.

The primary philosophy used in implementing this library is the top-down approach and features are added conservatively in a well thought-through manner. No features will be added ad-hoc. Everything is meant to fit well together and follow some greater design strategy. For comparison, consider the difference between OpenFst and Solomonoff.

— OpenFst has Matcher that was meant to compactify ranges. In Solomonoff all transitions are ranged and follow the theory of (S,k)-automata. They are well integrated with regular expressions and Glushkov's construction. They allow for more efficient squaring and subset construction. Instead of being an ad-hoc feature, they are well integrated everywhere.
— OpenFst has no built-in support for regular expression and it was added only later in form of Thrax grammars, that aren't much more than another API for calling library functions. In Solomonoff the regular expressions are the library. Instead of having separate procedures for union, concatenation and Kleene closure, there is only one procedure that takes arbitrary regular expression and compiles it in batch. This way everything works much faster, doesn't lead to introduction of any ϵ-transitions (that's right! In Solomonoff, ϵ-transitions aren't even implemented, because they were never needed thanks to Glushkov's construction). This leads to significant differences in performance. Compiling a dictionary txt file that has several thousands of entries, takes hours for OpenFst but only 2-5 seconds for Solomonoff. That comes out-of-the-box without hand-optimising any code.

For embedded systems Java might not be the ideal language of choice, hence we (will soon) provide C backend. The compiler itself will remain written purely in Java, but there will be alternative C runtime capable of running all automata as well. Moreover, for places where resources are really tight, Solomonoff will offer direct compilation of transducers into C code (similarly to how Flex and Bison or other parser generators work, except that Solomonoff will generate transducers instead of parsers).

Compilation of UNIX regexes and Thrax grammars will be soon supported as well. Hence Solomonoff will try to compete with Google's RE2. Solomonoff will do all this via active learning. This approach has numerous advantages.

— First there is no need to explicitly write and converters. You only give Solomonoff some Java function that is of type String someFunction(String input)... and Solomonoff will treat it as oracle for grammatical inference. For example someFunction might call input.replaceAll("some|complicated(regex)*","substitution") (even multiple times) and Solomonoff will learn and produce some transducer. This way lookahead's and lookbehind's will be supported (so it can do more than Google's RE2 library).
— Second it will not only convert but also hugely optimise all regexes.
— Third it will be very generic. It could in fact learn any Java function, as long as it's functionality can be expressed by a transducer (context free or context sensitive transductions are not supported). Hence Solomonoff might be a perfect tool for simplify your ancient codebase of spaghetti regexes, written by "some guy who no longer works here".

Słowa kluczowe: klasa

Abstract

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Keywords: klasa

Tu możesz umieścić swoją dedykację.

Spis treści

Rozdział 1

Treść mojej pracy

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.